Course Information

This handout describes basic course information and policies. Most of the sections will be useful throughout the course. The main items to pay attention to **NOW** are:

1. Please note the dates of the quizzes on the course calendar and plan trips accordingly. Notify the staff if you have an unavoidable conflict, e.g., an exam in another class.

2. Please note the collaboration policy for homeworks.

3. Please note the grading policy.

1 Staff

Lecturers:	Silvio Micali `silvio@csail.mit.edu`	32-G644	617-253-5949
	Constantinos (Costis) Daskalakis `costis@csail.mit.edu`	32-G694	617-253-9643
Teaching Assistants:	Alan Deckelbaum `deckel@mit.edu`	32-G604	
	Rafael Oliveira `rmendes@mit.edu`	24-321	
	Shaunak Kishore `skishore@mit.edu`		
	Dragos Ionescu `dionescu@mit.edu`		
	Jeff Wu `jeffwu@mit.edu`		
World Wide Web:	`http://courses.csail.mit.edu/6.006/spring12`		
Email:	`6.006-staff@mit.edu`		

2 Prerequisites

A strong understanding of programming in Python and a solid background in discrete mathematics are necessary prerequisites to this course.

You are expected to have taken 6.01 *Introduction to EECS I* and 6.042J/18.062J *Mathematics for Computer Science*, and received a grade of C or higher in both classes. If you do not meet these requirements, you must talk to a TA or a professor before taking the course.

3 Course 6 requirements

Under the new curriculum, 6.006 serves as a Foundational Computer Science course. It is a prerequisite for 6.046, which serves as a Computer Science theory header.

4 Lectures

Lectures will be held in Room 32-123 from 11:00 A.M. to 12:00 P.M. ET on Tuesdays and Thursdays.

You are responsible for material presented in lectures, including oral comments made by the lecturer.

5 Recitations

One-hour recitations will be held on Wednesdays and Fridays. Please go to the section assigned to you by the registrar. If you have a conflict, please email staff (see email above) and we will try to accomodate you.

You are responsible for material presented in recitation. Attendance in recitation has been well correlated in the past with exam performance. Recitations also give you a more intimate opportunity to ask questions and interact with the course staff.

6 Problem sets

Six problem sets will be assigned during the semester. The course calendar, available from the course webpage, shows the tentative schedule of assignments, and due dates. The actual due date will always be on the problem set itself.

A large portion of each problem set will be a coding assignment to be done in Python. Any code for submission must uploaded to the class website, and the *final* submission will be graded.

- Late homework will generally not be accepted. If there are extenuating circumstances, you should make *prior* arrangements with your recitation instructor.

  An excuse from the Dean's Office will be required if prior arrangements have not been made.

- We require problem set solutions (other than code) to be written in LaTeX using the template provided on the website. They should be uploaded to the class website in PDF form by 11:59PM of the due date.

Be sure to fill in the "Collaborators" section of each problem. If you solved the problem alone, write "none".

7 Exams

There will be two evening quizzes, whose dates will be updated on this handout and one the website soon. We will announce the dates for the quizzes soon.

There will also be a final exam during finals week.

8 Grading policy

The final grade will be primarily based on 6 problem sets, two quizzes, and a final. The problem sets will together be worth 30 points, each quiz will be 20 points, and the final exam 30 points.

The specifics of this grading policy are subject to change at the discretion of the course staff.

Grading of Code

Code will be graded for correctness and for the algorithm used.

Correctness You will be given a public set of unit tests to test your code. For grading purposes, we may run your code against a more thorough private set of unit tests. Your code must run within the time allotted (which will vary by assignment).

Algorithm Your code must come well-commented describing the algorithm used. Your code must be readable so the TAs will believe that your code does what it claims to do. Your algorithm should be efficient.

9 Collaboration policy

The goal of homework is to give you practice in mastering the course material. Consequently, you are encouraged to collaborate on problem sets. In fact, students who form study groups generally do better on exams than do students who work alone. If you do work in a study group, however, you owe it to yourself and your group to be prepared for your study group meeting. Specifically, you should spend at least 30–45 minutes trying to solve each problem beforehand. If your group is unable to solve a problem, talk to other groups or ask your recitation instructor.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You are asked on problem sets to identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." If you obtain a solution through research (e.g., on the web), acknowledge your source, but write up the solution in your own words. **It is a violation of this policy to submit a problem solution that you cannot orally explain to a member of the course staff.**

Code you submit must also be written by yourself. You may receive help from your classmates during debugging. Don't spend hours trying to debug a problem in your code before asking for help. However, regardless of who is helping you, only you are allowed to make changes to your code. **A suite of algorithms will be run to detect plagiarism in code.**

No other 6.006 student may use your solutions; this includes your writing, code, tests, documentation, etc. It is a violation of the 6.006 collaboration policy to permit anyone other than 6.006 staff and yourself read-access to the location where you keep your code.

Plagiarism and other anti-intellectual behavior cannot be tolerated in any academic environment that prides itself on individual accomplishment. If you have any questions about the collaboration policy, or if you feel that you may have violated the policy, please talk to one of the course staff. Although the course staff is obligated to deal with cheating appropriately, we are more understanding and lenient if we find out from the transgressor himself or herself rather than from a third party.

10 Textbook

The primary written reference for the course is the Second Edition of the textbook *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein.

The textbook can be obtained from the MIT Coop, the MIT Press Bookstore , and at various other local and online bookstores.

We also recommend *Problem Solving With Algorithms And Data Structures Using Python* by Miller, and Ranum.

11 Course website

The course website `http://courses.csail.mit.edu/6.006/spring12/` contains links to electronic copies of handouts, corrections made to the course materials, and special announcements. You should visit this site regularly to be aware of any changes in the course schedule, updates to your instructors' office hours, etc.

12 Extra help

Each TA will post the time and location of his or her office hours on the course website. Of course, you are also encouraged to ask questions of general interest in lecture or recitation. If you have questions about the course or problem sets, please mail `6.006-staff@mit.edu` as opposed to an individual TA or lecturer – there is a greater probability of getting a speedy response.

Extra help may be obtained from the following two resources. The MIT Department of Electrical Engineering and Computer Science provides one-on-one peer assistance in many basic undergraduate Course VI classes. During the first nine weeks of the term, you may request a tutor who will meet with you for a few hours a week to aid in your understanding of course material.

You and your tutor arrange the hours that you meet, for your mutual convenience. This is a free service. More information is available on the HKN web page:

`http://hkn.mit.edu/act-tutoring.html.`

Tutoring is also available from the Tutorial Services Room (TSR) sponsored by the Office of Minority Education. The tutors are undergraduate and graduate students, and all tutoring sessions take place in the TSR (Room 12-124) or the nearby classrooms. For further information, go to

`http://web.mit.edu/tsr/www.`

13 Guide in writing up homework

You should be as clear and precise as possible in your write-up of solutions. Understandability of your answer is as desirable as correctness, because communication of technical material is an important skill.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions.

This course has great material, so HAVE FUN!

14 Key Dates

For the details on the schedule, please refer to the calendar on the course website.

6.006 calendar Spring 2012

Tue Feb 7, 2012

11am - 12pm Lecture 1 - Intro
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

Thu Feb 9, 2012

All day pset 1 out
Thu Feb 9, 2012 - Fri Feb 10, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

11am - 12pm Lecture 2 - Divide & Conquer, Peak Finding
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

Tue Feb 14, 2012

11am - 12pm Lecture 3 - Binary Search Trees
Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Feb 15, 2012

10am - 11am Recitation with Shaunak
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

11am - 12pm Recitation with Shaunak
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

12pm - 1pm Recitation with Alan
Where: 34-302
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

1pm - 2pm Recitation with Jeff
Where: 34-302
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

2pm - 3pm Recitation with Rafael
Where: 36-156
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

3pm - 4pm Recitation with Dragos
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

3pm - 4pm Recitation with Rafael
Where: 36-156
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Feb 16, 2012

11am - 12pm Lecture 4 - Balanced Binary Search Trees
Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Feb 17, 2012

10am - 11am Recitation with Shaunak
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

11am - 12pm Recitation with Shaunak
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

12pm - 1pm Recitation with Alan
Where: 34-302
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

1pm - 2pm Recitation with Jeff
Where: 34-302
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

2pm - 3pm Recitation with Rafael
Where: 36-156
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

3pm - 4pm Recitation with Dragos
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

3pm - 4pm Recitation with Rafael
Where: 36-156
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Feb 21, 2012

All day No class, due to Monday Schedule!
Tue Feb 21, 2012 - Wed Feb 22, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 calendar Spring 2012

Wed Feb 22, 2012

All day Problem set 1 due
Wed Feb 22, 2012 - Thu Feb 23, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com
Description: At 11:59pm

10am - 4pm Recitations 10-4
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Feb 23, 2012

All day pset 2 out
Thu Feb 23, 2012 - Fri Feb 24, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

11am - 12pm Lecture 5 - Hashing I : Chaining, Hash Functions
Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Feb 24, 2012

10am - 4pm Recitations 10-4
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Feb 28, 2012

11am - 12pm
Lecture 6 - Hashing II : Table Doubling, Rolling Hash of Karp and Rabin
Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Feb 29, 2012

10am - 4pm Recitations 10-4
Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Mar 1, 2012

11am - 12pm Lecture 7 - Hashing III : Open Addressing
Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 calendar Spring 2012

Fri Mar 2, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Mar 6, 2012

11am - 12pm Lecture 8 - Sorting I : Insertion Sort, Merge Sort, Master Theorem

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Mar 7, 2012

All day pset 2 due

Wed Mar 7, 2012 - Thu Mar 8, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Mar 8, 2012

All day pset 3 out

Thu Mar 8, 2012 - Fri Mar 9, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

11am - 12pm Lecture 9 - Sorting II : Heaps, Heapsort

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Mar 9, 2012

All day ADD DATE!

Fri Mar 9, 2012 - Sat Mar 10, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Mar 13, 2012

11am - 12pm Lecture 10 - Sorting III: Lower Bounds, Counting Sort, Radix Sort

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 calendar Spring 2012

Wed Mar 14, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

7:30pm - 9:30pm Quiz 1

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

Thu Mar 15, 2012

11am - 12pm Lecture 11 - Searching I: Graph Search and Representations

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Mar 16, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Mar 20, 2012

11am - 12pm

Lecture 12 - Searching II: Breadth-First Search and Depth-First Search
Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Mar 21, 2012

All day pset 3 due

Wed Mar 21, 2012 - Thu Mar 22, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Mar 22, 2012

All day pset 4 out

Thu Mar 22, 2012 - Fri Mar 23, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

11am - 12pm Lecture 13 - Searching III: Topological Sort

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 calendar Spring 2012

Fri Mar 23, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Mon Mar 26, 2012

All day Spring Break! NO CLASS!

Mon Mar 26, 2012 - Tue Mar 27, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Mar 27, 2012

All day Spring Break! NO CLASS!

Tue Mar 27, 2012 - Wed Mar 28, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Mar 28, 2012

All day Spring Break! NO CLASS!

Wed Mar 28, 2012 - Thu Mar 29, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Mar 29, 2012

All day Spring Break! NO CLASS!

Thu Mar 29, 2012 - Fri Mar 30, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Mar 30, 2012

All day Spring Break! NO CLASS!

Fri Mar 30, 2012 - Sat Mar 31, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Apr 3, 2012

11am - 12pm Lecture 14 - Shortest Paths I: Intro

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Apr 4, 2012

All day pset 4 due

Wed Apr 4, 2012 - Thu Apr 5, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Apr 5, 2012

All day pset 5 out

Thu Apr 5, 2012 - Fri Apr 6, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

11am - 12pm Lecture 15 - Shortest Paths II: Bellman-Ford

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Apr 6, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Apr 10, 2012

11am - 12pm Lecture 16 - Shortest Paths III: Bellman-Ford on DAGs and Dijkstra

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Apr 11, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Apr 12, 2012

11am - 12pm Lecture 17 - Shortest Paths IV: Speeding Up Dijkstra

Where: 32-123
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Apr 13, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Apr 17, 2012

All day NO CLASS (Patriot's day)

Tue Apr 17, 2012 - Wed Apr 18, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 calendar Spring 2012

Wed Apr 18, 2012

All day pset 5 due

Wed Apr 18, 2012 - Thu Apr 19, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu Apr 19, 2012

All day pset 6 out

Thu Apr 19, 2012 - Fri Apr 20, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

11am - 12pm Lecture 18 - Dynamic Programming I

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri Apr 20, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue Apr 24, 2012

11am - 12pm Lecture 19 - Dynamic Programming II

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed Apr 25, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

7:30pm - 9:30pm quiz 2

Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

Thu Apr 26, 2012

All day DROP DATE!!!

Thu Apr 26, 2012 - Fri Apr 27, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

11am - 12pm Lecture 20 - Dynamic Programming III

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 calendar Spring 2012

Fri Apr 27, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue May 1, 2012

11am - 12pm Lecture 21 - Dynamic Programming IV

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed May 2, 2012

All day pset 6 due

Wed May 2, 2012 - Thu May 3, 2012
Calendar: 6.006 calendar Spring 2012
Created by: konstantinos.daskalakis@gmail.com

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu May 3, 2012

11am - 12pm Lecture 22 - NP Completeness

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Fri May 4, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue May 8, 2012

11am - 12pm Lecture 23 - Numerics I

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed May 9, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu May 10, 2012

11am - 12pm Lecture 24 - Numerics II

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 calendar Spring 2012

Fri May 11, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Tue May 15, 2012

11am - 12pm Lecture 25 - Crypto

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Wed May 16, 2012

10am - 4pm Recitations 10-4

Where: 36-153
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

Thu May 17, 2012

All day Last day of classes!

Thu May 17, 2012 - Fri May 18, 2012
Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

11am - 12pm Lecture 26 - Surfing

Calendar: 6.006 calendar Spring 2012
Created by: rafaeloliveira.mit@gmail.com

6.006 Lecture 1

2/7

(No slides handed out)
Costis Daskalakis - accent ?

Silvio Micali

Today: (missed it)

Def: a well specified method for solving a problem using a finite sequence of instructions

- English
- Psudo code
- real code

) as long as unambigious

From Al-khwārizmi
to solve quadratic equations

Efficient Algorithms
- time
- space
- energy

Bigger problems consume more resources
`so want algorithms that scale

②

- Questions in job interviews

How:

- unamb def of desired result
- abstract irrelevant details
- pull tools from algorithmic toolbox
- implement
- iterate

Content

- 8 modules
- today: Linked Data Structures
- next time: Divide + Conquer Peak Finding
- Soon: Hashing
- ~~Sorta~~ Sorting

Graph Search: Rubik's Cube

Shortest Path: Google Maps

Dynamic Programming

Wildcard

end of year ↓

(3)

Admin

Sign up on hw submission site sec.csail.mit.edu

Use Piazza

Pre req 6.01, 6.042

Class 2x as big as historically
- 260 people

Class uses Python
- not the focus
- method of communicating

P-Sets Theory + Programming
- effecincy - running time
- not astetics of code

Grading

P-set 30%

Quiz 1 20%

2 20%

Exam 30%

Read collab policy - acknolge
- ~~do~~ understand what you submit

(4)

Document Distance (application)

Given 2 documents - how similar are they?

- So can detect plagiarism
- Goal: algorithm to compute similarity

But what does it mean for doc to be similar

- Word = sequence of alpha characters
  - ignore punctuation, formatting
  - ignore sequence of words

So have multi set of words

$D(w)$ = # occurances of w in D

Similarity is # of words that over lap

Called Vector-Space Model

- Salton, Wong, Yang

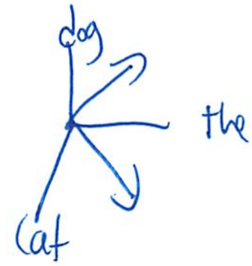
Multi-dimension Euclidean space

↳ 1 word in Eng dictionary = 1 dimension

⑤

"the cat"

"the dog"



Take dot product

$$D_1 \circ D_2 = \sum_w D_1(w) \cdot D_2(w)$$

But not scale invariant

the the cat cat) appear closer than doc 1 + doc 2
the the dog dog

Can divide by product of magnitudes to normalize

$$\frac{D_1 \circ D_2}{\|D_1\| \cdot \|D_2\|}$$

Measure by angle

$$\theta(D_1, D_2) = \operatorname{acos}\left(\frac{D_1 \circ D_2}{\|D_1\| \cdot \|D_2\|}\right)$$

$\theta = 0$ if identical

$\theta = \frac{\pi}{2}$ if don't share a word

(6)

Algorithm

1. Read File
2. Make word list
3. Count frequencies
4. Compute dot product
   - if word appears in other doc
   - if it does multiply the freq.

Worst case $O(n^2)$

Since $D_1 \times D_2$ (#words D_1 × #words D_2)

Optimizations

Could sort doc into alpha order

2 finger algorithm
- Start at begining at first doc
- if same - multiply
- if not - word is smaller alphabetically
  - increase
  - repeat w/ smaller word

$O(2n)$ #words D_1 + #words D_2 after sorting

(7) Python code on website

read_file(filename)

get_words_from_line_list(L)

count_frequency(word_list)

insertion_sort()

inner_product(D1, D2) ← 2 finger algorithm

Have some books to test

How to know where weak?

Are profiling tools (docdist2.py)

Something looks slow

Investigate

Its because of stupid way Python appends lists w/ +

Creates a bigger array

Copys both lists into new bigger array

Time $1 + 2 + \ldots + n$ (proportional to length of lists)

$= \frac{n(n+1)}{2} = O(n^2)$

(8) In 3

Using list.extend(...)

is time $O(n)$ ← linear

So 23 sec → .1 sec for that subroutine

In 4 | total time

Use dictionaries to count frequencies — 42 sec

In 5

Diff word processing — 17 sec

In 6

Use merge sort not insertion sort — 6 sec

In 7

No sorting — just dictionaries — .5 sec

So together 94 sec → .5 sec

(9)

Will see dicitionaries / hasing som

Next time: Peak Finding

nxn table

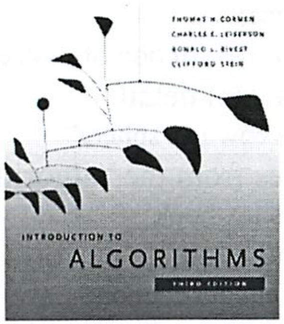
Find pt bigger than neighbors

Could do in $O(n^2)$

Can we beat it?

2/7

6.006-Introduction to Algorithms



Lecture 1
Prof. Costis Daskalakis

Today's Menu

- Motivation
- Course Overview
- Administrivia
- Linked Lists and Document Distance
- Intro to "Peak Finding"

Lecture 1

"Al-go-rithms": what?

- Nothing to do with Log-arithms ☺
- **Def:** A well-specified method for solving a problem using a finite sequence of instructions.
- Description might be English, Pseudocode, or real code
- Key: no ambiguity

Al-Khwārizmī (780-850)

2/7

Efficient Algorithms: Why?

- Solving problems consumes resources that are often limited/valuable:
  - Time: Plan a flight path
  - Space: Process stream of astronomical data
  - Energy: Save money
- Bigger problems consume more resources
- Need algorithms that "scale" to large inputs, e.g. searching the web…
- Market value: 6.006 is useful in all kinds of job interviews ;-)

Efficient Algorithms: How?

- Define problem:
  - Unambiguous description of desired result
- Abstract irrelevant detail
  - "Assume the cow is a sphere"
- Pull techniques from the "algorithmic toolbox"
  - [CLRS] class textbook
- Implement and evaluate performance
  - Revise problem/abstraction
- Generalize
  - Algorithm to apply to broad class of problems

Class Content

- 8 modules with motivating problem/pset
- **Linked Data Structures**: Document Distance/ Flight Planning
- **Divide & Conquer**: Peak Finding
- **Hashing**: Efficient File Update/Synchronization
- **Sorting**
- **Graph Search**: Rubik's Cube
- **Shortest Paths**: Google Maps
- **Dynamic Programming**: print justification
- **Wildcard: numerical/NP-hardness/crypto**

Administrivia

- Course information: class website
- Profs: Costis Daskalakis, Silvio Micali
- TAs: Deckelbaum, Ionescu, Kishore, Oliveira, Wu
- Sign-up to the homework submission website: https://alg.csail.mit.edu (same as https://sec.csail.mit.edu/)
- Piazza: online discussion
- Prereqs: 6.01, 6.042 (if you don't have them, talk to us)
- Python
- Grading: Problem sets (30%)
Quiz1 (March 14 (?): 7.30-9.30pm; 20%)
Quiz2(April 18 (?): 7.30-9.30pm; 20%)
Exam (30%)
- Read collaboration policy!

Document Distance

- Given 2 documents, how similar are they?
  - if one “document ” is a query, this is web search
  - if the two documents are homework submissions, can detect plagiarism
  - …
- Goal: algorithm to compute similarity

Problem Definition

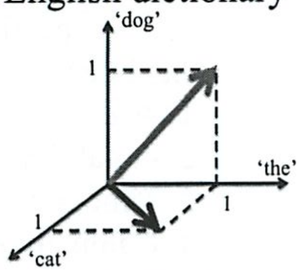
- Need unambiguous definition of similarity
- Word: sequence of alpha characters
  - Ignore punctuation, formatting
- Document: sequence of words
- Word frequencies:

  D(w) is number of occurences of w in D
- Similarity based on amount of word overlap

Vector Space Model

- [Salton, Wong, Yang 1975]
- Treat each doc as a vector of its words
  - one coordinate per word of the English dictionary

    e.g. doc1 = “the cat”
    doc2 = “the dog”



  - similarity by dot-product

$$D_1 \circ D_2 \equiv \sum_w D_1(w) \cdot D_2(w)$$

d1∘ d2 = 1

  - trouble: not scale invariant

    documents “the the cat cat” and “the the dog dog” will appear closer than doc1 and doc2

Vector Space Model

- Solution: Normalization
  - divide by the length of the vectors

$$\frac{D_1 \circ D_2}{||D_1|| \cdot ||D_2||}$$

  - measure distance by angle:

$$\theta(D_1, D_2) = \text{acos}\left(\frac{D_1 \circ D_2}{||D_1|| \cdot ||D_2||}\right)$$

e.g. θ=0 documents “identical”
(if of the same size, permutations of each other)

$\theta=\pi/2$ not even share a word

Algorithm

- Read file
- Make word list (divide file into words)
- Count frequencies of words
- Suppose each document has been processed into a list of distinct words with their frequencies
- Compute dot product
  - for every word in the first document, check if it appears in the other document; if yes, multiply their frequencies and add to the dot product
    - worst case time: order of #words(D_1) x #words(D_2)
  - micro-optimization:
    - sort documents into word order (alphabetically)
    - after having sorted, can compute inner product in time #words(D_1) + #words(D_2)

Python Implementation

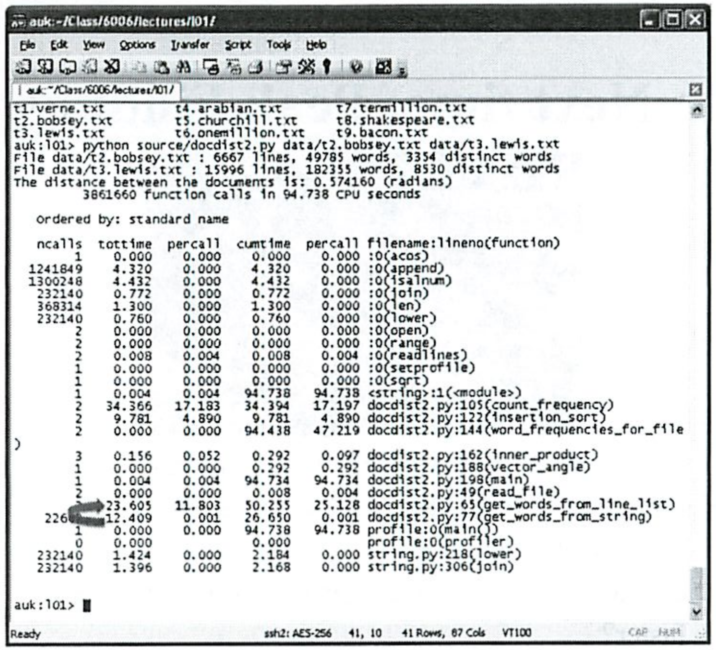
- Docdist1.py (on course website)
- Read file: read_file(filename)
  - Output: list of lines (strings)
- Make word list: get_words_from_line_list(L)
  - Output: list of words (array)
- Count frequencies: count_frequency(word list)
  - Output: list of word-frequency pairs
- Sort into word order: insertion_sort()
  - Output: sorted list of pairs
- Dot product: inner_product(D1, D2)
  - Output: number

Inputs:

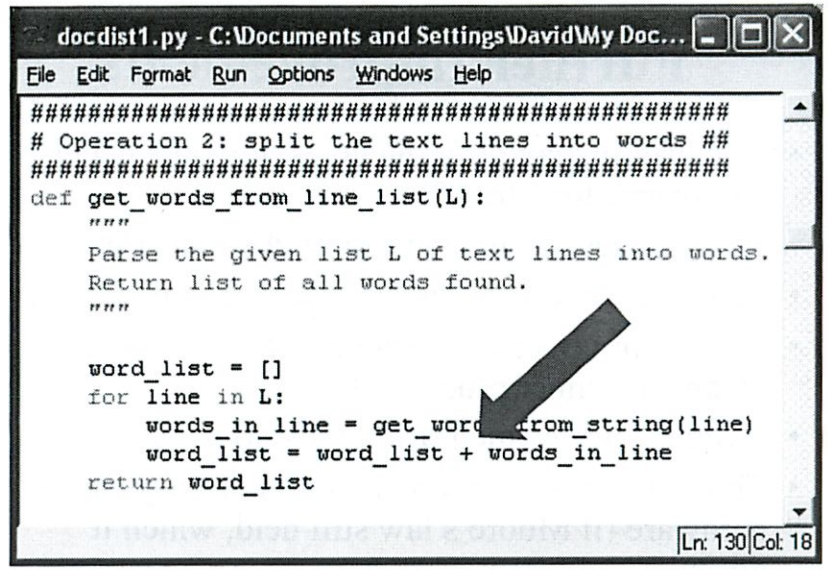
- Jules Verne: 25K
- Bobbsey Twins: 268K
- Francis Bacon: 324K
- Lewis and Clark: 1M
- Shakespeare: 5.5M
- Churchill: 10M

Profiling (docdist2.py)

- Tells how much time spent in each routine
  - import profile
  - profile.run("main()")
- One line per routine reports
  1. #calls
  2. #total time excluding subroutine calls
  3. Time per call (#2/#1)
  4. Cumulative time, including subroutines
  5. Cumulative per call (#4/#1)



```
auk:~/Class/6006/lectures/l01/
File Edit View Options Transfer Script Tools Help
t1.verne.txt          t4.arabian.txt         t7.termillion.txt
t2.bobsey.txt         t5.churchill.txt       t8.shakespeare.txt
t3.lewis.txt          t6.onemillion.txt      t9.bacon.txt
auk:101> python source/docdist2.py data/t2.bobsey.txt data/t3.lewis.txt
File data/t2.bobsey.txt : 6667 lines, 49785 words, 3354 distinct words
File data/t3.lewis.txt : 15996 lines, 182355 words, 8530 distinct words
The distance between the documents is: 0.574160 (radians)
         3861660 function calls in 94.738 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(acos)
  1241849    4.320    0.000    4.320    0.000 :0(append)
  1300248    4.432    0.000    4.432    0.000 :0(isalnum)
   232140    0.772    0.000    0.772    0.000 :0(join)
   368314    1.300    0.000    1.300    0.000 :0(len)
   232140    0.760    0.000    0.760    0.000 :0(lower)
        2    0.000    0.000    0.000    0.000 :0(open)
        2    0.000    0.000    0.000    0.000 :0(range)
        2    0.008    0.004    0.008    0.004 :0(readlines)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(sqrt)
        1    0.004    0.004   94.738   94.738 <string>:1(<module>)
        2   34.366   17.183   34.394   17.197 docdist2.py:105(count_frequency)
        2    9.781    4.890    9.781    4.890 docdist2.py:122(insertion_sort)
        2    0.000    0.000   94.438   47.219 docdist2.py:144(word_frequencies_for_file
)
        3    0.156    0.052    0.292    0.097 docdist2.py:162(inner_product)
        1    0.000    0.000    0.292    0.292 docdist2.py:188(vector_angle)
        1    0.004    0.004   94.734   94.734 docdist2.py:198(main)
        2    0.000    0.000    0.008    0.004 docdist2.py:49(read_file)
           23.605   11.803   50.255   25.128 docdist2.py:65(get_words_from_line_list)
      226  12.409    0.001   26.650    0.001 docdist2.py:77(get_words_from_string)
        1    0.000    0.000   94.738   94.738 profile:0(main())
        0    0.000             0.000          profile:0(profiler)
   232140    1.424    0.000    2.184    0.000 string.py:218(lower)
   232140    1.396    0.000    2.168    0.000 string.py:306(join)

auk:101>
Ready    ssh2: AES-256   41, 10   41 Rows, 87 Cols   VT100   CAP NUM
```



```
docdist1.py - C:\Documents and Settings\David\My Doc...
File Edit Format Run Options Windows Help
#################################################
# Operation 2: split the text lines into words ##
#################################################
def get_words_from_line_list(L):
    """
    Parse the given list L of text lines into words.
    Return list of all words found.
    """

    word_list = []
    for line in L:
        words_in_line = get_words_from_string(line)
        word_list = word_list + words_in_line
    return word_list
Ln: 130 Col: 18
```

What's with +?

- L=L1+L2 is concatenation of arrays
- Take L1 and L2
- Copy to a bigger array
- Time proportional to sum of lengths
- Suppose n single-word lines
- Time 1+2+…+n = $n(n+1)/2 = \Theta(n^2)$

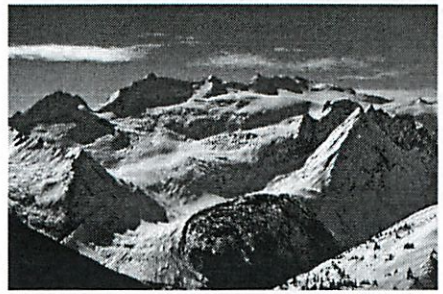
Solution

- word_list.extend(words_in_line) : appends list named "words_in_line" to list named "word_list"
- Takes time proportional to length of list "words_in_line"
- Total time in example of n single-word lines: $\Theta(n)$
- resulting improvement:
  - get_words_from_line_list 23s→0.12s

Further Improvements

- Docdist4.py: count frequencies of words using dictionary: total to 42s
- 5.py: Process words instead of chars: to 17s
- 6.py: merge sort instead of insertion sort: 6s
- 7.py: remove sorting altogether and use dictionary (again) for inner product: 0.5s
- Overall improvement from 94 s to 0.5 s.
- This is the equivalent of 12 years of progress in hardware (if Moore's law still held, which it doesn't)

Next time: Peak Finding



- n x n table of numbers (heights of points)
- Find a point that is bigger than its neighbors
- i.e. a local maximum
- can do this by querying $O(n^2)$ locations of table
- faster?

6.006 Recitation 1

2/8

Shawn ah Kishore

Skishore@mit.edu

But email staff list

Python 2.6 or 2.7
- not 3.0

(lots of people here) – class full

(hes not a morning person

---

order
- Binary Search
- Sorts
- BSTS and Binary Trees
  - ↑ class (BSTS), ↑ real world (Binary Trees)
- Hashing

Will ~~code~~ ~~"reify"~~ functions
- use ↑ Reified

L goes w/ λ
function that is turned into an object

(2)

Can be past around and used as an argument to others

Coding Binary Search

have list - unsorted
want to see if target is in list
Could have linear scan
if input is sorted
can do it faster
Open up to middle
branch on 1st or 2nd half
recurse on that half
(take half again)

input: list of int - sorted
target: return true if found

low, high are the positions we are currently looking at

```
def bsearch(input, target):
→ low = 0
  high = len(input) - 1
  ⋮
```

Redo w/ recursive

```
def bsearch(input, target, low = 0, high = None):
  if high is None:
     high = len(input) - 1
  mid = (low + high) / 2
  ↓ contine
```

← if high < low: return False

↙ So that we return False if item not in the list

(It was good he described problem - I was thinking something different....)

(3)

```
if input[mid] == target:
    return True
elif input[mid] < target:
    return bsearch(input, target, mid+1, high)
else:
    return bsearch(input, target, low, mid-1)
```

If list is not an integer - it will crash
in Python you need to hope its right

Python is strongly typed
it is not statically typed
its called "duck" typing
- if value supports that operation than it works

(4)

Let's test on 2 test cases

x = range(100)
bsearch(x, 82.5)
  └ False

bsearch(x, 82)
  └ True   ← since we are just returning True, not the index on the list

If items are on list twice, does it still work?
  └ TA says yes

Running Time

We can't talk exactly
  Since processes varry greatly
So talk about times in terms of N
  ℕ → N in CS
  ↑
  natural #s

(5)

$f(n) = O(g(n))$
↑ order of

if $f(n) \leq c\, g(n)$
↑ for some $c > 0$
for all $n \geq N$ for some N

So

$10n + 5 = O(n)$ is true

$\frac{n^3}{3} + 5n = O(n^4)$ is true
↑ just means less than

$= O(n^3)$ is also true
↑ gives us more info
Since is tighter bound

linear search → $O(n)$
since goes through each
worst case

(6)

binary search → $O(\log_2 n)$

We can write a recurrence for how long it takes

$T(n)$ is the <u>worst case</u> of bsearch on input of size n

$T(n) = T(n/2) + O(1)$

↑since shrinking list in half

The $O(1)$ since all the stuff does not depend on size of list

if we were copying list it would be $O(n)$

$= O(\log_2 n)$

↑we keep cutting in half

the base 2 does not matter

logs of diff bases have same asy complexity

$\log_2 n = \frac{\ln n}{\ln 2}$

$= O(\log n)$

So at very large lists log n is better than n

Even though we did 5 calls vs 2 calls each time it runs, asymptotically its the same

---

We could could expand recurrence more

T( ) is the time of the function

Most recurrences are not that easy

More recurrences

$$T(n) = T(n/2) + O(1)$$

1st tool: almost all involve cutting by constant factor
build tree

amt of work from each level ↓

Count height + width

→ recursive term = # blocks per level

# of levels

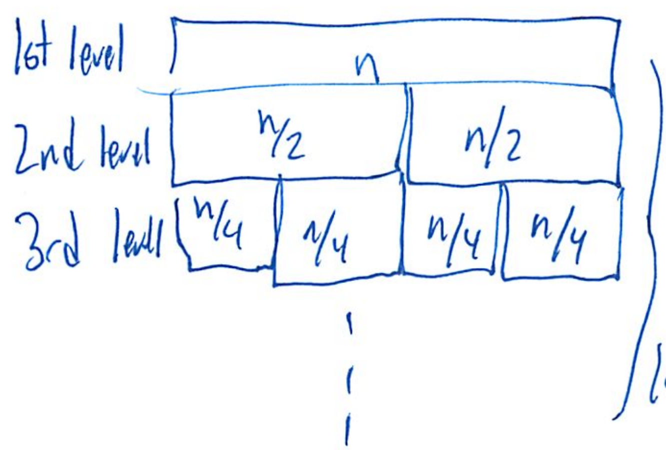
$$T(n) = 1T(n/2) + O(1)$$ ← size of blocks

$O(\log n)$ by reducing # of steps

Harder Recurrence

$$T(n) = 2\,T(n/2) + O(n)$$

↑ blocks come here



$O(n)$ = size each level

$\log(n)$ = # of levels
comes from $T(n/2)$

$= O(n \log n)$

Last year's but close

Asymptotic analysis

Asymptotic analysis or "big O" notation is a way of describing the growth of the runtime of an algorithm without without having to worry about different computers, compilers, or implementations.

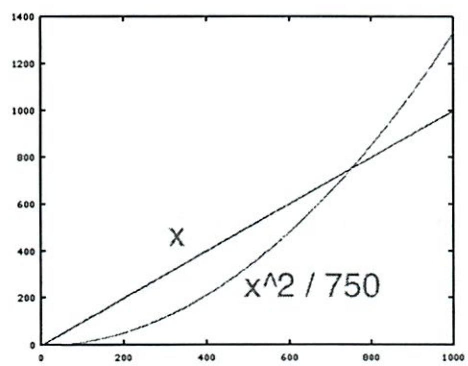
For functions $f(n)$, $g(n)$, $O(g(n))$ is a class of functions such that $f(n) \in O(g(n))$ if there exist M, x_0 such that

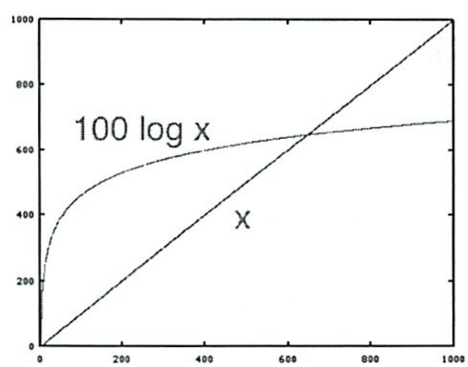
$$|f(n)| \leq M \cdot |g(n)| \text{ for all } x > x_0.$$

Similarly, $f(n) \in \Omega(g(n))$ if there exist M, x_0 such that

$$|f(n)| \geq M \cdot |g(n)| \text{ for all } x > x_0.$$

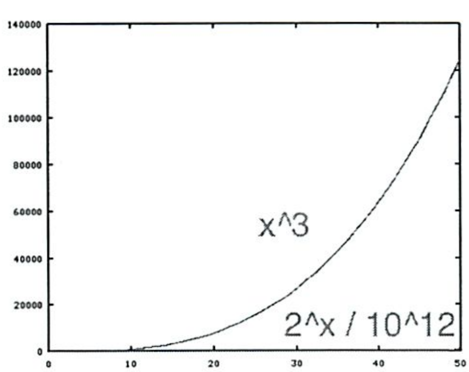
If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then we write $f(n) \in \Theta(g(n))$.

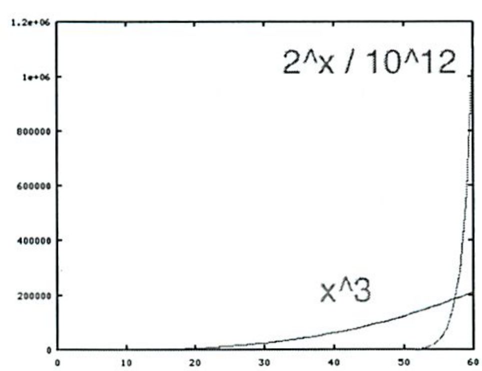




$x = \_\_\_ \ (x^2)$ $\qquad M = \_\_\_, x_0 = \_\_\_$ (1)

$x = \_\_\_ \ (\log x)$ $\qquad M = \_\_\_, x_0 = \_\_\_$ (2)





$x^3 = \_\_\_ \ (2^x)$ $\qquad M = \_\_\_, x_0 = \_\_\_$ (3)

Python

This class uses Python 2.6. Do not use Python 3. If you're not familiar with Python, there are numerous resources available on the Internet:

- Python tutorial: `http://docs.python.org/tutorial/`
- Python libraries: `http://docs.python.org/library/`
- 6.006 resources page: `http://courses.csail.mit.edu/6.006/spring11/resources.shtml`

Docdist code samples

Insertion sort

```
def insertion_sort(A):
    for j in range(len(A)):
        key = A[j]
        # insert A[j] into sorted sequence A[0..j-1]
        i = j-1
        while i>-1 and A[i]>key:
            A[i+1] = A[i]
            i = i-1
        A[i+1] = key
    return A
```

Count frequency

```
def count_frequency(word_list):
    """
    Return a list giving pairs of form: (word,frequency)
    """
    L = []
    for new_word in word_list:
        for entry in L:
            if new_word == entry[0]:
                entry[1] = entry[1] + 1
                break
        else:
            L.append([new_word,1])
    return L
```

Improved count frequency

```
def count_frequency(word_list):
    """
    Return a dictionary mapping words to frequency.
    """
    D = {}
    for new_word in word_list:
        if new_word in D:
            D[new_word] = D[new_word]+1
        else:
            D[new_word] = 1
    return D
```

Get words from line list

```
def get_words_from_line_list(L):
    """
    Parse the given list L of text lines into words.
    Return list of all words found.
    """
    word_list = []
    for line in L:
        words_in_line = get_words_from_string(line)
        word_list = word_list + words_in_line
    return word_list
```

Improved get words from line list

```
def get_words_from_line_list(L):
    """
    Parse the given list L of text lines into words.
    Return list of all words found.
    """
    word_list = []
    for line in L:
        words_in_line = get_words_from_string(line)
        word_list.extend(words_in_line)
    return word_list
```

Inner product

```
def inner_product(L1,L2):
    """
    Inner product between two vectors, where vectors
    are represented as alphabetically sorted (word,freq) pairs.
    Example: inner_product(
        [["and",3],["of",2],["the",5]],
        [["and",4],["in",1],["of",1],["this",2]]) = 14.0
    """
    sum = 0.0
    i = 0
    j = 0
    while i<len(L1) and j<len(L2):
        # L1[i:] and L2[j:] yet to be processed
        if L1[i][0] == L2[j][0]:
            # both vectors have this word
            sum += L1[i][1] * L2[j][1]
            i += 1
            j += 1
        elif L1[i][0] < L2[j][0]:
            # word L1[i][0] is in L1 but not L2
            i += 1
        else:
            # word L2[j][0] is in L2 but not L1
            j += 1
    return sum
```

Improved inner product

```
def inner_product(D1,D2):
    """
    Inner product between two vectors, where vectors
    are represented as dictionaries of (word,freq) pairs.
    Example: inner_product(
        {"and":3,"of":2,"the":5},
        {"and":4,"in":1,"of":1,"this":2}) = 14.0
    """
    sum = 0.0
    for key in D1:
        if key in D2:
            sum += D1[key] * D2[key]
    return sum
```

Prof. Silvio

(accent as well) (funny) (talks fast)

Peak Finding 1D and 2D

Technique: Divide + Conquer

Peak - not smaller than neighbors
- must not be the maximum

13 5 8 3

↑ peak

- also includes ends

15 13 ....

↑ Peak

- So want any peak, one of them as fast as possible

②

Niere

Iterate through each
Look left and right
but $O(n)$ in worst case
- ie a sorted list

Can it happen that no peak?
Postpone q/, Assume there is

Take 2

Go to middle
Compare to left
if >, look left
Otherwise <, look right
Otherwise =, found peak

(full code on slides)

$O(\log n)$

Since $T(n) = T(n/2) + O(1)$

do more recursion $= \underbrace{O(1) + O(1) + \cdots + O(1)}_{\log_2 n \text{ times}} = O(\log n)$

(3)

Divide + Conquer

↑ disjont problems

↑ each part separtly recursive

Occasionally we need to combine results

2D

N rows, N columns

must be not smaller (≥) its (at most) 4 neighbors

Now want <u>any</u> 2D peak

^notice problem subilites

<u>Idea</u>: Recycling

Never solve from scratch if you can't recycle

Your time is valuable

For each row, until you find a global peak

1. find a row peak
2. Look North, South
3. <u>If</u> ≥, then done

(4)

Is this a good algorithm?

Can you have a 2D array w/ no peak

You can have multiple peaks in 1 row

Must look at <u>all</u> peaks in row, look North/South for each

This will work now

But running time?

<u>Algorithm I</u>

For each column find <u>global</u> maximum B[j]

Now use old row code to find <u>a</u> peak

Notice it matches that cols' global max

So return it

(I don't like since not easily extensible - but I need to think this way)

(5)

So that works

Complexity

2 variables n and m since 2D
(n ↑ row, m ↑ col)

O(n · m)
↑ m cols
must look at all n

Then also fancy thing w/ recycling
(why not included?)

I think
n·m + m log n
↑ but only biggest one
confirm ✓

Algorithm 1b

Recall peak finder used O(log m)
Modify so only compute B[i] when needed

O(n log m)
↑ Need O(log m) for B[i]
Each computed in O(n) time

(b)

Algorithm 2

1. Pick middle column $j = m/2$
2. Find global max $a = A[i, m/2]$ in that col
   quit if $m = 1$
3. Compare a to $b = A[i, m/2 - 1]$ and $c = A[i, m/2 + 1]$
4. If $b > a$
   then recurse on left
5. If $c > a$
   recurse on right
6. Otherwise a is a peak

Can prove by contradiction

Assume no peak on left
B must have a value that is greater (B1)
B1 must have higher neighbor (B2)
(stay in column)
Until out of items must find peak

7

Complexity

We have

tool for toolbox
give things name

$$T(n,m) = T\left(n, \frac{m}{2}\right) + O(n)$$

↑ recursion ↑ scanning middle column

$$= \underbrace{O(n) + O(n) + \cdots + O(n)}_{\log_2 m}.$$

$$= O(n \log m)$$

~~Is that any different?~~

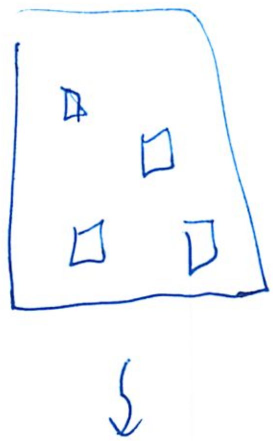
Is there anything faster?

Reading $(n+m)$ elements reduce an array of $n \times m$ candidates to any $\frac{n}{2} \times \frac{m}{2}$ candidates.

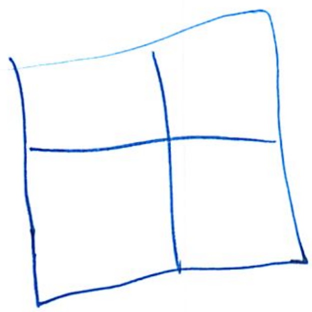
Can we solve it in linear time?

(8)

Pictorially



read only $O(n+m)$ elements



Clever recursion

$$T(n,m) = T\left(\frac{n}{2}, \frac{m}{2}\right) + O(n+m)$$

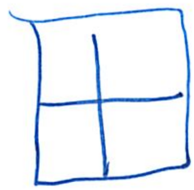
Hense

$$T(n,m) = O(n+m) + O\left(\frac{n+m}{2}\right) + O\left(\frac{n+m}{4}\right) + \ldots \text{ (could not read)}$$

← this is just $O(n+m)$ discard info

(9)

Where to look?

Look at axis 

Find global max on cross

L is n + m

If middle element → done 

But if like this? 

<u>Claim</u> The ~~dir~~ sub square always contains a peak

? Is it right? Easy to make claims?

First P-Set is out

<u>Post</u> Shi

Go into sub square

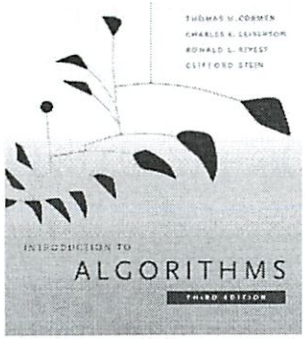
Do the ~~x~~ cross again

But what if both? - still O(n+m)

- a large multiple of
- but still O(n+m)

(I don't like how this is lossy)

6.006- *Introduction to Algorithms*



Lecture 2

Prof. Silvio Micali

Menu

Problem: peak finding

1 dimension

2 dimensions

Technique: *Divide and conquer*

Peak Finding: 1D

Consider an array A[1…n] :

15	13	5	8	3	2	1

Element A[i] is a *peak* if **not smaller** than its neighbor(s).

if $i \neq 1, n$: $A[i] \geq A[i-1]$ and $A[i] \geq A[i+1]$
If $i=1$: $A[1] \geq A[2]$
If $i=n$: $A[n] \geq A[n-1]$

Problem: find ***any*** peak.

Peak-Finding Ideas ?

Algorithm I:

Scan the array from left to right

Compare each A[i] with its neighbors

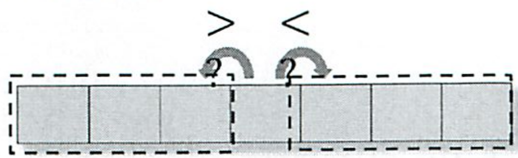
Exit when found a peak

Complexity:

Might need to scan all elements, so $T(n)=\Theta(n)$

2/9

Next Idea



Algorithm II:
Compare middle element with neighbors

If A[n/2-1]>A[n/2]
  then search for a peak among A[1]… A[n/2-1]
Else, if A[n/2]<A[n/2+1]
  then search for a peak among A[n/2]… A[n]
Else A[n/2] is a peak!

Running time ?

Algorithm II: Complexity

Algorithm II: Complexity

- We have

Time needed to find peak in array of length n

Recursion

Time for comparing A [n/2] with neighbors

$$T(n) = T(n/2) + \Theta(1)$$

- Unraveling the recursion,

$$T(n)= \underbrace{\Theta(1) + \Theta(1) + \ldots + \Theta(1)}_{\log_2 n} = \mathbf{\Theta(\log n)}$$

- log n is much much better than n !

Divide and Conquer

- Very powerful design tool:
  – *Divide* input into multiple **disjoint** parts
  – *Conquer* each of the parts **separately** (using recursive call)
- *Occasionally*, we need to **combine** results from different calls (not used here)

Peak Finding: 2D

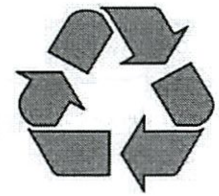
Consider a 2D array A[1…n, 1…m] :

10	8	5
3	2	1
7	13	4
6	8	3

A[i] is a *2D peak* if not smaller than its (at most 4) neighbors.

Problem: find any 2D peak.

2D-Peak-Finding Ideas?



Algorithm 0:

For each row, until you find a peak:

1. find a row-peak
2. compare it with North- and South-neighbors
3. If ≥, then done

?

Algorithm I: recycle better 1D algorithm

For each column j, find its *global* maximum B[j]

Apply 1D peak finder to find a peak (say B[j]) of B[1...m]

Correctness: …

Complexity: Θ(n·m)

Recycling is an art…

Return it!

12	8	5
11	3	6
10	9	2
8	4	1

"Map it back"

12	9	6

Algorithm I': use the 1D algorithm

- **Recall:** 1D peak finder uses only O(log m) entries of B
- Modify Algorithm I so that it only computes B[j] ***when needed*** !
- Total time ?

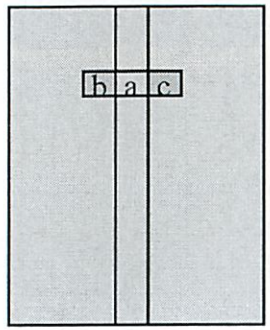
  …only O(n log m) !
  - Need O(log m) entries B[j]
  - Each computed in O(n) time

12	8	5
11	3	6
10	9	2
8	4	1

12	9	6

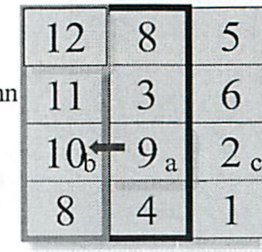
Algorithm II

- Pick middle column (j=m/2)
- Find *global* maximum a=A[i,m/2] in that column (and quit if m=1)
- Compare a to b=A[i,m/2-1] and c=A[i,m/2+1]
- If b>a
  then recurse on left columns
- Else, if c>a
  then recurse on right columns
- Else a is a 2D peak!



Algorithm II: Example

- Pick middle column (j=m/2)
- Find *global* maximum a=A[i,m/2] in that column (and quit if m=1)
- Compare a to b=A[i,m/2-1] and c=A[i,m/2+1]
- If b>a
  then recurse on left columns
- Else, if c>a
  then recurse on right columns
- Else a is a 2D peak!



Algorithm II: Correctness

Claim: If b>a, then there is a peak among the left columns

Proof (by contradiction):

Assume no peak on the left

Then b must have a neighbor b1 with higher value

And b1 must have a neighbor b2 with higher value

…

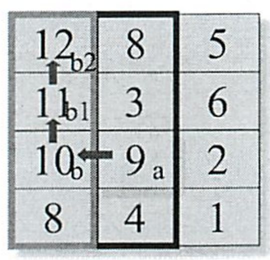
We have to stay on the left side – why?

(because we cannot enter the middle column)

But at some point, we would run out the elements of the left columns

Hence, we have to find a peak at some point.

Question: Does the above claim suffice for the proof of correctness of the algorithm?



Algorithm II: Complexity

- We have

$$T(n,m) = T(n,m/2) + \Theta(n)$$

Recursion ($T(n,m/2)$); Scanning middle column ($\Theta(n)$)

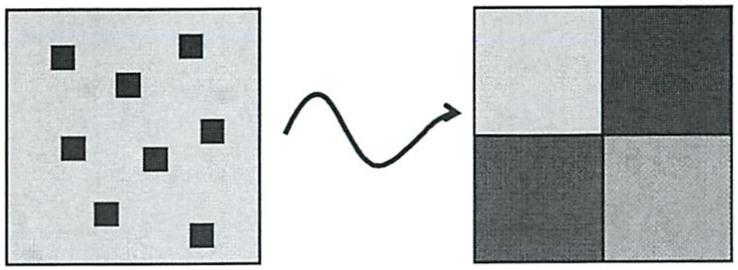
- Hence:
  - $T(n,n) = \underbrace{\Theta(n) + \Theta(n) + \ldots + \Theta(n)}_{\log_2 m} = \mathbf{\Theta(n\log m)}$

Faster than O(n log n) ?

- Idea:

  Reading only $O(n + m)$ elements, reduce an array of $n \times m$ candidates to an array of $n/2 \times m/2$ candidates

- Pictorially:



read only $O(n + m)$ elements

Faster than O(n log n) ?

- Hypothetical algorithm has recursion:

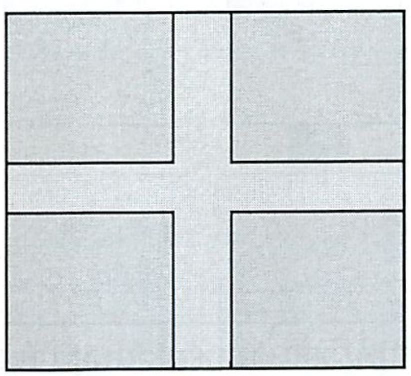
$$T(n,m) = T\left(\frac{n}{2}, \frac{m}{2}\right) + \Theta(n+m)$$

- Hence: $T(n,m) = \Theta(n+m) + \Theta\left(\frac{n+m}{2}\right) + \Theta\left(\frac{n+m}{4}\right) + \ldots + \Theta(1)$

$$= \Theta(n+m) \quad !$$

Towards a linear-time algorithm

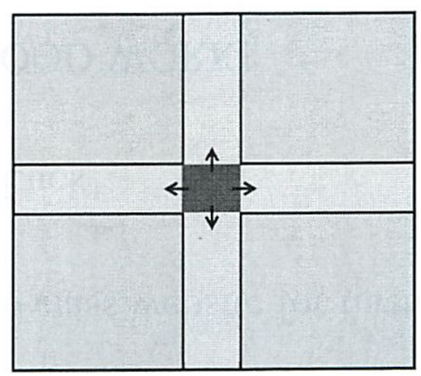
What elements are useful to check?



- suppose we find global max on the cross

Towards a linear-time algorithm

What elements are useful to check?

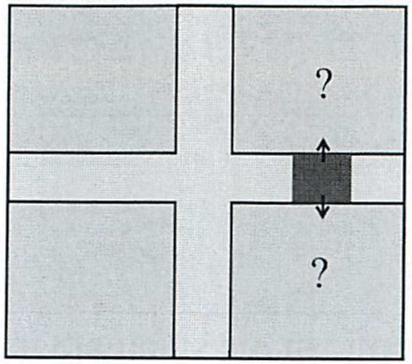


- suppose we find global max on the cross

- if middle element done!

Towards a linear-time algorithm

What elements are useful to check?



- find global max on the cross
- if middle element done!
- o.w. two candidate sub-squares
- determine which one to pick by looking at its neighbors not on the cross (as in Algorithm II)

Claim: The sub-square chosen by the above procedure (if any), always contains a peak of the large square.

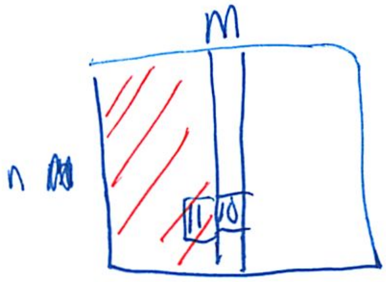
OK, what else is needed for an O(n+m) algorithm?
Hmmm…

First Problem Set Out Today !

- Refer to class website for further information!
- Good Luck!
- I.e., GOOD WORK!

6.006 Recitation 2 — 2/10

Last lecture

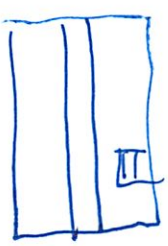


Then call 

$O(n \log m)$

Since $T(n, m) = T(n, \frac{m}{2}) + \Theta(n)$ ← Θ is lower bound as well exact bound (so far have not been careful O vs Θ)

Then call in middle of new row



Repeat ↳

$= T(n, \frac{m}{4}) + \Theta(n) + \Theta(n)$

$= T(n, \frac{m}{8}) + \Theta(n) + \Theta(n) + \Theta(n)$

repeat till $m = 1$ ~~Can add them all together~~ ↑ will be $\log m$ of these

②

Proof from lecture was wrong

↳ wrong year's notes

You can always find a peak by starting somewhere
and then always going up till peak

↳ Hillclimbing

So worst O(n)

When we found middle column max we know
we never want to cross that column
since we found the max

We won't do this, too slow
But proof
↳ by induction

(3)

But not peak if the middle Column is larger if searching Column just to the left

But algortnm works - since we don't get to it, we get the 4

So the point is we can't use any peak finding algorithm, only if we continue our middle Column peak finding algortim

middle column Will always make matrix smaller + smaller and then find peak on that

Proof 1st step

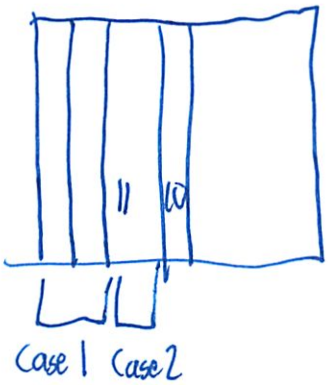
Case 1 Return a peak to left of middle column

Case 2 " " " " right " " "

- we know will be a peak on the right
- maximal element of row just to the right is still a peak

(4)

2nd step



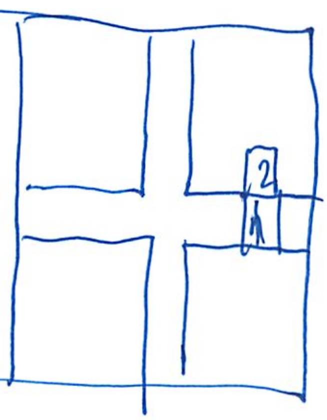
Case 1: return a peak in left side not adj to middle column

Case 2: return a peak in the col adj to middle

? Use diff methods to prove next step for each case

Still need to prove peak we returned is a maximum

Cross algorithm



← Then recurse on this submatrix

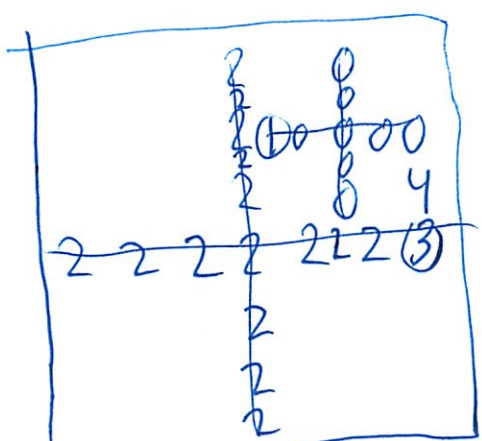
induction
↓
Ind. Hyp: Picking max always returns a peak in the submatrix

(5)

But does it also mean a peak I return is also a peak of the larger matrix

└ The cols near the side 

But didn't we already check the ^originals columns for the max?

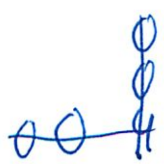


So it returns a 1
but this is not enough to prove a peak on the whole array

That algorithm does not work, since proof given in class does not work

---

Could we draw a cross that goes through the max each time



Correct
But slow O(nm) ← bad worse case, leads thru whole thing

Cross algorm. and proof of col algo. wrong in lecture

Since every cross we see is bigger than the one before

So we can fix original cross algorithm

1. keep track of the value, location of the max that we've seen before

- Every time we find cross - check if max value we got is more than stored value if less, then go to the quadrant that containes the old (stored) value

- Difficult to implement - lots of "book keeping" and case logic

- Peak must be as least as good as what we've seen before

- Check right after we find a max in the cross

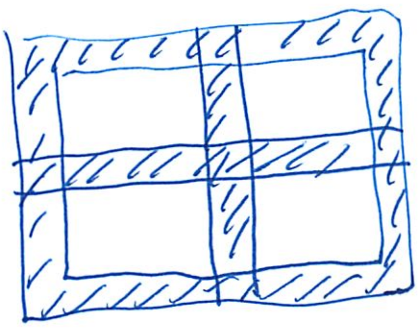
- Quad after doing that latest cross

(2)

So if even # of rows + cols
- we don't really care
- just pick one
- doesn't matter for asymptotics as well

This is $O(m+n)$

Fix 2. Easier to write code for
Look at middle cross, plus border



Take max in whole shaded area
Recurse in that submatrix
Then do look left, right thing
On every recursive call, draw border as well
This is particularly slow $3n + 3m$

(8)

The best algorithm is none of these

∟ Randomize best

Randomize (amusing aside)

linear time

Pick $n+m$ random locations

Look at all

Find largest ones

Hill climb from there

Worst case $O(nm)$

Can prove w/ probability

$P\left(\frac{1}{\varepsilon}(m+n) \text{ time}\right) \leq \frac{1}{n^{\Theta(\frac{1}{\varepsilon})}}$

Prob decreases exponentially

(4)

Cross Runtime

$$T(m,n) = T(m/2, n/2) + \Theta(m+n)$$

$$= \Theta(m+n) + \Theta\left(\frac{m+n}{2}\right) + \Theta\left(\frac{m+n}{4}\right) + \dots$$

falls off geometrically

So

$$= \Theta(m+n)$$

Window Runtime

$$T(m,n) = T\left(\frac{m}{4}, \frac{n}{4}\right) + \Theta(m+n)$$

amt of work done at each level decreases

1D Peak Finding

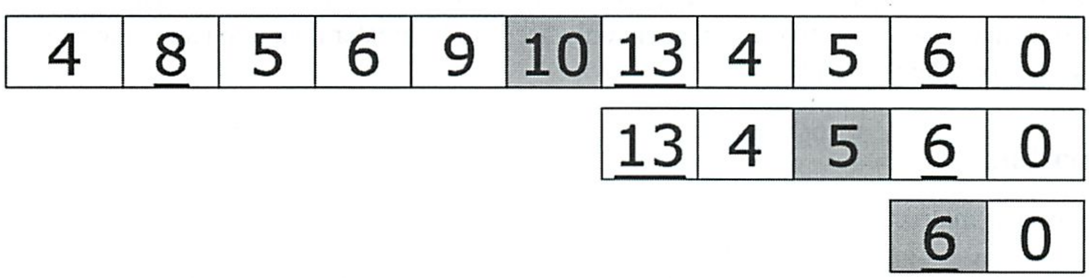
Objective

Given an array A with n elements, find the index i of the peak element $A[i]$ where $A[i] \geq A[i-1]$ and $A[i] \geq A[i+1]$. For elements on the boundaries of the array, the element only needs to be greater than or equal to its lone neighbor to be considered a peak. Or, say $A[-1] = A[n] = \infty$.

Algorithm

Given an array A with n elements:

- Take the middle element of A, $A[\frac{n}{2}]$, and compare that element to its neighbors
- If the middle element is greater than or equal to its neighbors, then by definition, that element is a peak element. Return its index $\frac{n}{2}$.
- Else, if the element to the left is greater than the middle element, then recurse and use this algorithm on the left half of the array, not including the middle element.
- Else, the element to the right must be greater than the middle element. Recurse and use this algorithm on the right half of the array, not including the middle element.



Runtime Analysis

When we recurse, we reduce size n array into size $\frac{n}{2}$ array in $O(1)$ time (comparison of middle element to neighbors). Show recursion in the form of "Runtime of original problem" ="Runtime of reduced problem" + "Time taken to reduce problem". Then use substitution to keep reducing the recursion.

$$T(n) = T(\frac{n}{2}) + c \quad (1)$$
$$T(n) = T(\frac{n}{4}) + c + c \quad (2)$$
$$T(n) = T(\frac{n}{8}) + c + c + c \quad (3)$$
$$T(n) = T(\frac{n}{2^k}) + ck \quad (4)$$
$$\text{Substitute } k = \log_2 n \quad (5)$$
$$T(n) = T(\frac{n}{2^{\log_2 n}}) + c\log_2 n \quad (6)$$
$$= T(1) + c\log_2 n \quad (7)$$
$$= O(\log n) \quad (8)$$

2D Peak Finding

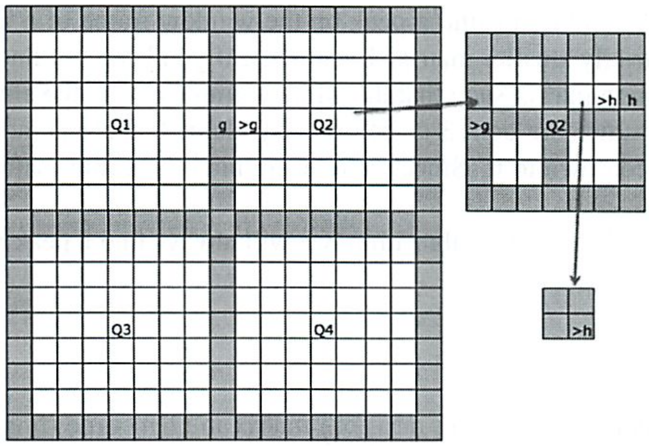
Objective

Given an nxn matrix M, find the indices of a peak element $M[i][j]$ where the element is greater than or equal to its neighbors, $M[i+1][j], M[i-1][j], M[i][j+1], and M[i][j-1]$. For elements on the boundaries of the matrix, the element only needs to be greater than or equal to the neighbors it has to be considered a peak.

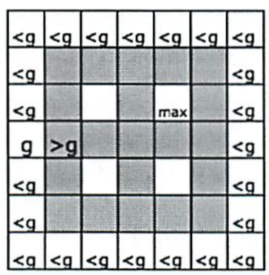
Algorithm

Given an nxn matrix M:

- Take the "window frame" formed by the first, middle, and last row, and first, middle, and last column. Find a maximum element of these $6n$ elements, $g = M[i][j]$.
- If g is greater than or equal to its neighbors, then by definition, that element is a peak element. Return its indices (i, j).
- Else, there's an element that neighbors g that is greater than g. Note that this element can't be on the window frame since g is the maximum element on the window frame, thus this element must be in one of the four quadrants. Recurse and use this algorithm on the matrix formed by that quadrant (not including any part of the window frame)



Proof of Correctness



Claim 1: If you recurse on a quadrant, there is indeed a global peak in that quadrant.

Proof: The quadrant we selected contains an element larger than g. Thus we know that the maximum element in this quadrant must also be larger than g. Since g is the maximum element surrounding this quadrant, the maximum element in this quadrant must be larger than any element surrounding this quadrant. This element must be greater than or equal to all of its neighbors since it is greater than all elements within the quadrant and directly outside of the quadrant, so the maximum element in this quadrant must be a global peak.

Claim 2: If you find a peak on the submatrix, then that peak is a global peak.

Proof: The window frame of the submatrix contains an element larger than g. Say m is the maximum element on the window frame. Since g is the largest element directly surrounding the submatrix, that means m is larger than all the elements surrounding the submatrix. If m is a peak in the submatrix and m is on the boundary, m must be a global peak since it is guaranteed that m is greater than any neighbors outside the scope of the submatrix. If m is a peak in the submatrix and m is not on the boundary, then clearly m is greater than or equal to its four neighbors and thus is a global peak.

Claim 3: You will always find a peak on the submatrix

Proof: In the case that we don't find a peak on the window frame of a matrix, we recurse to try to find a peak in a strictly smaller matrix. Eventually, if you keep not finding a peak, you will recurse into a small enough matrix such that the window frame covers the entire matrix (i.e. if the number of rows and columns are both 3 or below). By claim 1, there is indeed a global peak in this matrix if we recursed down to it. Since we're examining the entire matrix, we must find that global peak.

By claim 2 and claim 3, using this algorithm, we will always find a peak and that peak will be a global peak.

Runtime Analysis

When we recurse, we reduce nxn matrix into $\frac{n}{2}$x$\frac{n}{2}$ matrix in $O(n)$ time (finding the maximum of $6n$ elements). Show recursion in the form of "Runtime of original problem" ="Runtime of reduced problem" + "Time taken to reduce problem". Then use substitution to keep reducing the recursion.

$$T(n) = T(\frac{n}{2}) + cn \tag{9}$$

$$T(n) = T(\frac{n}{4}) + c\frac{n}{2} + cn \tag{10}$$

$$T(n) = T(\frac{n}{8}) + c\frac{n}{4} + c\frac{n}{2} + cn \tag{11}$$

$$T(n) = T(1) + cn(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}...) \tag{12}$$

$$= O(n) \tag{13}$$

Algorithms Book

2/12

Intro Reading

Algorithm = well defined computational procedure

has inputs and outputs

Can define problem formally

Sorting

Input A seq of n #s $\langle a_1, a_2, a_3, \ldots a_n \rangle$

Output A permutation (reordering) $\langle a_1', a_2', a_3', \ldots a_n' \rangle$

Such that $a_1' \leq a_2' \leq \ldots a_n'$

Correct - if for every input instance it halts w/ correct output

└ a correct alg. solves the problem

Data structure - way to store + org data

└ diff ones for diff purposes

Efficiency "insertion sort vs merge sort

(2)

Chap 2 Getting Started / Insertion Sort

- Same sorting problem as before
- #s to sort called <u>keys</u>
- assume input = array

[Uses <u>Psudo code</u> - eaiser to read]

- insertion sort
  - have ~~stack~~ fan of cards in hand
  - start w/ 0 cards
  - add 1 at a time in proper place

A[1......n] ← #s to pe sorted

  - but rearanges <u>in</u> the array (it moves #s around)
- (psudo code in book)
- <u>loop invarient</u> (remembering) something that never changes as loop is iterated

(3)

3 parts of loop invarient

Initalization: true prior to 1st iteration

Maintance: it remains true on each iteration

Termination: when loop terminates invariant gives useful property that helps to show alg. is correct

(like induction)
- base case
- inductive step
- so true at end!

* check in for loop? after 1st iteration initial assignment, but before first test

Formally should prove for both loops

i = j = e
- both i and j set to e

otherwise petty similar and/or short circiting

④

Analying

Use RAM ~~mafle~~ model

- └ ie one at a time
- only typical computer commands
- each takes constant time
- 2^k = constant time (shift left / right)
- no caches, etc

Pick a way to represent input size

└ insertion sort → # of elements

Running time → # of steps

So find cost for each step, and # times each step runs

└ usually constant

line	cost	times
1	c_1	n
2	c_2	$n-1$
3	0	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=2}^{n} t_j$
6	c_6	$\sum_{j=2}^{n} (t_j-1)$
7	c_7	$\sum_{j=2}^{n} (t_j-1)$
8	c_8	$n-1$

(5)

So add that all up

$$T(n) = \sum_{\text{lines}} \text{cost} \cdot \text{times}$$

Can have good/easy cases

└ Insertion sort → if already sorted!

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_3 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$
$$= an + b$$

↑ a linear function of n

Worst case

↳ Insertion sort → Sorted backwards

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$
$$= an^2 + bn + c$$

↑ quadratic function of n

6)

We care about <u>worst case</u>

- upper bound
- can actually occur pretty often
- average case usually as bad as worst case
  - <u>Insertion sort</u> $t_j = \frac{j}{2}$ unsorted

    So still <u>quadratic</u>

Can look at <u>probability</u> of each case

<u>Order of growth</u>

→ Only care about the leading term

- an^2

- not the other terms
- not c_i

And we don't really care about a (leading term coefficient)

So say $\Theta(n^2)$

↑ Theta

Most informative for <u>large algorithms</u>

7

Designing Algorithms

Some basic patterns

We used incremental for insertion sort

Can use divide + conquer

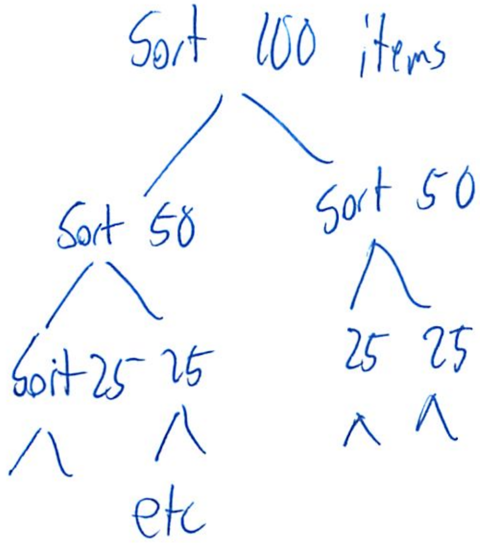
Divide + conquer

break into smaller ^ problems [same]

Solve recursively

Combine results

merge sort does it divide and conquer



(8)

Merge (A, p, q, r)

$p \leq q < r$ indices to array A

A

$A[p \ldots q]$ $A[q+1 \ldots r]$

$\Theta(n)$

$\leftarrow n = r - p + 1$

Use sentinel value to mark end of deck

↳ here ∞

(watched video on Wikipedia - much clearer)
(the recombining operation here is difficult)

You ~~always~~ split it down to 2 cards
Then put the 2 cards in a pile
So have 2 piles 2 cards each
You then pick the smallest card from the 2 top cards
And put it face down in output deck
When input deck empty, you do other 2 piles of 2 cards
Then 2 piles of 4 cards, etc

(9)

(book was confusing since code was just for merging)

Can prove loop invariant

Can find running time by looking at the recurrence

For small problem $n \le c \rightarrow \Theta(1)$

↑ some constant

Each division → a subproblems

< $\frac{1}{b}$ size of original

Merge sort → $a = b = 2$

Takes $T(n/b)$ to solve one subproblem $\frac{n}{b}$

So takes $aT(n/b)$ to solve a of them

Takes $D(n)$ to divide and $C(n)$ to combine

↑ ?? constant ??

?or did we include this info earlier?

So $T(n) = aT(n/b) + D(n) + C(n)$

So for worst case merge sort of n #s

<u>Divide</u> finding middle just constant

$$D(n) = \Theta(1)$$

<u>Conquer</u> Solve two subproblems each n/2

$$2T(n/2)$$

<u>Combine</u>

Merge takes $\Theta(n)$ on n-array

$$C(n) = \Theta(n)$$

↑(I knew it was not constant!)

$\Theta(n) + \Theta(1) = \Theta(n)$
Just take largest term
since linear fn of

In chap 4 see $T(n) \to \Theta(n \lg n)$ which is
<u>slower than linear</u> (for large inputs)

But
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{if } n > 1 \end{cases}$$

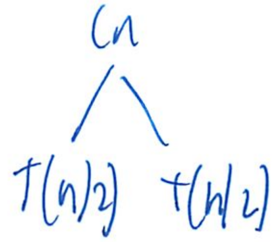
↑diff cs, so pick upper bound

(11)

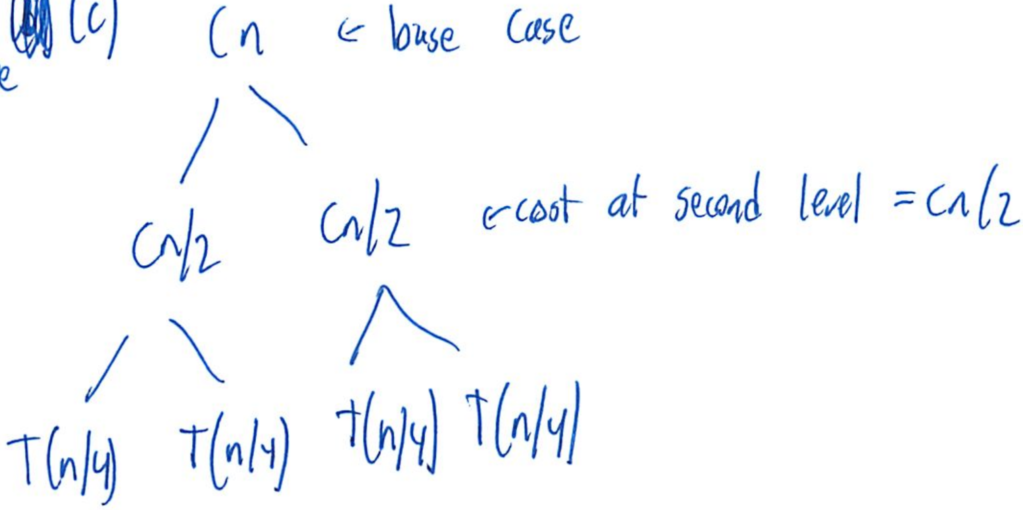
(I want to learn this)

Recursion Tree

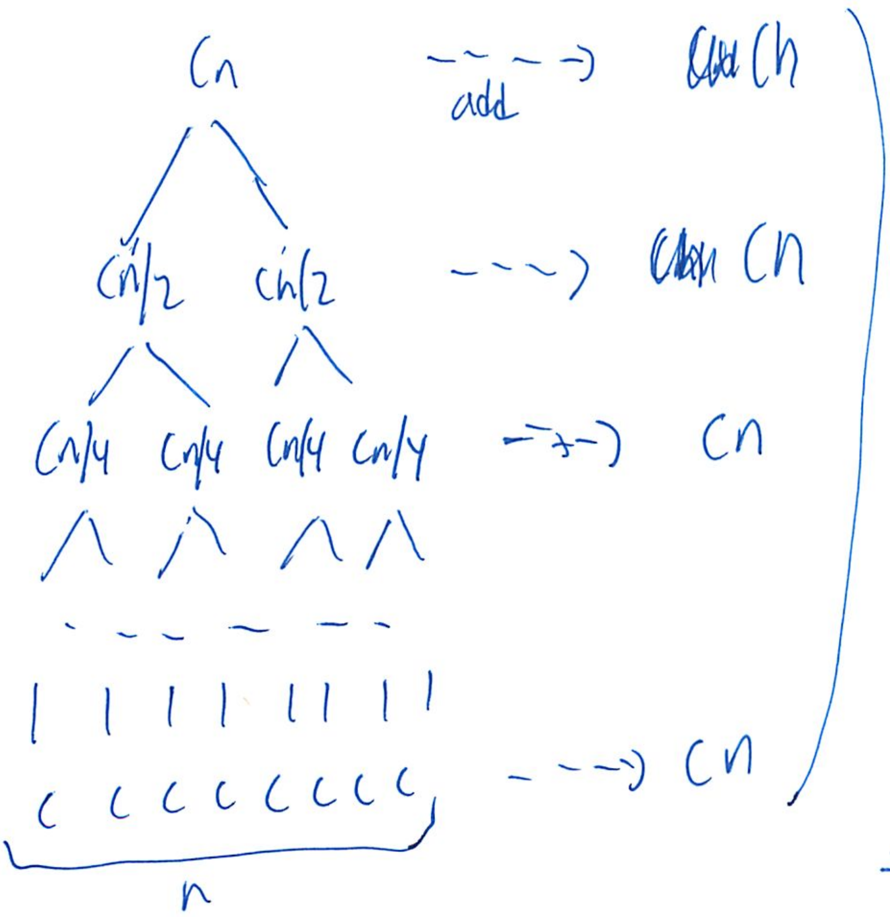
(a) $T(n)$ —break down→ (b)



break down more ↓ (c)



even more ↓ (d)



$\log_2 n + 1$

$lg = \log_2$

$= cn \lg n + cn$

$= \Theta(n \lg n)$

12

But how did we find tree has lg n + 1 levels?

Try examples (inductive)

$n=1$ → tree has 1 level

? ~~# leaves~~ $\lg(1) = 0$

\# leaves so $\lg(n) + 1$

$n = 2^i$ $\underbrace{\lg(2^i) + 1}$ $= i + 1$

? # levels ~~# leaves~~

$n = 2^{i+1}$ $\lg(2^{i+1}) + 1 = (i+1) + 1$

? total #s of level

~~So total # of levels lg n + 1~~

(don't really get)

└ they were trying to do an inductive proof

WP: logarithm

└ The log of a # is the exponent by which another fixed value has to be raised to produce that #

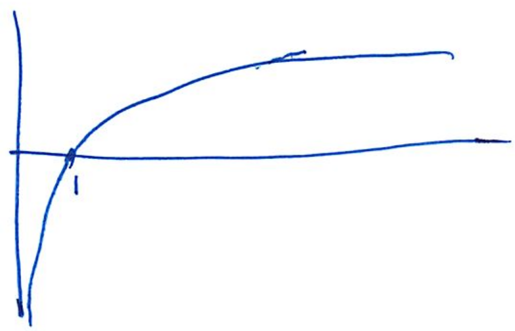
(13)

ie log of 1000 to base 10 = 3

$10^3 = 1000$

so $\log_{10}(1000) = 3$

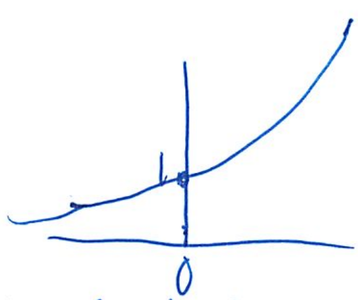
(why do I keep forgetting this?)



Self Explore: So something growing the log of is growing 2^n, right?

Like a lenght of # a binary # can represent

n	2^n
1	2
2	4
3	8
4	16
5	32



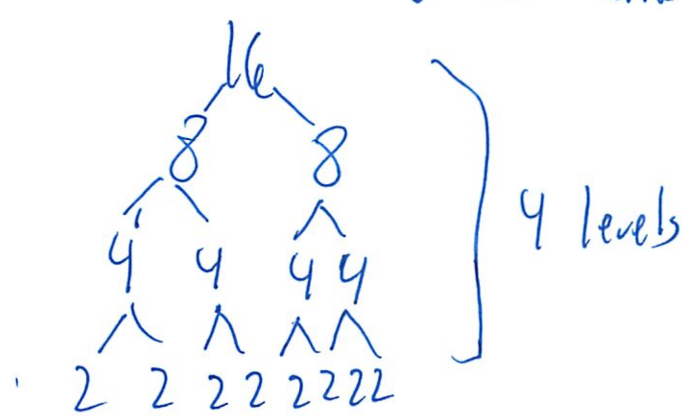
? So like opposet of logrithm?

(14)

n	$\log_2(n)$
1	0
2	1
3	1.58
4	2
5	2.2
8	3
16	4

↑ log is growing slower than constant

its like splitting stuff
like 16 cutting in half



↑ as far as merge sort goes

(15)

Chap 3

Growth of Functions

$\Theta(\ )$

don't care about extra precision

want the asymptotic efficiency

as limit as input ↑

$T(n)$ = worst case running time

generally talk about running time but

also can talk about space, etc

Θ-notation

We said insertion sort $T(n) = \Theta(n^2)$

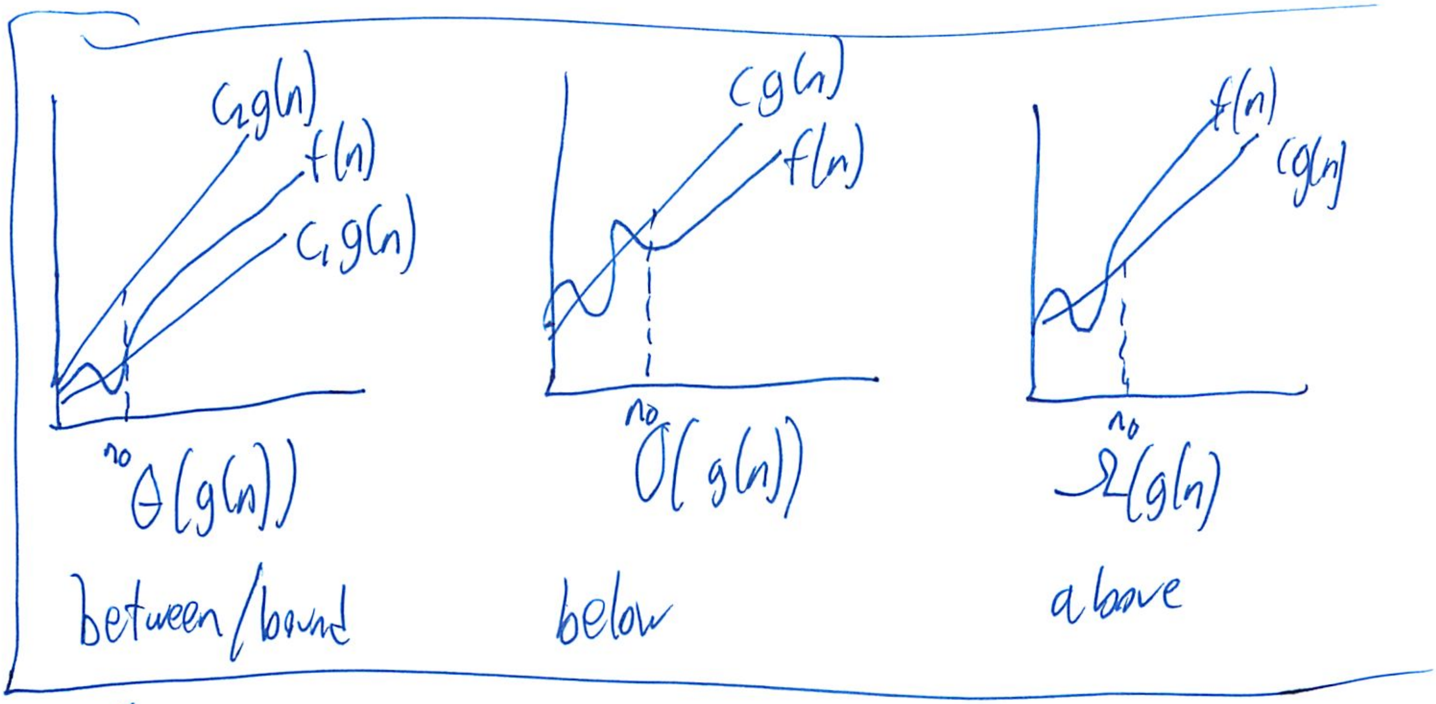
$\Theta(g(n)) = \{ f(n)$ there exists + constants

c_1, c_2, n_0 such that

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$

for all $n \ge n_0$

(16)



Θ is a set

$f(n) \in \Theta(g(n))$

but we write

$f(n) = \Theta(g(n))$

$g(n)$ = asymptotically tight bound

assume all asy nonneg

---

example

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

Find

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

Divide n^2

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

(17)

Since $n_0 \geq 1$ we want $n \geq 1$

~~Say $n = 1$~~

~~$c_1 \leq \frac{1}{2} - 3 \leq c_2$~~

So $c_2 \geq \frac{1}{2}$

For $n \geq 7$ $c_1 \leq 1/14$

So $c_1 = \frac{1}{14}$ $c_2 = \frac{1}{2}$ $n_0 = 7$

Try it out

If $n = 1$ $\frac{1}{2} - 3$

$-2.5 \leq c_2$

want non neg right?

So where is this 1?

$$\frac{1}{2} - \frac{3}{n} = 1$$

$$-\frac{3}{n} = -\frac{1}{2}$$

$$\frac{3}{n} = \frac{1}{2}$$

$$6 = n$$

So why $n \geq 7$?

(18)

If $n=6$

$c_1 \leq 0$

~~So where is $c_1 \leq \frac{1}{14}$ from?~~

~~Oh if n is more~~

~~If $n = 12$~~

~~$\frac{1}{2} - \frac{1}{4} = \frac{1}{4}$~~

~~Now $c_1 \leq \frac{1}{4}$~~

~~If $n = \infty$~~

~~$\frac{1}{2} - \frac{3}{\infty}$~~

~~$3/\infty \approx 0$~~

~~$c_1 \leq \frac{1}{2}$ larger?~~

← Oh duh $\frac{1}{2} - \frac{1}{2} = 0$ can't have, right

So make it larger

$n=7$

$\frac{1}{2} - \frac{3}{7} = \frac{1}{14}$

Ahh so $c_1 \leq \frac{1}{14}$

Now other side

$\frac{1}{14} \leq c_2$

But this is when n is large

$\frac{1}{2} - \frac{3}{\infty} = \frac{1}{2} - 0 = \frac{1}{2}$

$c_2 \geq \frac{1}{2}$

✓ Nice

(19)

O-notation

only asy upper bound
"big Oh"
: no worse then

$\Theta(g(n)) \subseteq O(g(n))$

Can say $n = O(n^2)$

: n is no worse than n^2
(silly to claim)

Ω-notation

asy lower bound
"big omega g of n"
: no better than ✓
"at least"

So insertion sort is $\Omega(n)$ since best case $\Theta(n)$
$O(n^2)$ " worst " $\Theta(n^2)$

20

When in a formula ↓

(ie $2n^2 + \Theta(n)$)

that means $2n^2 + f(n)$ where $f(n)$ = some function in the set $\Theta(n)$

So can say $T(n) = 2T(n/2) + \Theta(n)$
↑ hide the rest in here

little
o-notation

upper bound that is not asy tight

big
O-notation may or may not be asy tight

So $2n = O(n^2)$ $2n = o(n^2)$
$2n^2 = O(n^2)$ $2n^2 \neq o(n^2)$

little
ω-notation

Again that a lower bound is not asy tight
little omega (ω Ω)

(skipping the other properties - from 6.042
- seem unimportant here)

3.2 Other notation

Monotocity

$f(n)$ montonically increasing if $m \leq n$ implies $f(m) \leq f(n)$

aka always ↑

Strictly increasing if $m < n$ implies $f(m) < f(n)$

Floors or ceilings

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

Modular

remainder

Polynomial

polynomal in n of degree d

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

↑ coeffitients of the polynomial

$$p(n) = \Theta(n^d)$$

Exponentials

$a^0 = 1$

$a^1 = a$

$a^{-1} = \frac{1}{a}$

$(a^m)^n = a^{mn} = (a^n)^m$

⇕ I think I get this now

$a^m a^n = a^{m+n}$

Logarithms

$\lg n = \log_2 n$

$\ln n = \log_e n$

$\lg^k n = (\lg n)^k$

$\lg(\lg(n)) = \lg \lg n$

$a = b^{\log_b a}$

$\log_b a^n = n \log_b a$

$\log_b a = \frac{1}{\log_a b}$

Factorials

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

(23)

Functional iteration

$f^{(i)}(n)$ is $f(n)$ applied i times

Iterated log

$\lg^* n$

"log star of n"

as above w/ $f(n) = \lg n$

~~not~~ $\lg^{(i)} n \neq \lg^i n$

$\lg^* n$ grows very slowly!

n	$\lg^* n$
2	1
4	2
16	3
65536	4
2^{65536}	5

Fibonacci #s

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$

ϕ = golden ratio

6.006 L3

2/14

(Signing valentines)

Today
New data structure: linked list

Runway reservation system
- simple solution

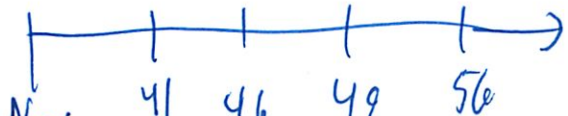
And Binary Search tree

Next time: Balanced Search Tree

Runway system

1 runway
landings only
allow reservations for time t
3 min window for planes from t

example



So requests 44 x - since 46   20 x - past
53 ✓

②

Obvious ~~...~~ linked list

traverse list looking for conflicting requests

but is O(n)

Can we do better?

Other options ① keep R as a sorted list — search slow, insert fast

- but still linear time

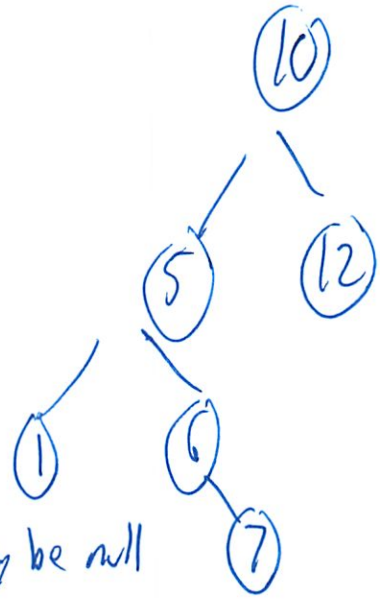
② keep R as sorted array — search fast, insert slow

- can access any value
- like binary search
- O(lg n) to find place
- but need to update array O(n)

③ ~~...~~ Binary Search tree — search fast, insert fast

(3)

Binary Search Tree



- each node x has
  - key[x] - the value
  - left[x] - child } may be null
  - right[x] - child }
  - p[x] - parent

(what I messed up on Google Interview - never knew about - did not think clearly about!)

(Best if you are told - should be able to figure out)

- Constraint

$$key[y] \leq key[x]$$

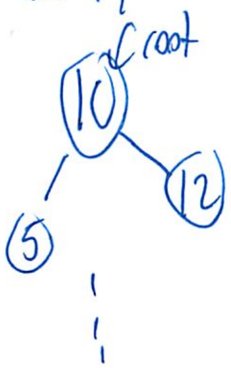
left subtree

$$key[y] \geq key[x]$$

right subtree

- How are Binary trees created?

- 10
- 12
- 5
- 1
- 6
- 7



} height

(thats not clear)

(a)

Supported Operations

insert (k)

- start from root
- ask value of root
- if k ≤ root value, go left
  $>$ right
- continue
- if no node to left or right, put it there

* We have only looked at the items along the path
So we can make inductive argument only look to left or right

find (k)

- start from root
- if k ≤ root value, go left
  $>$ right
- then find it
- or return can't find

(5)

delete(k)

- Start from root
- Same steps as find...
- then just remove the node
- if the node has children
  - go to right tree
  - take smallest element (by default the right entry)
  - and replace its value into the k node
  - instead of deleting the node

- full details next lecture

find min(x)

- finds min ele of tree rooted at x

delete min() - can be a node w/ a right child only

next-larger (x) - finds the one that has next largest key

If right[x] ≠ NIL then
 find min on right child ← (oh duh simple reuse functionality!)
Otherwise
 y ← p[x]
 while y ≠ NIL and x = right[y] do
  x ← y y ← p[y] → then return y

6

(essentally the first time the backbone turns...)
- are some subtiles when proving

Runway System

What if wanted to ask how many planes land at time ≤ t

in regular binary search tree - this is hard to answer. Need to explore the tree a lot in order to answer.

What if we tracked # nodes on left and right
Could calculate while we are inserting each node
So not much overhead
But good for answering this qu

Lets simplify more -> just track # of nodes in subtree rooted here (# left + # right)

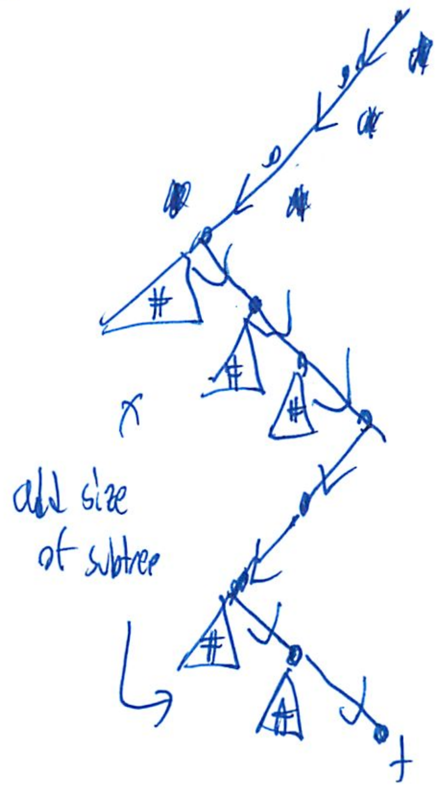
Why helpful?

- we walk down tree as though we were inserting t
- every node we went right, add 1 + size of subtree on the left
- (more details on slide)

↑ for backbone ↑ when we went right

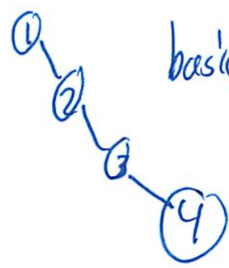
(7)

← go to left to find value



Complexity

Worst Case → $O(n)$
?height

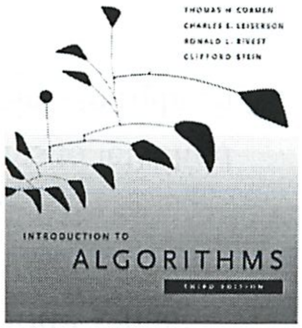


basically a sorted list

But next lecture we will fix so $O(\lg n)$

Note: correction $<$ is $\leq$ in #4 on PS 1

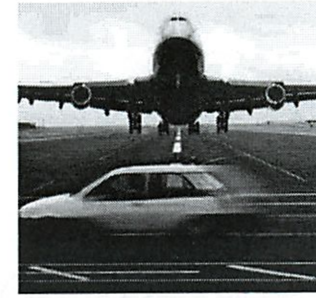
6.006- *Introduction to Algorithms*



Lecture 3

Prof. Costis Daskalakis

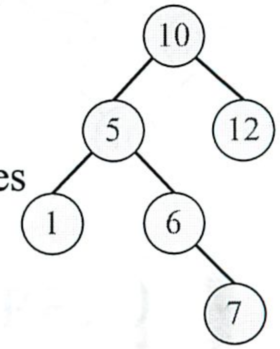
Overview



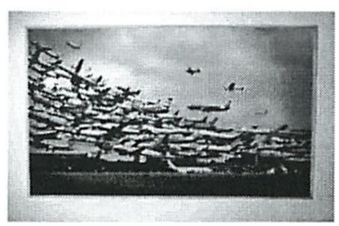
http://izismile.com/tags/Gibraltar/

- Runway reservation system:
  - Definition
  - How to solve with linked-lists
- Binary Search Trees
  - Operations
- Next time: Balanced Search Trees

Readings: CLRS 10, 12.1-3



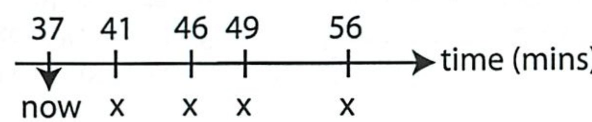
Runway reservation system



- Problem definition:
  - Single (busy) runway
  - Reservations for landings
    - maintain a set of future landing times
    - a new request to land at time t
    - add t to the set if no other landings are scheduled within < 3 minutes from t
    - when a plane lands, removed from the set

Runway reservation system

- Example



  - R = (41, 46, 49, 56)
  - requests for time:
    - 44 => reject (46 in R)
    - 53 => ok
    - 20 => not allowed (already past)
- Ideas for efficient implementation ?

2/14

Proposed algorithm

- (keep R as a linked-list)

```
init: R = [ ]

req(t): if t < now: return "error"
for i in range (len(R)):
if abs(t-R[i]) < 3: return "runway busy"
R.append(t)
```

- Complexity?
- Can we do better?

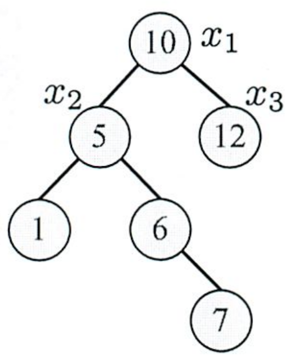
Some other options:

- Keep R as a sorted list:
  - on request t, it takes linear time to find the right location in the list where t needs to be inserted
  - before inserting t at found location check whether the numbers on the left and right of the location are ≤t-3 and ≥t+3 respectively
- Keep R as a sorted array:
  - takes O(log n) to find the place to insert new t
  - but still requires linear time to actually insert (requires shifting of elements)

Need best of both worlds:
***fast* insertion into *sorted* list**

Binary Search Trees (BSTs)

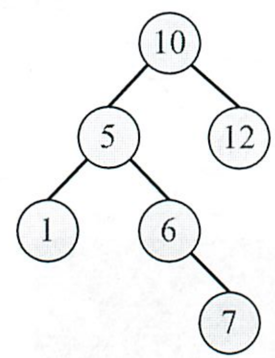
- A tree …
- …where each node x has:
  - a key[x]
  - three pointers:
    - left[x] : points to left child
    - right[x] : points to right child
    - p[x] : points to parent



- E.g. key[x_1]=10
- left[x_1]=x_2
- p[x_2]=x_1
- p[x_1]=NIL

Binary Search Trees (BSTs)

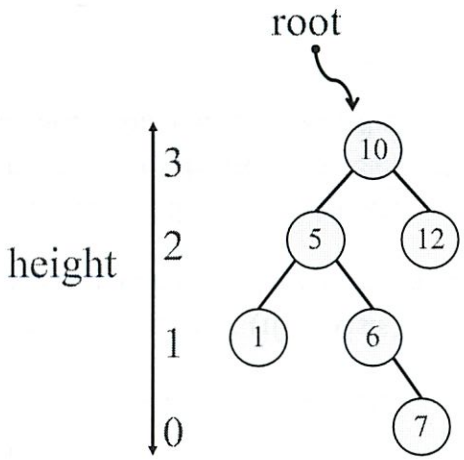
- ***Defining*** property (i.e. what makes it a binary SEARCH tree):
- for any node x:
  - for all nodes y in the left subtree of x:
    key[y] ≤ key[x]
  - for all nodes y in the right subtree of x:
    key[y] ≥ key[x]



- How are BSTs created?

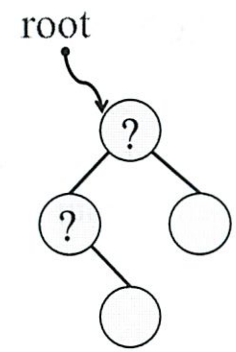
Growing BSTs

- Insert 10
- Insert 12
- Insert 5
- Insert 1
- Insert 6
- Insert 7



BST as a data structure

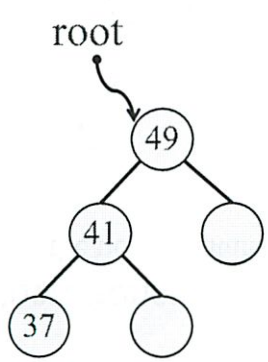
- Supported Operations:
  - insert(k): insert a node with key k at the appropriate location of the tree



e.g. insert(37)

BST as a data structure

- Supported Operations:
  - insert(k): insert a node with key k at the appropriate location of the tree

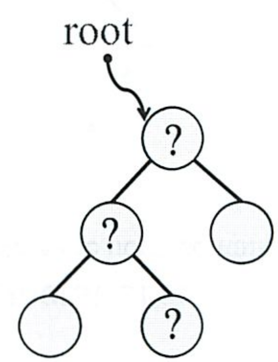


e.g. insert(37)

Aside: Can do the "within 3" check for reservation system during insertion.

BST as a data structure

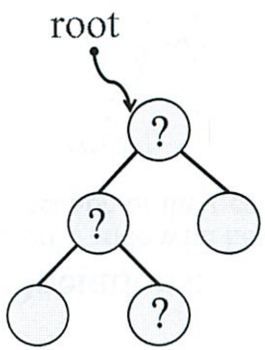
- Supported Operations:
  - find(k): finds the node containing key k (if it exists)



e.g. find(46)

BST as a data structure

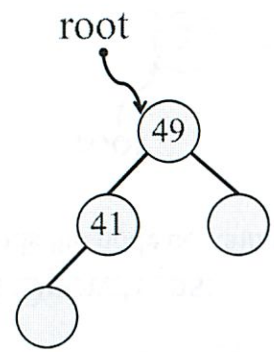
- Supported Operations:
  - delete(k): delete the node containing key k, if such a node exists



e.g. delete(46)

BST as a data structure

- Supported Operations:
  - delete(k): delete the node containing key k, if such a node exists



e.g. delete(46)

Question: What if we have to delete a node that is internal?
How do we fill in the hole? A: next lecture.

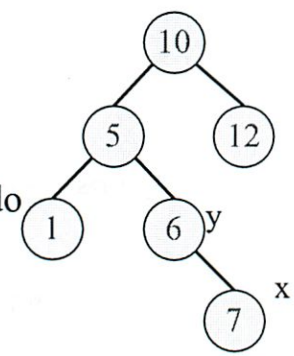
BST as a data structure

- Supported Operations:
  - insert(k): insert a node with key k at the appropriate location of the tree
  - find(k): finds the node containing key k (if it exists)
  - delete(k): delete the node containing key k, if such a node exists
  - findmin(x): finds the minimum of the tree rooted at x
  - deletemin(): finds the minimum of the tree and deletes it
  - next-larger(x): finds the node containing the key that is the immediate next of key[x]

Next-larger

next-larger(x):

- If right[x] $\neq$ NIL then
  return findmin(right[x])
- Otherwise
  - y $\leftarrow$ p[x]
  - While y$\neq$NIL and x=right[y] do
    - x $\leftarrow$ y
    - y $\leftarrow$ p[y]
  - Return y



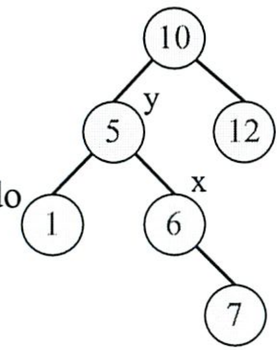
next-larger(⑤) = ⑥

next-larger(⑦)

Next-larger

next-larger(x):
- If right[x] ≠ NIL then return findmin(right[x])
- Otherwise
  y ← p[x]
  While y≠NIL and x=right[y] do
    - x ← y
    - y ← p[y]

  Return y



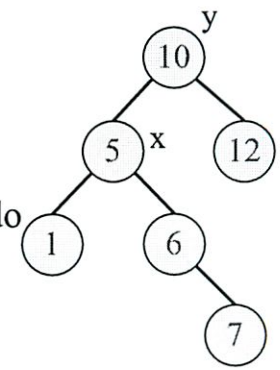
next-larger(⑤) = ⑥

next-larger(⑦)

Next-larger

next-larger(x):
- If right[x] ≠ NIL then return findmin(right[x])
- Otherwise
  y ← p[x]
  While y≠NIL and x=right[y] do
    - x ← y
    - y ← p[y]

  Return y

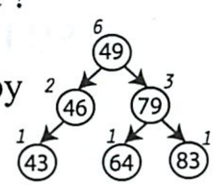


next-larger(⑤) = ⑥

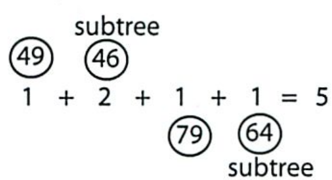
next-larger(⑦) = ⑩

Back to runway reservation system

- Introducing extra requirements: e.g. how many planes are scheduled to land at times ≤ t ?
- Augment the BST structure by keeping track of size of subtrees rooted at all nodes
- To figure out how many planes will land ≤ t:
  - Walk down the tree to find where key t would have been inserted in the tree…
  - … and for every node where you forked to the right:
    - add 1 + size of subtree on the left of that node



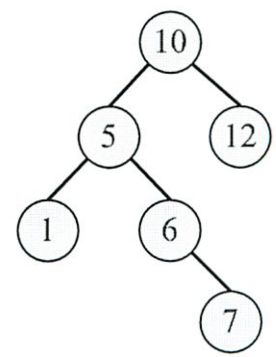
e.g. #planes land ≤ 80?



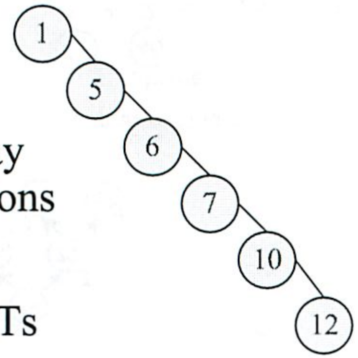
e.g. #planes land ≤ 75? A: 4

Analysis

- We have seen insertion, deletion, search, findmin, etc.
- How much time does any of this take ?
- Worst case: O(height)
  => height really important
- After we insert n elements, what is the worst possible BST height ?



Analysis



- n-1
- so, still O(n) for the runway reservation system operations
- Next lecture: balanced BSTs
- **Readings: CLRS 13.1-2**
- **Hw: notice correction in question 4: a '>' was turned to a '≥'**

6.006 Recitation

2/15

[I went to a diff hr b/c of a time conflict]
L 3PM

Shri said 10AM did
- binary search trees
- linked list

Recurrences

to analyze running time

$T(n)$ = time/work done on input of size n

e.g.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

^ input of size n

^ twice the time it takes if input is size n/2

^ essentially $O(n)$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$ expand

$$= 4T\left(\frac{n}{4}\right) + 2n$$ consolidate

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$$ expand

$$= 8T\left(\frac{n}{8}\right) + 3n$$ consolidate

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$ generalize

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$ generalize

but how many steps (k)

(2)

Go to $T(1)$ - which is base of recurrence

$\frac{n}{2^k} \approx 1$

$n = 2^k$

$k = \lg n$

Only care asy growth

$2^k < n < 2^{k+1}$ ← don't care about constants

So saying $n \approx 2^k$

* Here in how many steps we go

$k = \lfloor \lg_2 n \rfloor$

← we go at most that many steps

So

$2^{\lg n} T\left(\frac{n}{2^{\lg n}}\right) + n \lg n$

$= n T(1) + n \lg n$

↑ the constant ↑ grows fastest

$T(n) = O(n \lg n)$

(I think I am confused since he didn't specify the method)

(3)

Some general steps

A) <u>Substitution method</u>

1) Make a guess about growth
   - hard to do
   - upper bound
   - unless they tell you it and want you to prove it

2) Prove by induction

<u>Example</u> Guess $n \lg n$

Assume true $\forall\ k < n \rightarrow T(k) \approx k \lg k$

$\rightarrow T(k) \leq c\, k \lg k$

↑ actually want to show

↑ Constant

then function $T(k)$ becomes the upper bound $O(\,)$

$k < n$

We have

$T(n) = 2T\left(\frac{n}{2}\right) + n$

How can we prove this $k = n$?

$T(n) \leq c\, n \lg n$

Use original RHS

(4)

We can fix $T(\frac{n}{2})$

$$T(n) \le 2 \cdot c \cdot \frac{n}{2} \lg\left(\frac{n}{2}\right) + n$$
$$= c \cdot n \log\left(\frac{n}{2}\right) + n$$
$$= \underbrace{c \cdot n \lg(n)}_{\text{what we want}} \underbrace{- c \cdot n \cdot \lg 2 + n}_{\text{Show that this is} -}$$

that it is $\le cn\lg(n)$

$$-cn + n \le 0$$
$$c \ge 1$$

But do we have the base case of our induction?

Base case $k=1$

$$T(1) \le c \cdot 1 \cdot \lg 1 = 0$$

If base case fails, then proof is wrong
So change base case

New base case $k=2$

$$T(2) \le c \cdot 2 \cdot \lg 2 = 2c$$

(5) True w/ $c \geq \frac{T(2)}{2}$

~~We want to~~

In conclusion $c \geq \max\{1, \frac{T(2)}{2}\}$

(I'm confused - this TA much more confusing)

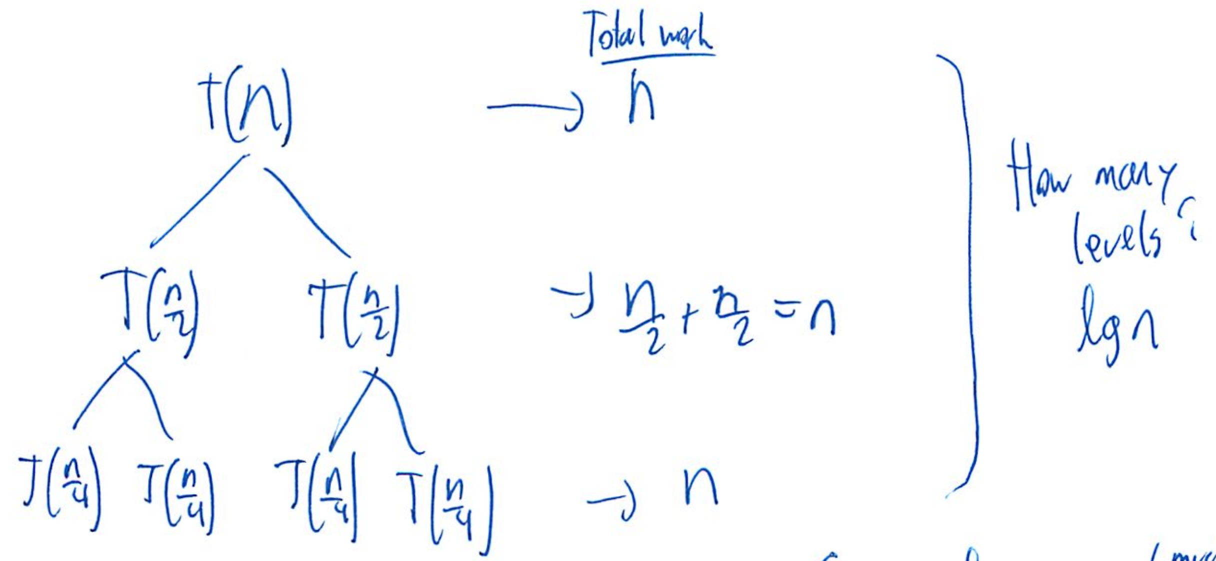
Need to make a guess and prove it

But if you're sure your guess is right - it can work

B) <u>Recursion Tree</u>

$T(n)$

Work at nth level is n



So $n \lg n$ (much easier!)

(6) (The 1st recitation I haven't really followed all semester...)

c) Master's Method

$a \geq 1$ $b > 1$

↑ branch factor

↑ how much shrink on recurrence

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

3 cases

1) $f(n) = O(n^{\lg_b a - \epsilon})$ $\epsilon > 0$

$f(n)$ is strictly less than $n^{\lg_b a - \epsilon}$

So if this happens

$$T(n) = \Theta(n^{\lg_b a})$$

2) $f(n) = \Theta(n^{\lg_b a})$

↳ $T(n) = \Theta(n^{\lg_b a} \cdot \lg n)$

(7)

3. $f(n) = \Omega(n^{\lg_b a + \epsilon})$ $\epsilon > 0$

↳ $T(n) = \Theta(f(n))$

Only true if $\exists c < 1$ s.t.

$a f(\frac{n}{b}) \leq c f(n)$ $\forall n \geq n_0$

BST

Augmentation in BST

↳ to find how many nodes have values < x

Food product

↳ has a freshness index, cost

eg (3, 10)

Q: Total cost of product w/ freshness index in range (a, b]

ie > a, ≤ b

(8)

Avilable products

(10,10) (5,3) (4,2) (12,15) (7,5) (16,20)

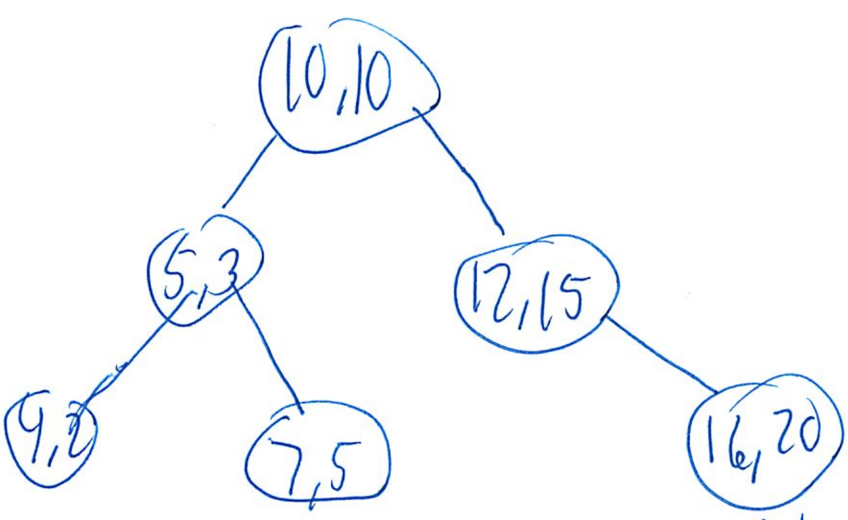
Interval [7,16]

So answer is

10 + 15 + 20 = 45

So do that thing we talked about in class w/ freshness index

ignore cost for now



Look at what cost for items freshness index ≤ 7 = S_1

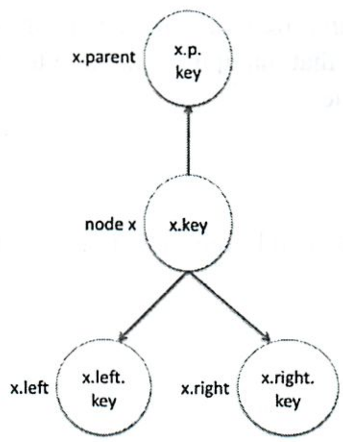
freshness index ≤ 16 = S_2

Now $S_2 - S_1$ = cost [7,16]

Binary Search Tree

A binary search tree is a data structure that allows for key lookup, insertion, and deletion. It is a binary tree, meaning every node of the tree has at most two child nodes, a left child and a right child. Each node of the tree holds the following information:

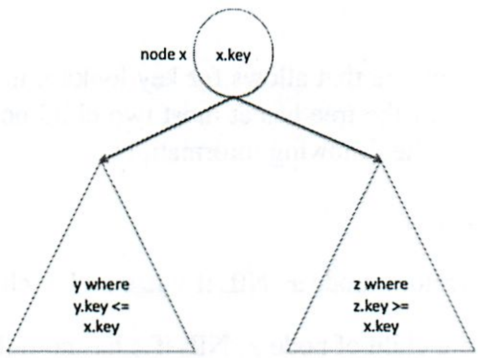
- x.key - Value stored in node x
- x.left- Pointer to the left child of node x. NIL if x has no left child
- x.right - Pointer to the right child of node x. NIL if x has no right child
- x.parent - Pointer to the parent node of node x. NIL if x has no parent, i.e. x is the root of the tree



Later on this week, we will learn about binary search trees that holds data in addition to the four listed above but for now we will focus on the vanilla binary search tree.

A binary search tree has two simple properties:

- For each node x, every value found in the left subtree of x is less than or equal to the value found in x
- For each node x, every value found in the right subtree of x is greater than or equal to the value found in x



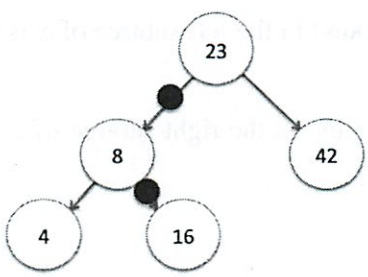
BST Operations

There are operations of a binary search tree that take advantage of the properties above to search for keys. There are other operations that manipulate the tree to insert new key or remove old ones while maintaining these two properties.

`find(x, k)`

Description: Find key k in a binary search tree rooted at x. Return the node that contains k if it exists or NIL if it is not in the tree

```
find(x, k)
  while x != NIL and k != x.key
    if k < x.key
      x = x.left
    else
      x = x.right
  return x
```

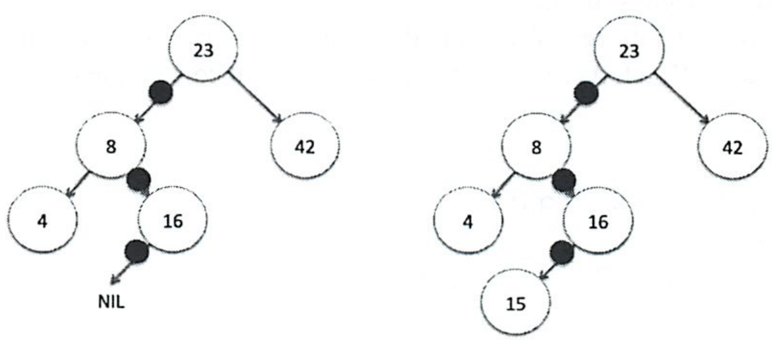


Analysis: At worst case, `find` goes down the longest branch of the tree. In this case, `find` takes $O(h)$ time where h is the height of the tree

`insert(x, k)`

Description: Insert key k into the binary search tree T

```
insert(T, k)
  z.key = k                       //z is the node to be inserted
  z.parent = NIL
  x = root(T)
  while x != NIL                  //find where to insert z
    z.parent = x
    if z.key < x.key
      x = x.left
    else
      x = x.right
  if z.parent = NIL               //in the case that T was an empty tree
    root(T) = z                   //set z to be the root
  else if z.key < z.parent.key //otherwise insert z
    z.parent.left = z
  else
    z.parent.right = z
```



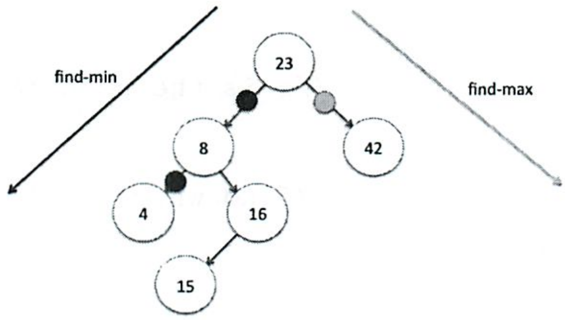
Analysis: At worst case, `insert` goes down the longest branch of the tree to find where to insert and then makes constant time operations to actually make the insertion. In this case, `insert` takes $O(h)$ time where h is the height of the tree

`find-min(x)` and `find-max(x)`

Description: Return the node with the minimum or maximum key of the binary search tree rooted at node x

```
find-min(x)
  while x.left != NIL
```

```
    x = x.left
  return x
```



Analysis: At worst case, `find-min` goes down the longest branch of the tree before finding the minimum element. In this case, `find-min` takes $O(h)$ time where h is the height of the tree

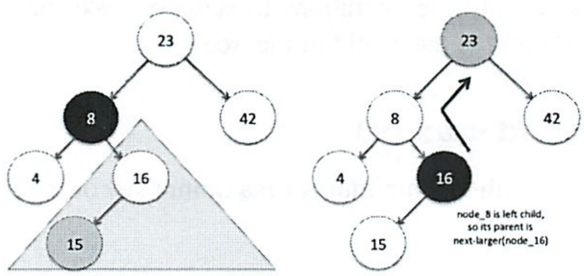
`next-larger(x)` and `next-smaller(x)`

Description: Return the node that contains the next larger (the successor) or next smaller (the predecessor) key in the binary search tree in relation to the key at node x

Case 1: x has a right sub-tree where all keys are larger than x.key. The next larger key will be the minimum key of x's right sub-tree

Case 2: x has no right sub-tree. We can find the next larger key by traversing up x's ancestry until we reach a node that's a left child. That node's parent will contain the next larger key

```
next-larger(x)
  if x.right != NIL                  //case 1
    return find-min(x.right)
  y = x.parent
  while y != NIL and x = y.right    //case 2
    x = y
    y = y.parent
  return y
```



Case 1: next-larger(node_8) = node_15 Case 2: next-larger(node_16) = node_23

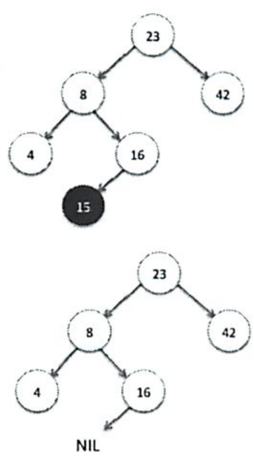
Analysis: At worst case, `next-larger` goes through the longest branch of the tree if x is the root. Since `find-min` can take $O(h)$ time, `next-larger` could also take $O(h)$ time where h is the height of the tree

`delete(x)`

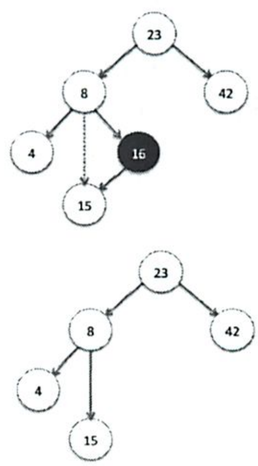
Description: Remove the node x from the binary search tree, making the necessary adjustments to the binary search tree to maintain its properties. (Note that this operation removes a specified node from the tree. If you wanted to delete a key k from the tree, you would have to first call `find(k)` to find the node with key k and then call `delete` to remove that node)

Case 1: x has no children. Just delete it (i.e. change parent node so that it doesn't point to x)
Case 2: x has one child. Splice out x by linking x's parent to x's child
Case 3: x has two children. Splice out x's successor and replace x with x's successor

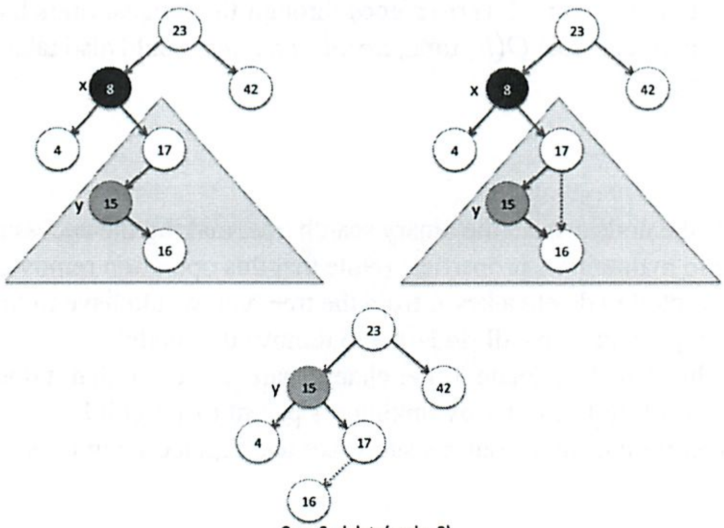
```
delete(x)
  if x.left = NIL and x.right = NIL    //case 1
    if x.parent.left = x
      x.parent.left = NIL
    else
      x.parent.right = NIL
  else if x.left = NIL                 //case 2a
    connect x.parent to x.right
  else if x.right = NIL                //case 2b
    connect x.parent to x.left
  else                                 //case 3
    y = next-larger(x)
    connect y.parent to y.right
    replace x with y
```



Case 1: delete(node_15)



Case 2: delete(node_16)



Case 3: delete(node_8)

Analysis: In case 3, `delete` calls `next-larger`, which takes $O(h)$ time. At worst case, `delete` takes $O(h)$ time where h is the height of the tree

inorder-tree-walk(x)

Description: Print out the keys in the binary search tree rooted at node x in sorted order

```
inorder-tree-walk(x)
  if x != NIL
    inorder-tree-walk(x.left)
    print x.key
    inorder-tree-walk(x.right)
```

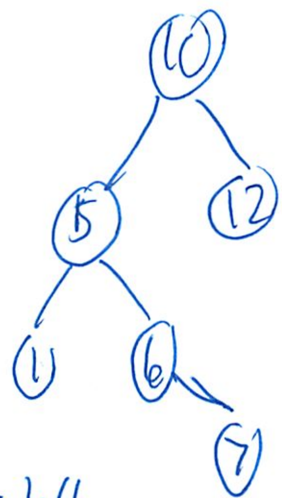
Analysis: `inorder-tree-walk` goes through every node and traverses to each node's left and right children. Overall, `inorder-tree-walk` prints n keys and traverses $2n$ times, resulting in $O(n)$ runtime

6.006 L4

2/16

Last time: Binary Search Treee
- important to be balanced

Today: Balanced tree AVL



lower → left
higher → right

Remove + **insert** – fairly simple in these examples

$O(h)$
↑ height

Can augment w/ tree size

Balanced height = $\log n$
Not Balanced height = n

✓ **Balanced**

Augment each node w/ useful info
Show invariant that tree is balanced

(2)

AVL Tree

- store its height (augmentation)
- leaf → height = 0
- nil → " = -1

* Invariant - height of left and right differ by ± 1

- Prove height = log n

  - n_h = min # of nodes of AVL of height h

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$n_h > 2 n_{h-2}$$

$$n_h > 2^{h/2}$$

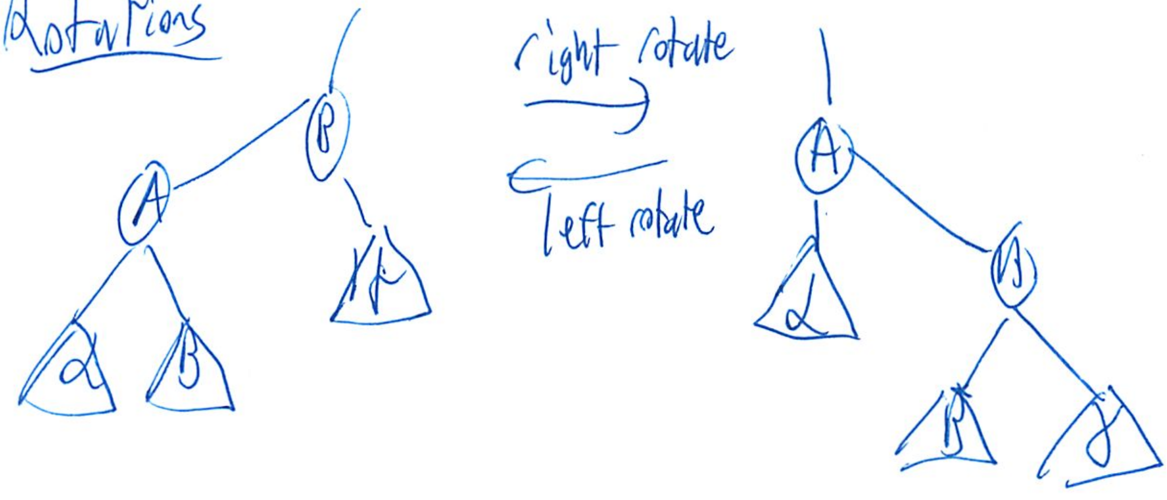
$$h = < 2 \lg n_h$$

- Optimal?

  (missed...)

(3)

Rotations

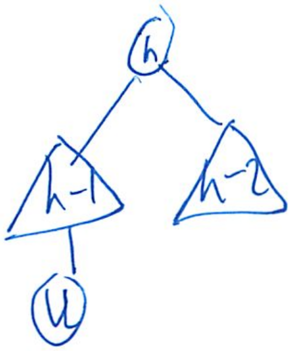


$$\forall a \in \alpha \quad \forall b \in \beta \quad \forall c \in \gamma$$

$$a \le A \le b \le B \le c$$

Insertions

- can create an imbalance



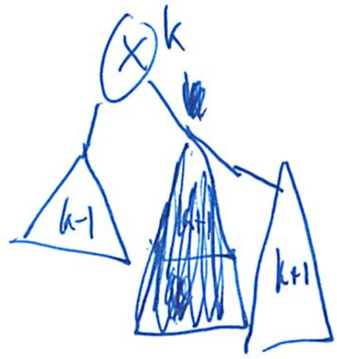
- so work way up the tree restoring balance

(4)

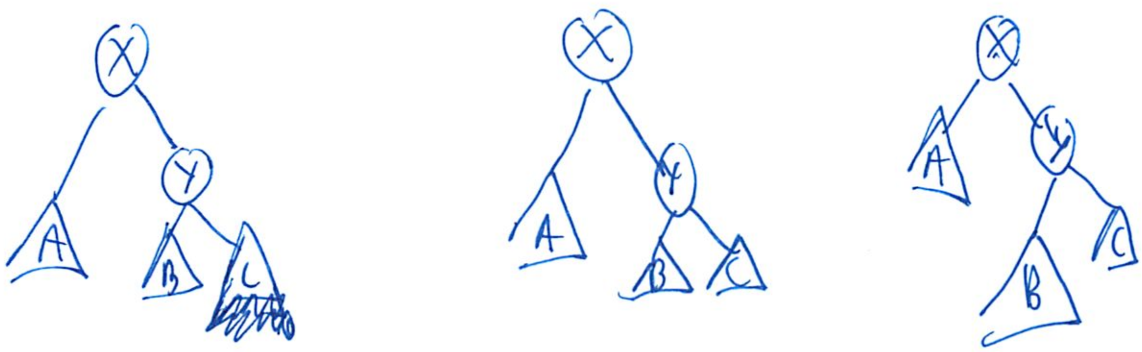
Balancing

X = lowest violating node

WLOG = x is right heavy
since right has more elements



3 cases
- others symetric



① Y right heavy ② Y balanced ③ Y left heavy

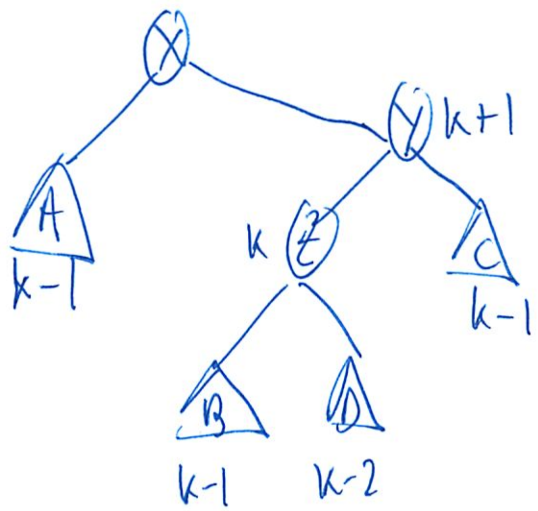
① Y right heavy
- Left rotate to restore balance

② Y balanced
- Left rotate to restore balance

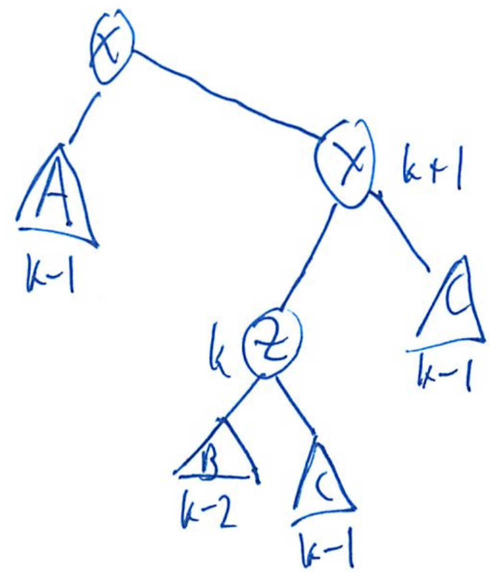
③ Y left heavy
- Left rotate makes smaller tree balanced - but not global tree

(5)

Can look into this in more depth



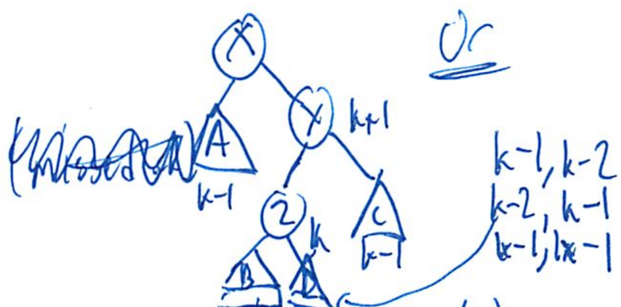
Or



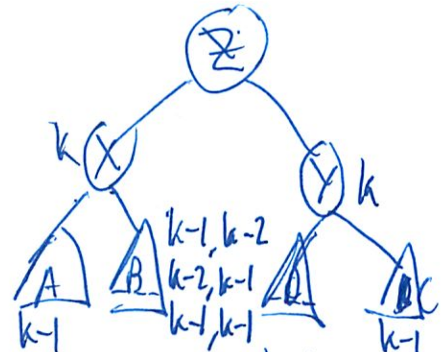
and z is left heavy

and z is right heavy

Or



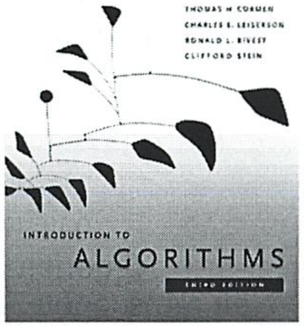
So we can right rotate(y)
left rotate(x)



tree now balanced
still binary search tree

So new operation called whimsy rotation
— which is

6.006- *Introduction to Algorithms*

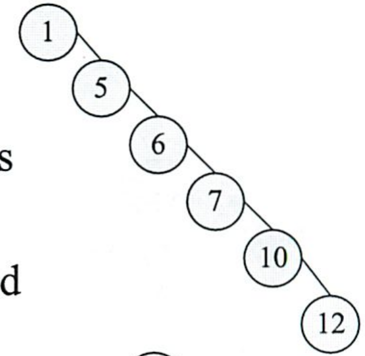


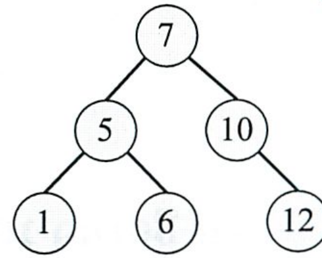
Lecture 4

Prof. Silvio Micali

Lecture Overview

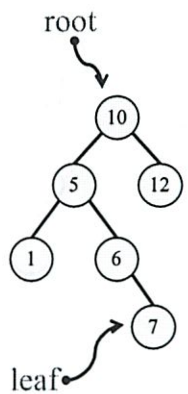
- Review: Binary Search Trees
- Importance of being balanced
- Balanced BSTs
  - AVL trees
  - Other balanced trees





Binary Search Trees (BSTs)

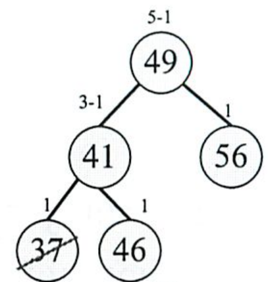
- Each node x has:
  - key[x]
  - Pointers: left[x], right[x], p[x]
- Property: for any node x:
  - For all nodes y in the **left** subtree of x:
    key[y] ≤ key[x]
  - For all nodes y in the **right** subtree of x:
    key[y] ≥ key[x]

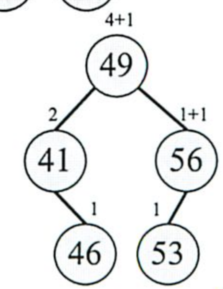


height = 3

BST Basic Operations

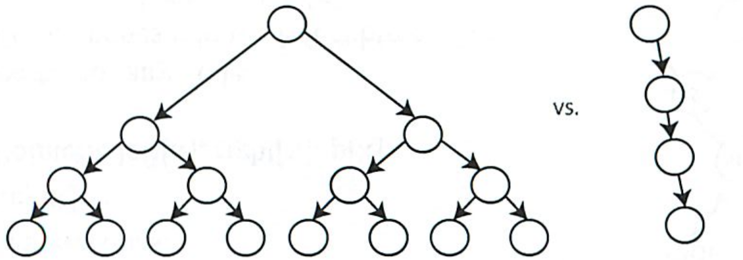
- Find, successor, min, …
- Remove an element (e.g., 37)
- Insert new element (e.g., 53)
- Delete & insert: O(h),
  where h is the height of the tree
- Useful to "augment" a BST (e.g., w/ tree size)





The importance of being balanced

for n nodes:



Perfectly Balanced

Path

$h = \Theta(\log n)$ $h = \Theta(n)$

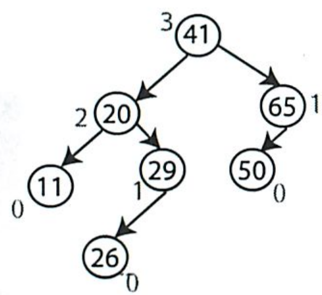
Balanced BST Strategy

- Augment every node with useful INFO
- Define a local invariant on INFO
- Show that invariant guarantees $\Theta(\log n)$ height
- Design algorithms to maintain INFO & invariant

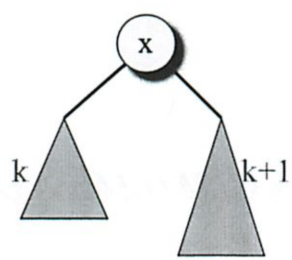
AVL Trees: Definition

[Adelson-Velskii and Landis'62]

- **INFO**: for every node, store its height ("augmentation")
  - Leaves have height 0
  - NIL has "height" -1



- **Invariant**: for every node x, the heights of its left child and right child differ by at most 1



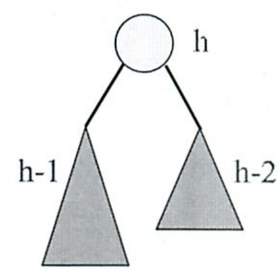
AVL trees have height Θ(log n)

- Let n_h be the minimum number of nodes of an AVL tree of height h
- We have $n_h \geq 1 + n_{h-1} + n_{h-2}$

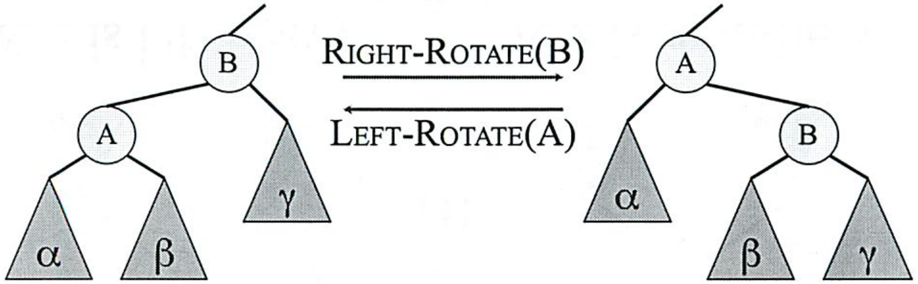
  $\Rightarrow n_h > 2n_{h-2}$

  $\Rightarrow n_h > 2^{h/2}$

  $\Rightarrow h < 2 \lg n_h$
- Optimal?

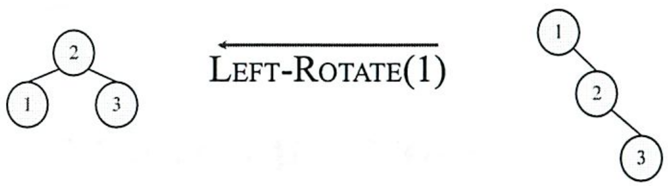


Rotations



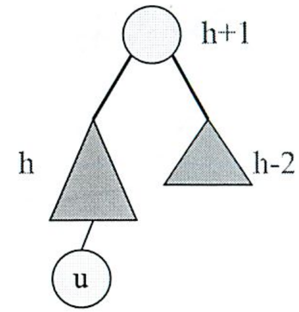
Rotations maintain the inorder ordering of keys:

- $\forall a \in \alpha \quad \forall b \in \beta \quad \forall c \in \gamma$: $\quad a \le A \le b \le B \le c$.



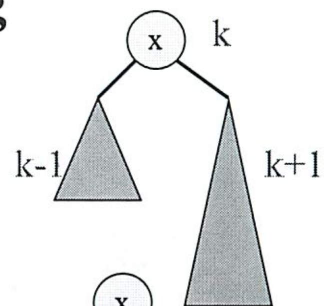
Insertions

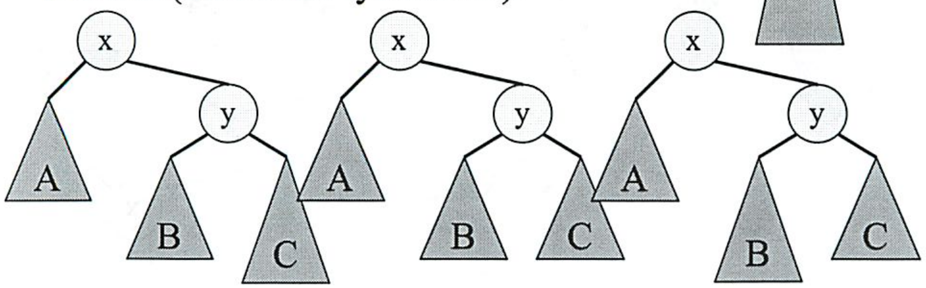
- Insert new node u as in the simple BST
  - Can create imbalance
- Work your way up the tree, restoring the balance



Balancing

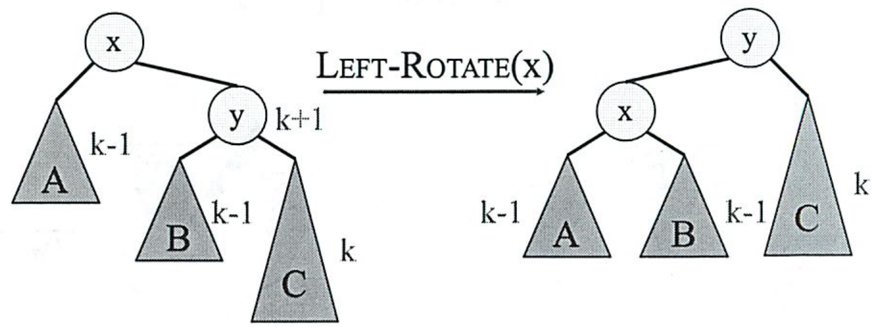
- Let x be the lowest "violating" node
- WLOG x is "right-heavy":
  Right[x] deeper Left[x]
- 3 Cases (others are symmetric):



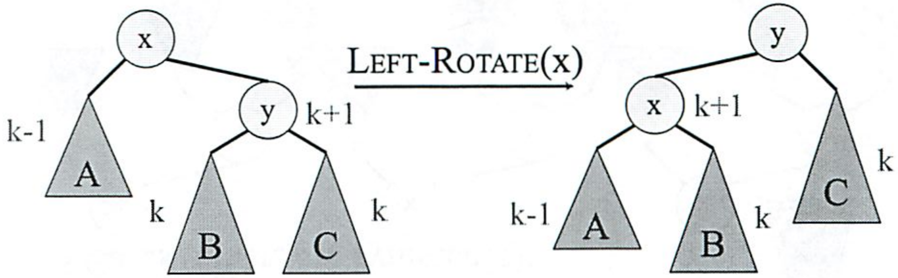


1. y right-heavy 2. y balanced 3. y left-heavy

Case 1: y is right-heavy

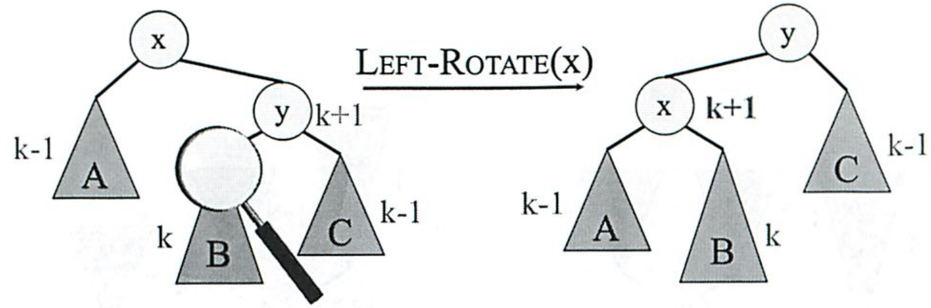


Case 2: y is balanced



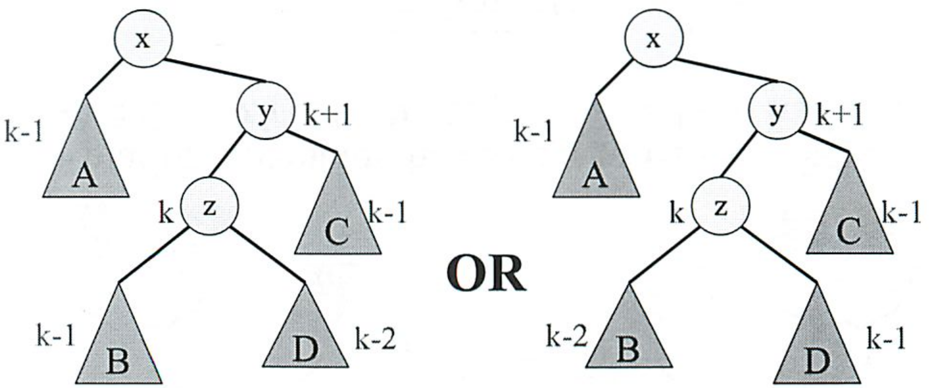
(Same as Case 1)

Case 3: y is left-heavy



Need to do more …

Case 3: y is left-heavy

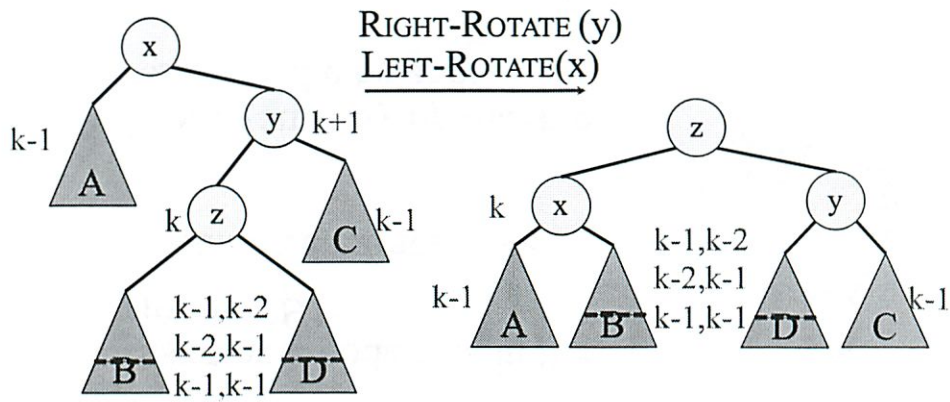


& z is left-heavy

& z is right-heavy

OR …

Case 3: y is left-heavy



And we are done!

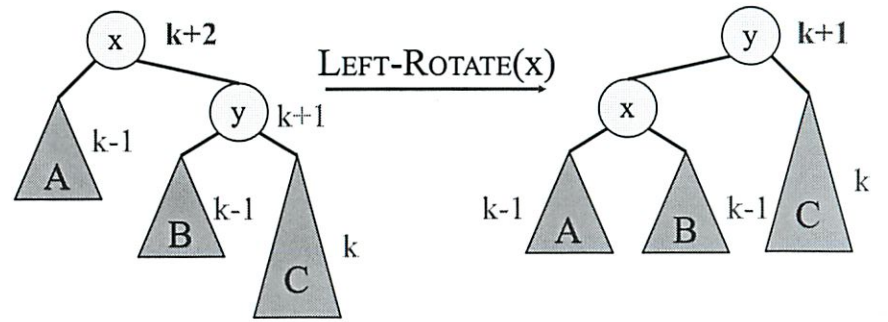
Complexity

Insertion:

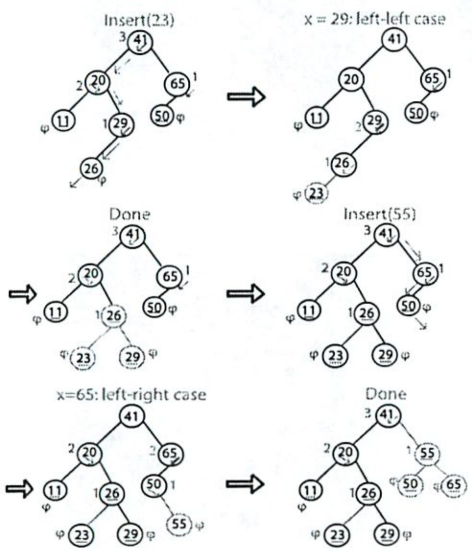
Local rebalancing:

How many local rebalancings after one insertion?

Recall Case 1: y is right-heavy



Examples of insert/balancing



Balanced Search Trees …

- AVL trees (Adelson-Velsii and Landis 1962)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Scapegoat trees (Galperin and Rivest 1993)
- Treaps (Seidel and Aragon 1996)
- ….

US capitol Mississippi Arkansas ?

Who invented the Multiplication Algorithm?

6.006 Recitation
Complexity

2/17

(back in the normal recitation)

No class Mon

Tue: Mon schedule – so no 6.006 lecture

Wed: Recitation will be OH

He posts stuff on Piazza

Feedback form at the end of recitation

OH 5–7
65 lounge
Tue (?)

Today: About runtime (dern did this in Wed's sub recitation)

Last time (missed): BST code
– but it will be posted on Piazza

---

1. Write a recurrence

a) Calc $f(n)$ – amt of time not doing recursive calls
ie. the amt of time spent on the top level

b) Let a be the # of recursive calls $\frac{n}{b}$
└ the size of input on those calls

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(f(n))$$

(2)

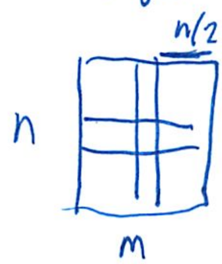
Usually you cut the input by a certain fraction

- Half → $\frac{n}{b} = \frac{n}{2}$
  - ^ everything we've done so far is half
  - trinary search is $\frac{n}{3}$
  - most things are $n/2$
- It can also be $n-1$
- This method works for $\frac{n}{b}$

The cross algorithm on squares

$T(n) = T(\frac{n}{2}) + \Theta(n)$

^ side lenght - one & look ↑



If did on a square basis (2D look)

$T(n,m) = T(n, \frac{m}{2}) + O(n)$

On ~~[illegible]~~ something else

$T(n) = T(\frac{n}{2}) + O(1)$

(3)

Merge Sort

Cut in half

Then combine the 2 halfs - in linear time

(I saw when reading the book)

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

↑ all the time spent doing <u>anything</u> that is not recursion

---

Master Theorm

Will just do - not prove

$$T(n) = \underbrace{a\,T\left(\frac{n}{b}\right)}_{\text{recursive runtime}} + \underbrace{O(f(n))}_{\text{Top level runtime}}$$

$n^{\lg_b a}$ $\quad$ $f(n)$

So for the 2D bar

= ← $n^{\lg_b a} = 1$

↙ # of leafs at final stage of recursion

(4)

The 2D square

~~~~

$n^{\lg_b a} = 1$

Merge sort

$n^{\lg_b a} = n$

cn leaves ~~~ at recursion

To get overall runtime

But first comparing two run times

Want algos. to run in polynomial time

$n^c \cdot (\lg n)^d$

Compare in order

1. Compare "c"s –
   - if "c"s are different, bigger c wins
   - we can say "significantly different" or "polynomially different"
~~~~

So for example

$$n(\lg n)^3 >> \sqrt{n}(\lg n)^3$$

↑ much greater than

2. Next compare d

- this time only "slightly" different.

$$n(\lg n)^3 < n(\lg n)^{10}$$

Back to Master Theorem

Case 1. if "significantly" different

- answer is the bigger one

Case 2. "slightly" different

- answer is bigger one $\cdot \log n$

(The answer to the recurrence $T(n)$

⑥

So

$$T(n) = T\left(\frac{n}{2}\right) + \theta(1)$$

$$\quad\quad 1 \quad\quad 1$$

Case 2 - not sig different (both equal)
So bigger one (both equal) (1) · log n = log n

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

$$\quad\quad 1 \quad\quad n$$

Case 1 - significantly different
So bigger one (n) = n

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$\quad\quad n \quad\quad n$$

Case 2 - not sig different
So bigger one (n) · log n = n log n

(So much more clear than other recitation)
└ (shows how much power a good teacher has!)

So why does this work?
It looks at the top and bottom step of the recursion

(7)

Another example: Dense matrix multiplication algorithm

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$n^{\log_2 7} = n^{2.78}$ $\quad n^2$

need to keep since asy

Case 1 – $2.78 > 2$

So run time

$$\Theta(n^{2.78})$$

One more

$$T(n) = 25T\left(\frac{n}{5}\right) + n^2/\log^3 n$$

n^2 $\qquad \frac{n^2}{\lg^3 n}$ ← dividing is like a Θd

Power of logs same

So slightly diff – case 2

$n^2 \log n$

(8)

If you have a programs code - want to find runtime

1. Look for loop

Runtime = time per iteration · # of iterations

~~Pulling Walk~~ Pulling items out of Binary search tree

- it took linear time to pull each one out

2. Change runtime to objects

in_band_traversal() = linear time

(Confused here)

A few things about Python

return peak_find(

array[n/2 : n] ... ← know peak is on right

← runtime here? linear - since Python copying

String slicing → O(1) Ok

array slicing → O(k) Not Ok

↑ size of slice

Instead - pass in original array + rebalance by passing around #s ?

(9)

ie (min, max) in parameters

---

array 1 + array 2 → linear time Not ok

array 1 += array 2 → linear time Not ok

└ instead array.append() → constant time Ok

array.extend(array 2) → linear 2 linear time Ok

string 1 + string 2 → linear time Not ok

└ depends on what doing

Usually better to turn into list of chars

└ linear time

Code example

```
def fib(n):
    if n in (0,1):
        return 1
    return fib(n-1) + fib(n-2)
```

Guess as close to run time w/o going over

longer than a min - so lots of leaves

- on the order of its output 1min 28 sec

Book Reading

2/18

(The lectures/class does not follow the book!)

Data Structure (Quick Review)

- Sets
  - can change over time → dynamic
  - dictionary - can insert, delete, test membership
  - other sets need other actions
  - has a key and satellite data
    - key ↑ what we search for
    - satellite data ↑ data that rides along
  - can do queries (read only) or modifying ops

Elementary Data Structures

Stacks + Queues
↑ LIFO ↑ FIFO

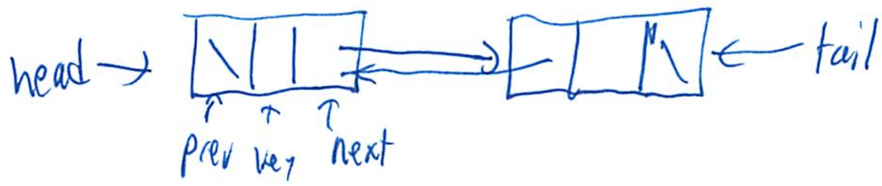
Underflow - trying to pop an empty stack

Queue has a head + tail

2

Linked List - data structure w/ objects in linear order

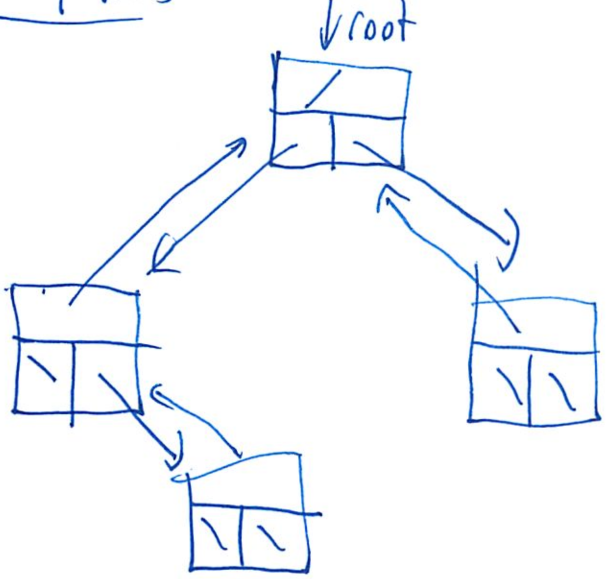
- doubly linked next and previous



- Singly linked - just next
- Sorted in order
- Circular ~~next~~

- insert splice into

Binary trees



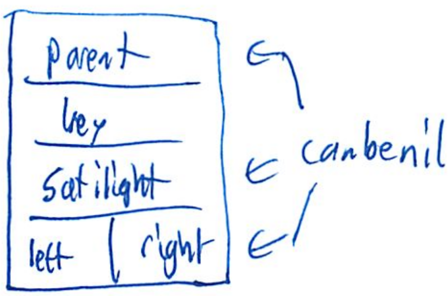
Storage

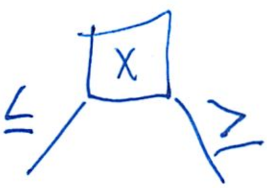
Items stored in memory like 6.004
(was there anything else relevant here?)

(3)

Binary Search Trees

- Supports many common ops
- Can use as both a dictionary and priority queue
- To do things in $\Theta(\lg n)$ instead of $\Theta(n)$





to print order → in order tree walk

(What I missed on Google interview)

if $x \neq$ Nil
  Inorder tree walk (x.left)
  print x.key
  I. T. W. (x.right)

(So simple!)
– (What did I do – can't remember) (Think recursively)

(4)

Time

n items so $\Omega(n)$

each row $T(k) + T(n-k-1) + d$

($T(k)$ ← left, $(n-k-1)$ ← right, d ← 1 op)

To show $T(n) = O(n)$ →

Show $T(n) \leq T(k) + T(n-k-1) + d$

↑

$O(n)$

⋮

$T(n) = (c+d)n + c$

~~So know $\Omega(n) = \Theta(n)$~~

So know

$\Omega(n)$ and $O(n)$ → so $\Theta(n)$

(4b)

root ↓ query

Searching (x, k)

```
if x == Nil or k = x.key
    return x

if k < x.key
    return tree Search (x.left, k)
else "   "   (x.right, k)
```

(so simple!)

(can also have more efficent iterative version

```
while x ≠ Nil and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

(also so simple!)

Min

just go all the way left!

Max

"   right!

(5)

Insertion (T, z)
↓ tree   ↓ new item

z.key = v
z.left = nil
z.right = nil

```
y = nil
x = T.root
while x ≠ Nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right

z.p = y
if y == Nil
    T.root = z   // If tree empty
else if z.key < y.key
    y.left = z
else y.right = z
```

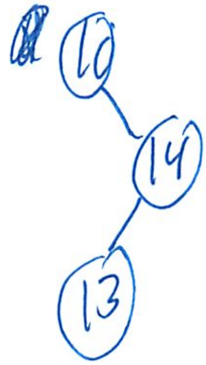
(I think I was confused b/w object and key of object)

So for fig 12.3 what if add 16?

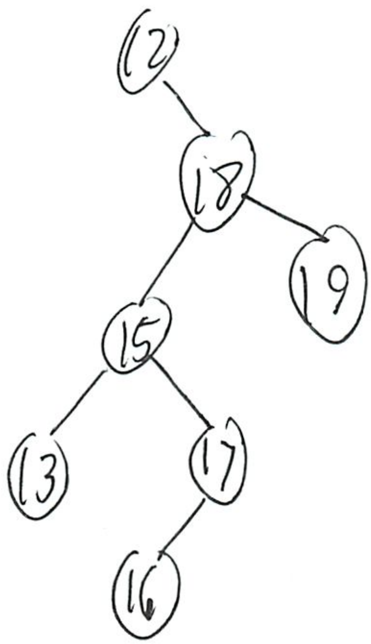
?!

(6)

According to WP example
this is possible



So how would walk see this?



~~iterative~~

On 12
left → nothing
print 12
~~the~~ right → on 18
left → on 15
left → on 13
print 13
return to 15
print 15
On 17
print 16
print 17
print 18
print 19

I guess it works!
not completly intuitive

(7)

Deletion

3 cases

- No children: just remove
- One child: replace w/ it
- Two children: Take the item on the right (y) to replace it
  - Original left (z.left) becomes new left (y.left)
  - y's original right tree remains
  - * wait - not always one on the right
    - y can lie within z's subtree but is not z's direct right child

(complex! - must have make sure to cover all cases!)

- Use Transplant subroutine

Augmented Data Structure

Store extra info in it

Like the # of sub nodes

Interval Tree search

Red-Black
- to ~~change~~ help make sure it is balanced

Reviewing L4

2/18

If not balanced $O(n)$

AVL Trees

- WP: more rigidly balanced than red-black trees
- key invariant left = right ± 1
- $n_h \geq 1 + n_{h-1} + n_{h-2}$
  ↑ min # of nodes

  $n_h > 2 n_{h/2}$

  $n_h > 2^{h/2}$

  $h < 2 \lg n_h$

  (I don't get the premise of this qu....)

- Rotation thing
  - maintains orders of keys

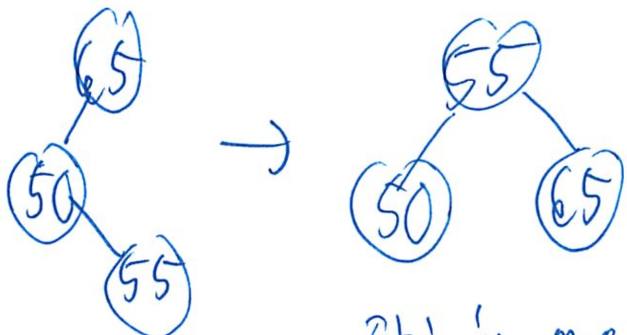
- Insert
  - put it on as normal
  - rebalance

(2)

Several cases for rebalancing...

No details on complexity in slides... is in my notes

Ahh balance is what I was writing about earlier as kinda weird



?this is more of what I traditionally thought as a BST

③

Complexity

Insertion: $\Theta(\lg n)$

Local rebalancing: $O(1)$

- constant

How many local rebalancings after 1 insertion

~~$O(n)$~~

~~(unconfirmed)~~ ~~(worst case need to rebalance from bottom to the top of tree~~

Recall Case 1 y is right heavy

Worst case $O(\lg n)$

Examples

- insertion + rebalancing
- (need to look at closely...)

(4)

People really liked AVL trees

- elegant
- solve a fundamental problem

Also other ways

- Red-Black trees
- Splay trees
- Scapegoat trees
- Treaps

Lots of copies of good ideas

- Lots of capitals have domes

Who invented the multiplication algorithm?

$$x \cdot y = x \cdot (y-1) + y$$

Then recurse down!

exponential slow down to add instead of multiply

Problem Set 1

This problem set is due **Wednesday, February 22** at **11:59PM**.

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice questions, your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to the correct solution which is described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **(15 points)** Order of Growth

   For each group of functions, sort the functions in increasing order of asymptotic (big-O) complexity. Partition each group into equivalence classes such that $f_i(n)$ and $f_j(n)$ are in the same class if and only if $f_i(n) = \Theta(f_j(n))$. (You do not need to show your work for this problem.)

   (a) **(5 points)** Group A:

$$f_1(n) = n \log n$$
$$f_2(n) = n + 100$$
$$f_3(n) = 10n$$
$$f_4(n) = 1.01^n$$
$$f_5(n) = \sqrt{n} \cdot (\log n)^3$$

   (b) **(5 points)** Group B:

$$f_1(n) = 2^n$$
$$f_2(n) = 2^{2n}$$
$$f_3(n) = 2^{n+1}$$
$$f_4(n) = 10^n$$

(c) **(5 points)** Group C:

$$f_1(n) = n^n$$
$$f_2(n) = n!$$
$$f_3(n) = 2^n$$
$$f_4(n) = 10^{10^{100}}$$

Solution Format:

Your answer to this problem should be a list of lists of integers. Each sublist should contain the indices of a set of functions which all have the same rate of growth. The order of the indices within the sublist does not matter. The sublists should be ordered from the slowest-growing functions to the fastest.

Example Question:

$$f_1(n) = n$$
$$f_2(n) = 2n^3$$
$$f_3(n) = n + 5$$
$$f_4(n) = n^2$$
$$f_5(n) = n^3$$

Example Answer:

```
# Note that 4 is in a list by itself
# Note that the order of 1 and 3 (and 5 and 2) does not matter
answer_for_example_for_problem_1 = [[1, 3], [4], [5, 2]]
```

2. **(10 points)** Recurrence Relations

   (a) **(5 points)** What is the asymptotic complexity of an algorithm with runtime given by the recurrence:

   $$T(n) = 4T(n/2) + \log n.$$

   1. $\Theta(n)$
   2. $\Theta(n \log n)$
   3. $\Theta(n^2)$
   4. $\Theta(n^2 \log n)$

   (b) **(5 points)** What is the asymptotic complexity of an algorithm with runtime given by the recurrence:

   $$T(n) = 9T(n/3) + n^2.$$

   1. $\Theta(n \log n)$
   2. $\Theta(n^2)$
   3. $\Theta(n^2 \log n)$
   4. $\Theta(n^3)$

Solution Format:

Your answer to this problem should be a single integer for each part. For example, if you thought the answer to part (a) was 5 and the answer to part (b) was 6, then your answer should look like:

```
answer_for_problem_2_part_a = 5
answer_for_problem_2_part_b = 6
```

3. **(20 points)** 2D Peak Finding

Consider the following approach for finding a peak in an $(n \times n)$ matrix:

1. Find a maximum element m in the middle column of the matrix.
   - If the the left neighbor of m is greater than it, discard the center column and the right half of the matrix.
   - Else, if the right neighbor of m is greater than it, discard the center column and the left half of the matrix.
   - Otherwise, stop and return m.
2. Find a maximum element m' in the middle row of the remaining matrix.
   - If the the upper neighbor of m' is greater than it, discard the center row and the bottom half of the matrix.
   - Else, if the lower neighbor of m' is greater than it, discard the center row and the top half of the matrix.
   - Otherwise, stop and return m'.
3. Go back to step 1.

(a) **(5 points)** Let the worst-case running time of this algorithm on an $(n \times n)$ matrix be $T(n)$. State a recurrence for $T(n)$. (You may assume that it takes constant time to discard parts of the matrix.)

(b) **(5 points)** Solve this recurrence to find the asymptotic behavior of $T(n)$.

(c) **(10 points)** Prove that this algorithm always finds a peak, or give a small $(n \leq 7)$ square counterexample on which it does not.

Solution Format:

Your answers to parts (a) and (b) should be Python strings. For example, you may write:

```
answer_for_problem_3_part_a = 'This is a Python string.'
answer_for_problem_3_part_b = '''
Here is a Python multiline string.
It starts and ends with three quotation marks.
'''
```

human graded

If your answer to part (c) is a proof of correctness, then return a string as above. If it is a counterexample matrix, then write the matrix as a list of lists of integers, **not as a string.**

4. **(30 points)** Programming Exercise: Peak In Circle

Write a function `find_peak_in_circle` that efficiently finds a peak value in a circle of integers. This function should take a list of integers as an input. Two elements in this list are *adjacent* if they are consecutive elements of the list or if they are the first and last element. A peak is an element of the list which is greater than or equal to both of its adjacent elements - your goal is to find the value of any peak.

You may assume that the input list is non-empty. However, you may not change the entries of the list, and your function should also accept (immutable) tuples. Here are some example test cases that your function should agree with:

```
# Both 4 and 5 are peaks in the array [1, 2, 5, 3, 4]
find_peak_in_circle([1, 2, 5, 3, 4]) in (4, 5)
# The element 3 is not a peak in [3, 2, 1, 4] because it is adjacent to 4
find_peak_in_circle([3, 2, 1, 4]) == 4
```

Solution Format:

You should answer this problem by filling in the body of the function `find_peak_in_circle` in the solution template.

Doing PS1

2/18

1. Categorize same

② n (=1	① $n^{1/2}$	③ $n \lg n$	④ k^n
f_2	$f5$	f_1	f_4
f_3			

duh ...

linear

$n^{1/2} < n^1$

n^k = polynomial

2^{n^k} = exponential

$\sqrt{n}\ (\log n)^3$

master theorm

$c = 1/2$

$d = 3$

So essentially $n^{1/2}$ for $n=100 \to 10$

2^n for $n=100$ ← much bigger

vs n^2

$n \lg n$ ← grows for $n=100 \to 200$

so $\lg n \to n \to n\lg n$

②

That took me a while

Important

Testing grading cprogram

⊗ ↳ need to do of them 1st

major bummer

$\lg n \rightarrow n^{1/2} \rightarrow n \rightarrow n \lg n \rightarrow n^2 \rightarrow 2^n$

for n=100	2	10	100	200	10000	huge!

B Rember to remove supervelus

$2^{2n} \neq 2^n$ since c diff

- but its not c!
- it changes tremendeously

2^{n+1} does not really change asy vs 2^n ?!

What does Θ mean exactly?

$\Theta(g(n))$ means $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

So 2^{n+1} is different?

since

③

How about 10^n vs 2^n - same?

again seem different

but here ~~ck~~ can add a c

f_1 f_3 f_2

f_4

---

c)

$10^{10^{100}}$ 2^n $n!$ n^n

↑

constant

4 3 2 1

---

2. Asy complexity of this

Master theorm

$a = 4$ $f(n) = \lg n$

$b = 2$

~~$n^{\lg_2 4} = \lg n$~~

~~∴ That didn't help...~~ don't think so

(4)

~~$4 \lg_2 4 \quad \lg n$~~

both $n^{\log_b a}$

So n^2 vs $\log n$

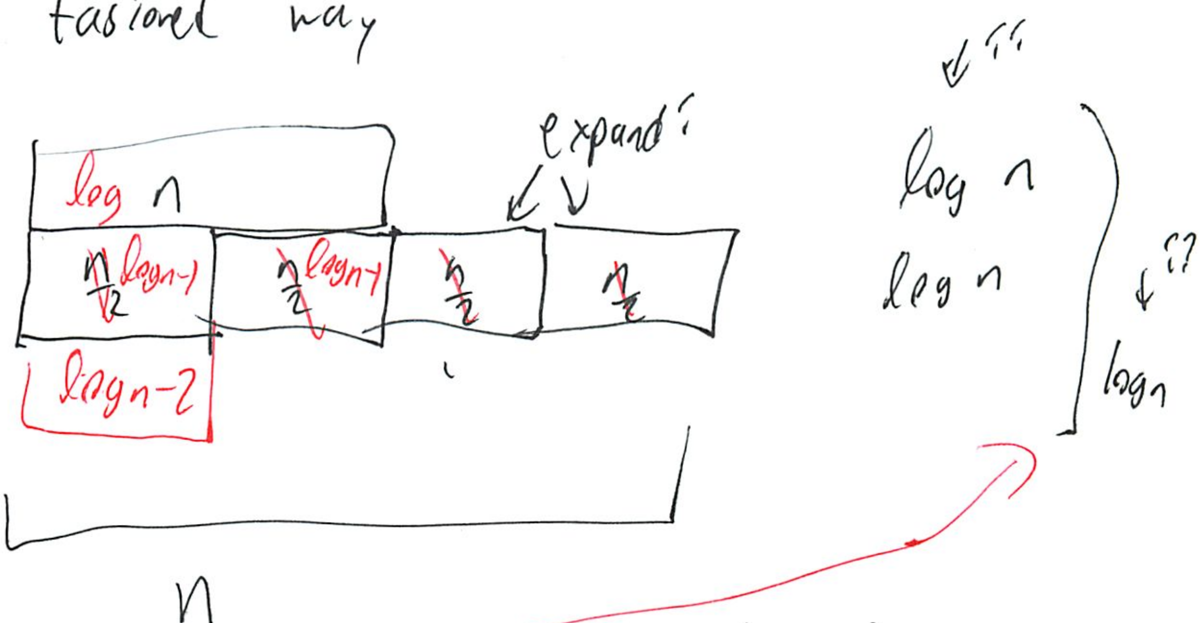
Just do!

Now the cases

C is 2 vs 0

So case 1 → $O(n^2)$ ✓

Confirm old fashioned way



$(\log n)^2$

?? Blocks nasty

levels

$\frac{n}{2} \to 1$

$\log_2$ levels

Size each level ~ 4

$4^{\log_2}$

~ = size of each level

(5)

Try anoter way

$T(n) = 4\ T\left(\frac{n}{2}\right) + n$ log n s - not n

$= 4\left(4\ T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$

$= 16\ T\left(\frac{n}{4}\right) + 2n + n$

$=$

$= 16\left(4\ T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 3n$

$= 64\ T\left(\frac{n}{8}\right) + 7n$

this is $4 \to 16 \to 64$ 4^k

total $7+3+1$ add $4+2+1$

Very wrong

but how many steps k

Go to T(1) which is base

where they get mis $\frac{n}{4^k} \approx 1$

$n = 4^k$

$k = \log_4 n$ ←

(c)

~~So $2^{\log_4 n} T\left(\frac{n}{2^{\log_4 n}}\right) + n \lg n$~~

No I don't really get this

Move on

---

b) $a = 9$
$b = 3$

$n^{\lg_3 9} = n^2$ vs n^2

Same
D is same

$n^2 \lg n$ ✓ by theorm

---

3. <u>2D Peak finding</u>

Write recurrence

$2T\left(\frac{n}{2}\right) + O(n)$

↑find max

or did we have something better?
~~[illegible]~~ not a peak - the max

⑦

But for the other D

- do this all twice?

$2\,2T(n/2)$

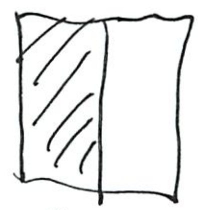
Also n vs m
- here the same
(I don't fully get this!)

So if $4T(\frac{n}{2})$ that is n^2 - from last problem

Expand it out

$T(\frac{n}{2}) + O(n)$

not doing both? recurse in half - find max - do rows next - same amt - of rows though!



$T(n/4) + \frac{n}{2}$

only scans half

now half

So ~~$T(n) + T(n)\,2T$~~

~~$T(n)$~~ $+ O(n)$

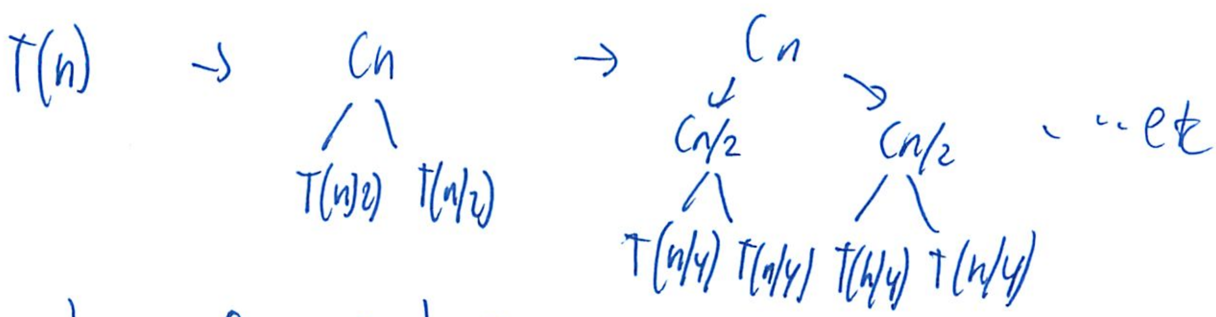
~~$T(n)$~~ $+ O(n)$

~~$T(\frac{n}{2})$~~ $+ O(n/2)$

$O(n/2)$

(8)

Book

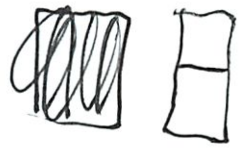


have lg n + 1 rows

So $T(n) = 2T(n/2) + cn$

n ~~rows~~ rows in a col 

|

n cols in a row 

|

n/2

|

n/2

no - I was looking at it wrong

---

n rows in a col 

n/2 cols in a row 

n/4 rows in a col 

So $T(n) = T(\frac{n}{2}) + 1$

↑ ignore

Or O(n) to look at

no that is here

(a)

No remember its a recurrence

So we are adding stuff

$+ O(n)$

Basically

$O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + \ldots$

thats $\ln n$??

Only deal 1D

In recitation

$T(n) = T(\frac{n}{2}) + O(n)$

$T(n,m) = T(n, \frac{m}{2}) + O(n)$

So this is $O(n) + O(\frac{n}{2}) + \ldots$

↑since fill in

master theorem

$a = 1$

$b = 2$

$n^{\lg_2 1} = n^0 = 1$

So 1 vs n

↑bigger

So $O(n)$

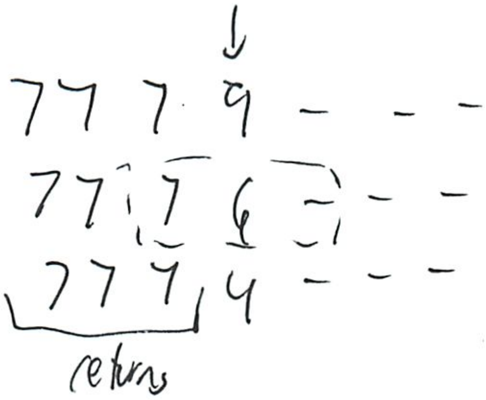
b) Just did → n

(10)

c) Prove it correct

I think it is

Try it



is 7 a peak?
Yeah - not smaller

Think correct

So test file runs

Does not tell you if correct

- but said score "100"
- even though 4 not done...

4. Coding problem Find Peak in a Circle

- input : list of integers
- start + end adjacent
- find any peak
- accept tuples

↑ So like a list

In class

- split in half
- But can we do that here?

↑ which would break!

well cutting next to it

Can check it

O(2) is still $\log_2 n$

do the (start, end) trick

(12)

What do I care here?

include the big one in the cut

This is like looking for a breakthrough

Something very clever

Test in Eclipse on ~~base~~ simple case....

Oh ran into end issue

if n-1 < 0 - make it array len....

Think carefully!

Wrapping actually easy

(Any better way to solve - then just start test cases...)

Be very careful $n \pm 1$

Think I got it...

(X) 3 2 1 4

(13)

Ahh what happens to array when it wraps around?

Remove the last item and append?

3 (2) 1 4

← return

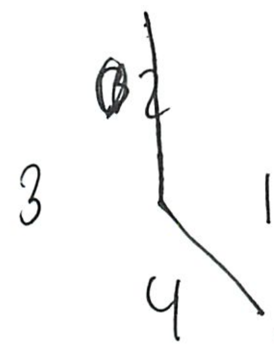
return 4 3

or need more??

4 3

here the start/stop is annoying....

Try circle



← So where we split this was my earlier idea

3 4

(14)

how about as simple as copy array)

↳ tail array head

✓ Passed a bunch more!

(I should prob be thinking more robustly ...)

Ohh gcc did n wrong

Still a bad case

22 23 24 25 18 19 20 21

Here where its clear my method does not work

I should re append ? each time this happens?

This is not very elegant...

; Move 1 each time

- need to fix start + stop

Ohh pop!

(Focus 1st on writing - then optimizing!)

Nice!

(15)

Now crashes on a diff test case.... :(

Special case lenght 1

- That should be it
- len 2
- len 3

len 2 - find max

len 3 - it will work as normal

could consolidate 2 and 1 special case...

(✓) Passes all small test cases

(✗) Fails large cases

- too much recursion

rewrite w/o recursion

use while loop....

Actually fairly easy

(–) So it doesn't error - but runs for 6.08 sec

max 5 sec

(16)

What am I doing wrong?

len(array)

⊗ - did not help...

Read Piazza

Nothing interesting

Pop → is O(1) for last el

but O(N) for other

↑ better way for 1st el?

Use a deque

↑ double ended queue

✓ Now all tests run

But take > 500 us

But 3.9 sec total

How to make it even faster

I'm making it slightly faster each time...

Need .0005 sec

Currently .5 sec...

I have no clue how to make it faster...

Deque can rotate ... ← helps somewhat...

(17)

Yeah I have no idea...

Oh can use Py profiler...

But how to read it??

I got to have the wrong core idea

Talk to TA or Shri

Hint: don't copy into deque
instead use indicies

2/19

Also before I got the hint I was thinking
shift by > 1

- Though prob small improvement

How to do other indicies?

Say 1 → 10

Want 8 → 2

8 9 10 1 2

or 2 → 8 } save!

8 9 10 1 2

2 3 4 5 6 7 8

(18)

So 2 cases — normal
— wrap

Could also do a virtual remapping system

So position 12 above is position ~~10~~ 2

12 % 10 = 2 ✓

-2 % 10 = 8 ✓

Lets do that!

So at 20 = len

get 18 19
'last' index

20 % 20 → 0

So comparing 19 and 0
21 22

return start = 19
stop = ← need to increment?

✓ Nick looks good

(19)

even longer

- remove sep functions

Emailed in

2/20

TA: have some ∞ loops
and off by 1 errors

So be very careful here...

Ah when it goes ↓ it never finishes...

27 11

When it uses first...

it's 0 1 1

So only add +1 to the other?

$$n+1$$

now $n-1 > n$

$n < n+1$

So back to n

$n-1 > n$

$n+1 < n+2$?

I'm guessing → ask *why*

(20)

So

22 ... 27 8 21

It keeps growing

But by default n is start + $\frac{stop - start}{2}$

0 4 19

4 14 19

14 16 19

16 17 19

17 18 19

18 18 19 ← stuck here

So if start = n, n++ ?

The reduction one 27 ... 11

0 8 16

0 3 7

0 1 2

0 0 0

It could be that the ns from the slide are not right here

(21)

Try if start = n → n++

Does not work on "first"

Only do on second.

So if 18 == 18

Then intead of 18 18 19
                ↑n
18 19 19
      ↑n+1

What ~~are~~ do we want ~~we doing~~ = comparing 19 to 0
↑n+1 ↑n+2

What were we doing 18 to 19
↑n ↑n+1

But then what do we do

Start = n+1
19

Stop is one more 19+1 = 20 ie 0

20 % 20 = 0

Then should compare 19 and 20

Why is n+1 still 18!

Ok seems to work!

(22)

ⓧ Limit exceeded!

So test <u>all</u> small cases

1st one fails

1 2 5 3 4

0 2 4
2 3 4
2 2 2

Ah in this case it narrowed down

So 2 5 3

7 <
x x

Should pass

I'm saying if n = start
it should be n = lenght − 1

✓ Passes now

(3)

Now [1 2 2 3 4] fails

0 2 4
2 3 4
3 3 4
↑ it should increment here
length = 2

Or pull shim out to global
✓ Passes

[3 2 1 4]

✓ Passes

[14 .... 27 8 .. 13] now fails

0 4 18
4 14 18
9 11 13
11 12 13
12 12 13 ← again having problem!
if stop - start = 1 and start = n

6/24

I'm still kinda guessing...

(✓) That passes all simple cases

(✓) Now passes all tests
— but is still too slow in a lot of cases

What else can I do?

Take larger steps
— but I may miss

22 ... 278 ... 21

Can take large jumps

But then it would do middle of that

And might not be right

How large jumps? 5, 10?

It should be algorithmic — $\frac{n}{2}$ or something

Emailed in

TA: Use a binary search when you wrap around
- right now am linear
? I know...

Binary search is that same split search...

Unless something more drastic like rotate array 90° and then look

Or 180°

Emailed in - but I could try that...

That seems to work - try it!

Passed all the small tests
and some large
But rotated a lot on one of tem... The large tests

1/4 didn't work either

Seems to be right when stop - start = 1

? Rotate again later? length / rotate ?

(26)

I wish I knew test case 3

$956679 - x + 833251 \% 1{,}000{,}000 = 1833250$

$|956679 - x| \% 1{,}000{,}000 = 1{,}000{,}499$

$956679 - x = 499$

957178 almost at end ...

but it should find ...

I'm not being strategic!

Needed also to reset if start = stop

Wrong rotate strategy?

Move random amt each time

✓ Random finds it

Try that

✓ All pass - too long

Oh have old code in here ... cut

2/20

1st check 1st + last Shri help
Normal ~~elka~~ peak finding

x No

Read TA Note! ~~Pla~~ Remember larger → go there

'1 9 . . . ~~2~~3 . . . 27 ~~8~~ 9 10 13

here ↑ 23

look here

look left and right

↑ 27 bigger

So b/w 23 + 27

or

8 9 10

↑

23 was larger

Than 9

go left - not right

(Laptop is so slow)

~~el~~ Initalize in the middle

②

[0 1 2 2 3 4]

n-1 ↑ n+1

↑ new largest

– then continue that way

else

Will this work?

↳ TA said is like hill climbing

Apply modulus to all

[12 11 27 26 ...]

↑ largest ↑ n=3

n= 2

0 – 2

n=1

but 12 < 27

→ look right

(3)

Now synonym on the other side
–need a test case...

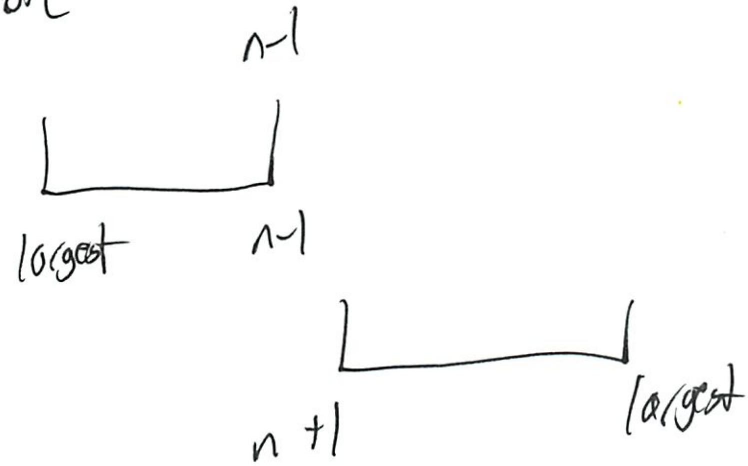
largest = 8

10 11 12

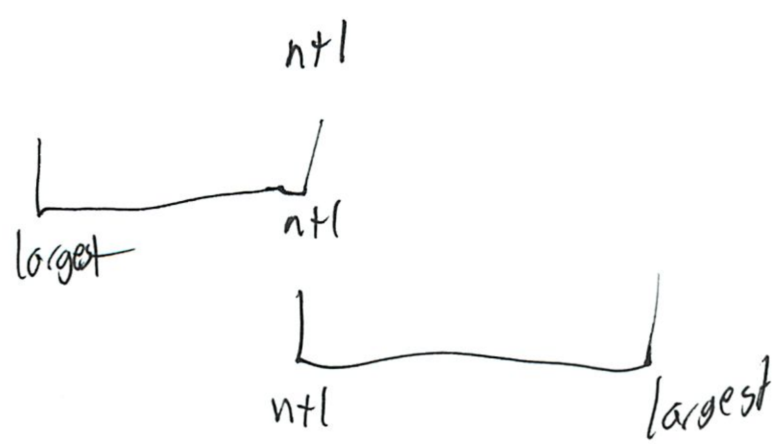
↑ ↑

look larges

Before



Here



(!) Passed all but one

(9)

Now need to patch up

It goes to end + hill climbs the rest of the way

So what is going on

Answer 1239903
↑
in position 493761

So Im guessing it did

~~0~~0 493761 - - - ~~0~~

~~3500~~

1239900 27000

35000 ~~400000~~ 239900

How to get it

Look at the ones that work - see why

is at 0 999999 999999 at one pts

finds new largest

⑤

Is at 0 0 1

n-1 = 999999
↑ w/ modulos

0 -1 -1
.99998 - new largest

0 -1 -1
on the right

0 99999 99998
↑ n+1 as start ↑ largest

I think it was that non mirrored thing
second largest left just n-1

Try it

Works

⊗ Limit exceeded

The 2 test cases I know of work...

Change to n

My 2 test cases still work

⊗ Limit exceeded

(6)

It fails on my fav test case

22 ... 22 8 - - 21

7 6 5

On the right

(we should never be here!)

4 6 9 positions

largest = 5

n = 5

largest = 5

So try if largest = n

return answer

(is that valid?)

(X) 4/5 small

4/5 big - now a diff test case

(7)

0 4 9

new largest 5

4 6 9

n+1 = 5 so said is largest

obviously not

Only if start - end ?

So print n-1 instead

Try

(✓) Fixed the small test

4/5 large tests

Wrong is largest ...

0 -1 -2

Guess random offsets ... Not

(✓) Fixed - but did it break others?

Not very scientific ...

(✗) Limit exceeded

(8)

Broke my earlier pset

Same 7 6 5 issue

But before did if largest == 1

~~8~~ 4 6 9

↳ largest = 5

5 is the correct ans

But do one more round

But why we go on the right?

✓ So works now...

But now are hill climbs...

— Same ~~4 5 6~~ 7 6 5 error...

I should think more robustly

— not just gt ✓

(9)

10 9 8

test this
left

or is largest → step

n-1
move to

was n+1

then test n-1

Now 0 1 2

largest = 2

first → on the right

If set to n

Now 1 1 2

largest = 2

first → on the right

✓ Passes all small cases

(10)

So

10 9 8

left

↑ is largest

right

then reverse

8 9 10

right

↑ is larges

left

(✓) Passes all small cases

(✓) Passes both troublesome (1+3) large cases

~~X~~ (X) But the wrong output

— that same 27 error

Go to return n in both cases

(X) All correct — but 1 too slow

Was here before

⑪

So always second on the right new largest

n+2
–still slow

So setting stop + start the same!

Before it crosses

first ~~Stop~~ start n+1 % ~~largest~~ lenght
stop n-1 % ~~largest~~ lenght

So for that

goes around

So what to do here

start = n+1
Stop = unchanged...

✓ Very Fast

¡Now all works!

✓ 10/10 !!!

(12)

Some weird thing

on first ie (10 9 8)

largest right start = n+1 % len

stop = n-1 % len

So 10 9 8 (arrow from 8 wrapping back to 10)

On second (ie 8 9 10)

largest right ~~stop = n~~

start = n+1

stop = unchanged!!!

(13)

4 5 6 largest = ~~9~~ 11

10 9 8

↑n 1

Start at 6 % length

end at 4 length

pos 6 7 8 9 10 1 2 3 4

6 7 8 11 12 1 2 3 4

↑ → ↑ largest

9 10 1 2 3 4

↑

11 12 1 2 3 4

↑

largest = 12

=

return 12

OH

2/21

$\sqrt{n} \cdot (\log n)^3$

↑ faster growing than $\sqrt{n}$ ← slow runtime

$2^{2n} \neq 2^n$

2^{n+1}

Systematic way

Divide one by the other

0 ← $\neq \theta$

Constant ← are θ

∞ ← $\neq \theta$

$\frac{2^{2n}}{2^n}$ Subtraction of exponent

$2n - n = 2^n$ ⊗ constant

(2)

$$\frac{2^{2n+1}}{2^{2n}} \qquad 2n+1 - 2n = 1$$

2^1 = constant ✓ same

2^n = exponential

n^2 = polynomial

Rest on the P-Set note sheets
Had everything else right...

$$\frac{2^n}{10^n} = 5^{-n}$$ ⊗ Not same

Problem Set 1 Solutions

1. **(15 points)** Order of Growth

   For each group of functions, sort the functions in increasing order of asymptotic (big-O) complexity. Partition each group into equivalence classes such that $f_i(n)$ and $f_j(n)$ are in the same class if and only if $f_i(n) = \Theta(f_j(n))$. (You do not need to show your work for this problem.)

   (a) **(5 points)** Group A:

$$
\begin{aligned}
f_1(n) &= n \log n \\
f_2(n) &= n + 100 \\
f_3(n) &= 10n \\
f_4(n) &= 1.01^n \\
f_5(n) &= \sqrt{n} \cdot (\log n)^3
\end{aligned}
$$

   Solution:

   `[[5],[2,3],[1],[4]]`

   We observe that f_5 has the smallest order of growth, since it grows sublinearly. (Recall that $\log n = o(n^\epsilon)$ for any $\epsilon > 0$.) Next, $f_2 = \Theta(f_3)$, since additive and multiplicative constants do not affect asymptotic growth. We know that $n \log n = \omega(n)$, and finally the largest growth is 1.01^n, which grows exponentially.

   (b) **(5 points)** Group B:

$$
\begin{aligned}
f_1(n) &= 2^n \\
f_2(n) &= 2^{2n} \\
f_3(n) &= 2^{n+1} \\
f_4(n) &= 10^n
\end{aligned}
$$

   Solution:

   `[[1,3],[2],[4]]`

   We observe that 2^n and 2^{n+1} are in the same equivalence class, since they differ only by a factor of two (and constant multiples do not matter.) Next, we observe that $f_2 = \omega(f_1)$, since

$$\lim_{n \to \infty} \frac{2^{2n}}{2^n} = \lim_{n \to \infty} 2^n = \infty.$$

Finally, $10^n = \omega(2^{2n})$, since

$$\lim_{n\to\infty} \frac{10^n}{2^{2n}} = \lim_{n\to\infty} \frac{2^n \cdot 5^n}{2^n \cdot 2^n} = \lim_{n\to\infty} \left(\frac{5}{2}\right)^n = \infty.$$

(c) **(5 points)** Group C:

$$\begin{aligned} f_1(n) &= n^n \\ f_2(n) &= n! \\ f_3(n) &= 2^n \\ f_4(n) &= 10^{10^{100}} \end{aligned}$$

Solution:

`[[4],[3],[2],[1]]`

The smallest order of growth is clearly f_4, since constant functions are $\Theta(1)$. Next, we observe that $2^n = o(n!)$, since

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{1}{2} \cdot \frac{2}{2} \cdot \frac{3}{2} \cdots \frac{n}{2} \geq \lim_{n\to\infty} \frac{1}{2} \cdot \left(\frac{3}{2}\right)^{n-2} = \infty.$$

Finally, the fact that $n! = o(n^n)$ follows from Stirling's approximation that $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, and noting that

$$\frac{n^n}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = \frac{1}{\sqrt{2\pi n}e^{-n}} = \frac{e^n}{\sqrt{2\pi n}}$$

which goes to ∞ as $n \to \infty$. We can also prove that $n! = o(n^n)$ directly by expanding $n!$ and n^n to notice that $n^n/n! \geq \frac{n}{2}$, which goes to ∞ as $n \to \infty$.

2. **(10 points)** Recurrence Relations

(a) **(5 points)** What is the asymptotic complexity of an algorithm with runtime given by the recurrence:

$$T(n) = 4T(n/2) + \log n.$$

1. $\Theta(n)$
2. $\Theta(n \log n)$
3. $\Theta(n^2)$
4. $\Theta(n^2 \log n)$

Solution:

3

The easiest way to see this is by the master theorem, since $n^{\log_2 4} = n^2$, and $\log n = o(n^{2-\epsilon})$. We can also solve this problem by expanding the recurrence to obtain a sum, and noting that the largest term in the sum dictates the asymptotic behavior. (Each subsequent term is less than half of the previous term, and therefore the entire sum is bounded by a constant multiple of the first term.)

(b) **(5 points)** What is the asymptotic complexity of an algorithm with runtime given by the recurrence:

$$T(n) = 9T(n/3) + n^2.$$

1. $\Theta(n \log n)$
2. $\Theta(n^2)$
3. $\Theta(n^2 \log n)$
4. $\Theta(n^3)$

Solution:

3

This follows by the master theorem, since $n^{\log_3 9} = n^2$. Since the additive term has the same asymptotic growth as $n^{\log_3 9}$, we gain an extra log factor.

3. **(20 points)** 2D Peak Finding

Consider the following approach for finding a peak in an $(n \times n)$ matrix:

1. Find a maximum element m in the middle column of the matrix.
   - If the the left neighbor of m is greater than it, discard the center column and the right half of the matrix.
   - Else, if the right neighbor of m is greater than it, discard the center column and the left half of the matrix.
   - Otherwise, stop and return m.
2. Find a maximum element m' in the middle row of the remaining matrix.
   - If the the upper neighbor of m' is greater than it, discard the center row and the bottom half of the matrix.
   - Else, if the lower neighbor of m' is greater than it, discard the center row and the top half of the matrix.
   - Otherwise, stop and return m'.
3. Go back to step 1.

(a) **(5 points)** Let the worst-case running time of this algorithm on an $(n \times n)$ matrix be $T(n)$. State a recurrence for $T(n)$. (You may assume that it takes constant time to discard parts of the matrix.)

Solution:

```
T(n) = T(n/2) + \Theta(n)
```

In each iteration, we reduce an $n \times n$ matrix to a $n/2 \times n/2$ submatrix. Doing so requires $\Theta(n)$ work, since we must check all elements in the appropriate row/column.

(b) **(5 points)** Solve this recurrence to find the asymptotic behavior of $T(n)$.

```
T(n) = \Theta(n)
```

This follows by expanding the recurrence. We can bound the $\Theta(n)$ terms above or below by cn, and then bound $T(n)$ by

$$cn + \frac{cn}{4} + \frac{cn}{8} + \frac{cn}{16} + \cdots \leq 2cn.$$

(c) **(10 points)** Prove that this algorithm always finds a peak, or give a small ($n \leq 7$) counterexample on which it does not.

Solution: This algorithm does not always find a peak. One possible counterexample matrix is given below.

```
[[0,0,0,3,2,1,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,7,0,0],
 [0,0,0,4,6,0,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0]]
```

When we run the algorithm, it will return 2 in location $(0, 4)$, which is not a peak. The algorithm will first find that 4 is the maximum element in the middle column, and therefore recurse on the right half of the matrix (since 4 is adjacent to 6). It will then find that 6 is the maximum element in right half of the middle row, and will therefore recurse on the upper right quadrant (since 7 is adjacent to 6). It will then find that 1 is the maximum element in the middle column of the 3×3 upper-right submatrix, and will therefore discard all but the top three elements in the column beginning with "2." Finally, it will return 2 in location $(0, 4)$ as the answer, since 2 is above the 0 in location $(1, 4)$ and is greater than it.

4. **(30 points)** Programming Exercise: Peak In Circle

Write a function `find_peak_in_circle` that efficiently finds a peak value in a circle of integers. This function should take a list of integers as an input. Two elements in this list are *adjacent* if they are consecutive elements of the list or if they are the first and last element. A peak is an element of the list which is greater than both of its adjacent elements - your goal is to find the value of any peak.

You may assume that the input list is non-empty. However, you may not change the entries of the list, and your function should also accept (immutable) tuples. Here are some example test cases that your function should agree with:

```
# Both 4 and 5 are peaks in the array [1, 2, 5, 3, 4]
find_peak_in_circle([1, 2, 5, 3, 4]) in (4, 5)
# The element 3 is not a peak in [3, 2, 1, 4] because it is adjacent to 4
find_peak_in_circle([3, 2, 1, 4]) == 4
```

Solution: An example algorithm is below. Notice that in this algorithm we keep track of the bounds into the array (instead of copying the array with each recursion) and we recurse on the side containing the maximum element we have seen thus far. This algorithm has worst-case running time $\Theta(\log n)$.

```
def find_peak_in_circle(input):
  max_location = 0
  if input[-1] > input[0]:
    max_location = len(input) - 1

  min_bound = 0
  max_bound = len(input)
  while min_bound < max_bound - 1:
    mid = (min_bound + max_bound) / 2
    if input[mid] < input[max_location]:
      if mid < max_location:
        min_bound = mid + 1
      else:
        max_bound = mid
    elif mid + 1 < max_bound and input[mid + 1] > input[mid]:
      max_location = mid + 1
      min_bound = mid + 1
    elif input[mid - 1] > input[mid]:
      max_location = mid - 1
      max_bound = mid
    else:
      return input[mid]

  return input[max_location]
```

6.006 Recitation

AVL Trees

(1 min tato)

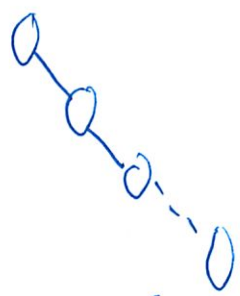
$$n! \sim \Theta\left(\sqrt{n}\ \frac{n^n}{e^n}\right)$$

n^n $\quad f(n)$ $\quad g(n)$

$$\frac{f(n)}{g(n)} = \begin{cases} 0 & \neq \Theta \\ \text{constant} & \rightarrow \Theta \\ \infty & \neq \Theta \end{cases}$$

AVL Trees

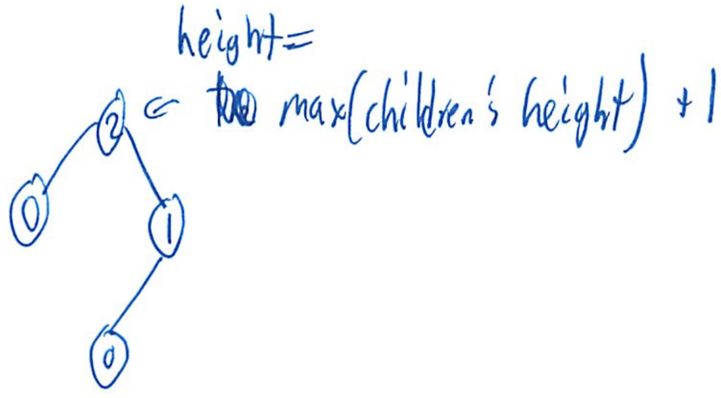
- Code to insert into BST very prestrictive



- AVL runs after insertion/deletion to balance tree
- very precise def'n of "balanced"
- then tree ops take $O(\log n)$

②

Heights



AVL's – height of two children of each node are within 1 of each other

Core goal / invarient

if child is None height is −1

Other option Scapegoat tree

- relatively balanced
- except ~~at~~ a section that is itself balanced
- can talk about amoritized / long-term avg
  - will talk about later

is log n in the worst case

(3)

Proof: If tree satisfies invarient, its branches are balanced

$f(h)$ = min # of nodes in an AVL tree w/ height h

Will show $f(h)$ big - need lots of node (missed details)

$g(n)$ = max height in an AVL tree of n nodes

$f(h)$ and $g(n)$ are inverses in the sense that
$f(g(n)) \leq n$

For example
$f(3) = 8$
$f(4) = 13$ ← he's not sure exactly

Say $f(h) = 2^h$ for perfectly (complete) balanced tree

If $n = \log 2^n$
height $\leq k$ → height $\leq \log_2 n$

(argument about # nodes I kina missed)

(4)

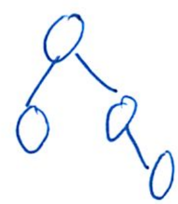
Natural thing to show: g(n) is small
But hard to do

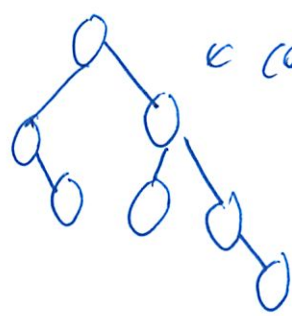
Based on h → get lower bound on # nodes

Claim f(h) is almost fibinichi #s

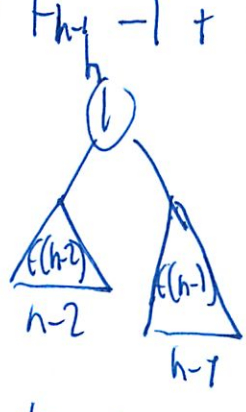
$f(0) = 1$ 

$f(1) = 2$ 

$f(2) = 4$ 

$f(3) = 7$  ← combine $f(1) + f(2) + 1$ nodes

$$f(h) = f(h-1) + f(h-2) + 1$$
$$= F_{h-1} - 1 + F_{h-2} - 1 + 1$$



$$= F_h - 1$$

Can prove w/ induction

(5)

Fib #s grow $\sim c(1.618\ldots)^h$

Suppose have

n node AVL tree

Suppose that $F_h - 1 \le n \le F_{h+1} - 1$

$h \le \log_{1.6} n \le h+1$

Any AVL tree of height $h+1$ has $F_{h+1} - 1 \ge n$ nodes

↳ This tree has $h \le \log_{1.6} n$ height

↳ so height $= O(\lg n)$

Proof is a bit backwards

h = height $= \lg n$

n = # of nodes

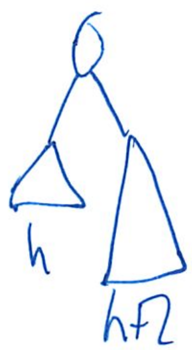
How to make AVL tree works

A diff way → fixing on search

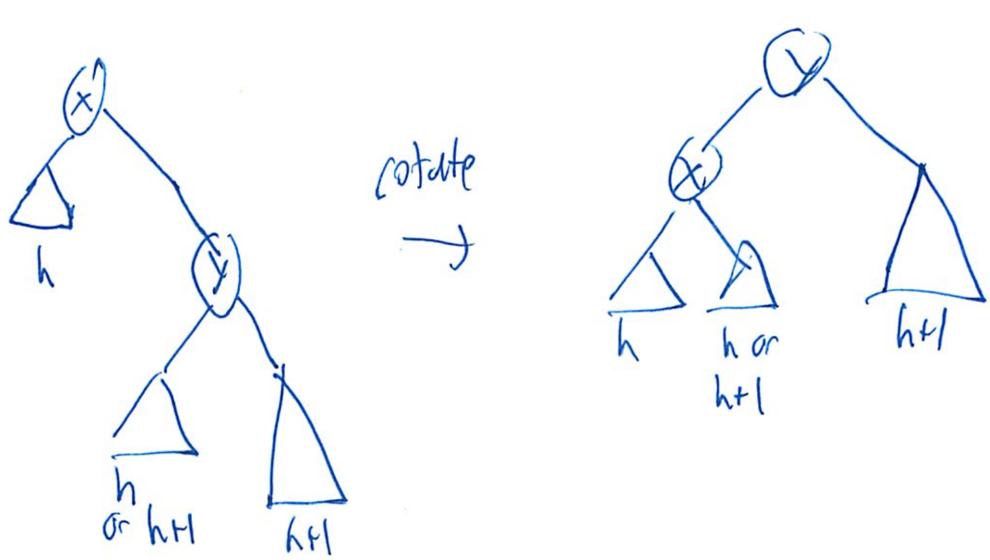
- way in class easy to understand, hard to write/implement
- This way hard easy

On search, at each node, check the heights of two children

- never more than 2 apart

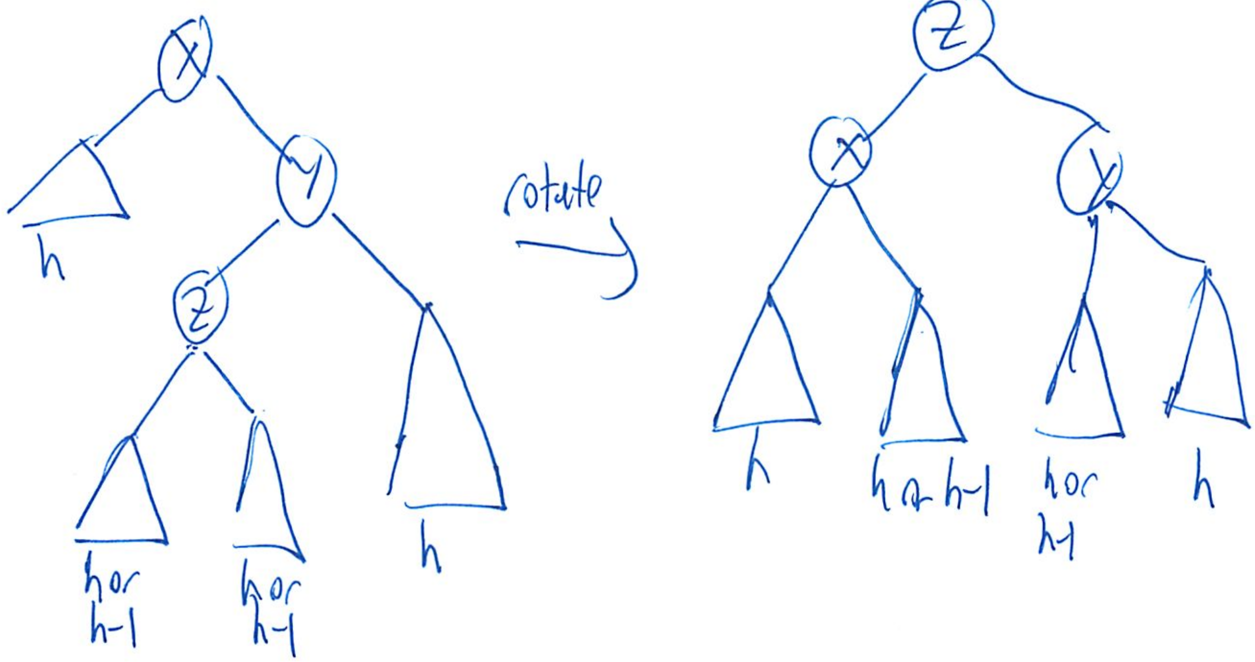


Divide into each possible case



(7)

2nd case



In AVL tree, all the lines drawn are pointers

In rotation, all relevant pointers are changed

- point to something else
- reassign a bunch of pointers

Other trees more complex

- esp their invariants

(8)

Replace b, cs w/ as

$\llcorner O(n^2)$

n grades to replace

each takes n times to copy string

AVL Trees

Recall the operations (e.g. `find`, `insert`, `delete`) of a binary search tree. The runtime of these operations were all $O(h)$ where h represents the height of the tree, defined as the length of the longest branch. In the worst case, all the nodes of a tree could be on the same branch. In this case, $h = n$, so the runtime of these binary search tree operations are $O(n)$. However, we can maintain a much better upper bound on the height of the tree if we make efforts to balance the tree and even out the length of all branches. AVL trees are binary search trees that balances itself every time an element is inserted or deleted. **Each node of an AVL tree has the property that the heights of the sub-tree rooted at its children differ by at most one.**

Upper Bound of AVL Tree Height

We can show that an AVL tree with n nodes has $O(\log n)$ height. Let N_h represent the minimum number of nodes that can form an AVL tree of height h.

If we know N_{h-1} and N_{h-2}, we can determine N_h. Since this N_h-noded tree must have a height h, the root must have a child that has height $h-1$. To minimize the total number of nodes in this tree, we would have this sub-tree contain N_{h-1} nodes. By the property of an AVL tree, if one child has height $h-1$, the minimum height of the other child is $h-2$. By creating a tree with a root whose left sub-tree has N_{h-1} nodes and whose right sub-tree has N_{h-2} nodes, we have constructed the AVL tree of height h with the least nodes possible. This AVL tree has a total of $N_{h-1}+N_{h-2}+1$ nodes (N_{h-1} and N_{h-2} coming from the sub-trees at the children of the root, the 1 coming from the root itself).

The base cases are $N_1 = 1$ and $N_2 = 2$. From here, we can iteratively construct N_h by using the fact that $N_h = N_{h-1} + N_{h-2} + 1$ that we figured out above.

Using this formula, we can then reduce as such:

$$N_h = N_{h-1} + N_{h-2} + 1 \quad (1)$$

$$N_{h-1} = N_{h-2} + N_{h-3} + 1 \quad (2)$$

$$N_h = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1 \quad (3)$$

$$N_h > 2N_{h-2} \quad (4)$$

$$N_h > 2^{\frac{h}{2}} \quad (5)$$

$$\log N_h > \log 2^{\frac{h}{2}} \quad (6)$$

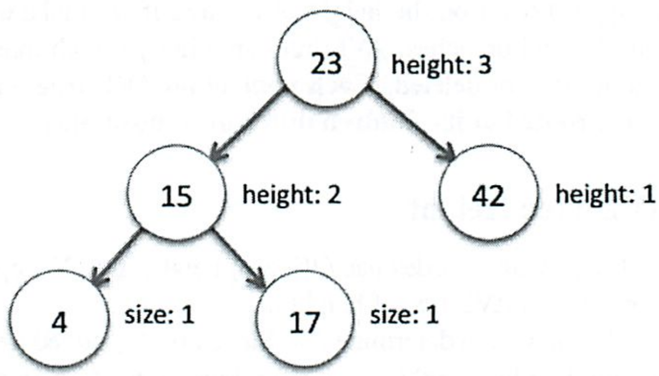
$$2 \log N_h > h \quad (7)$$

$$h = O(\log N_h) \quad (8)$$

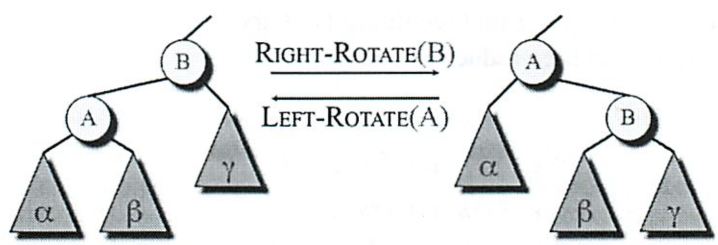
Showing that the height of an AVL tree is indeed $O(\log n)$.

AVL Rotation

We've shown that if we can indeed keep the tree balanced, we can keep the height of the tree at $O(\log n)$, which speeds up the worst case runtime of the tree operations. The next step is to show how to keep the tree balanced as we insert and delete nodes from the tree.



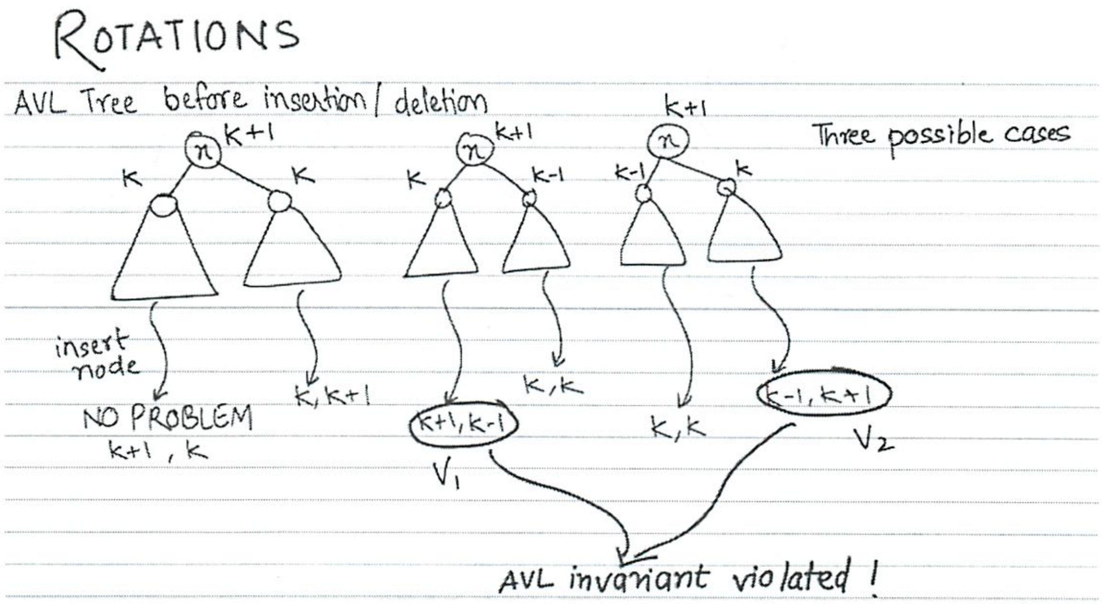
Since we need to maintain the property that the height of the children must not differ by more than 1 for every node, it would be useful if we could access a node's height without needing to examine the entire length of the branch that it's on. Recall that for a binary search tree, each node contained a `key`, `left`, `right`, and a `parent`. AVL trees will also contain an additional parameter, `height` to help us keep track of balance.



There are two operations needed to help balance an AVL tree: a left rotation and a right rotation. Rotations simply re-arrange the nodes of a tree to shift around the heights while maintaining the order of its elements. Making a rotation requires re-assigning `left`, `right`, and `parent` of a few nodes, but nothing more than that. Rotations are $O(1)$ time operations.

AVL Insertion

Now delegating to recitation notes from fall of 2009:

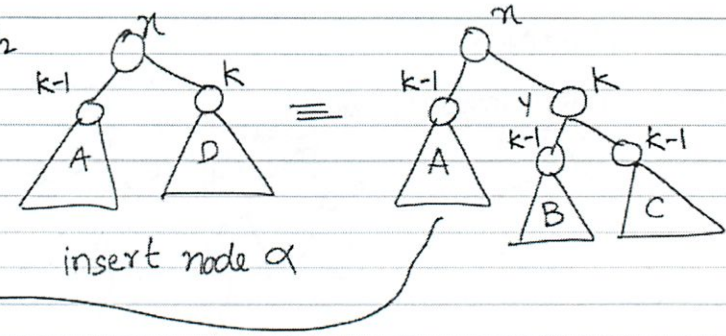


When we insert a node into node n, we have three possible cases.

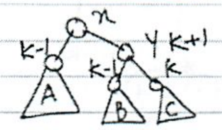
1. Children of n have same height k. Inserting into either sub-tree will still result in a valid AVL tree
2. The left child of n is heavier than the right child. Inserting into the left child may imbalance the AVL tree
3. The right child of n is heavier than the left child. Inserting into the right child may imbalance the AVL tree

When the AVL tree gets imbalanced, we must make rotations in the tree to re-arrange the nodes so that the AVL tree becomes balanced once again. Note that adding a node into a k height tree does not always turn it into a $k+1$ height tree, since we could have inserted the node on a shorter branch of that tree. However, for now, we are looking specifically at the situations where adding a node into a k height tree does turn it into a $k+1$ height tree. Let's examine the case where we insert a node into a heavy right child.

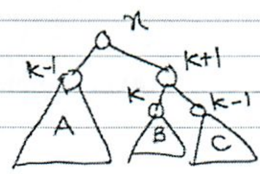
Consider a violation V_2



Case 1: goes to C

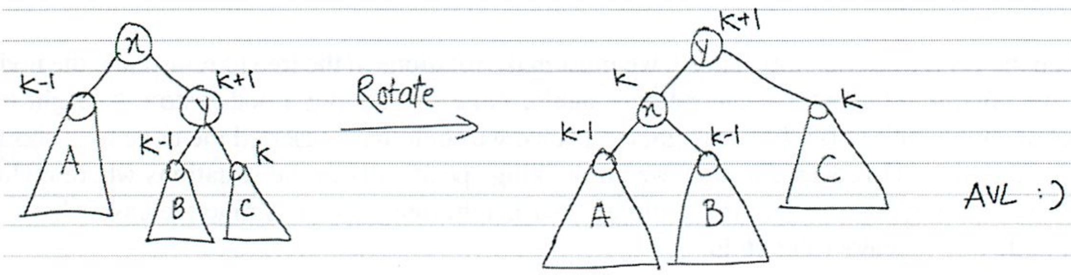


case 2: goes to B



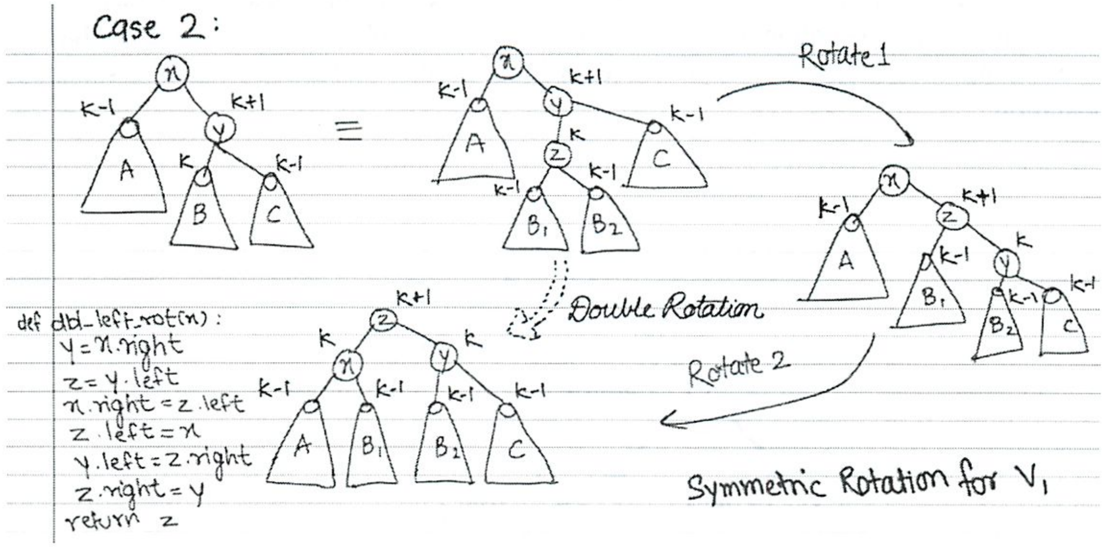
There are two cases here that will imbalance the AVL tree. We will once again look at the problem on a case by case basis.

Case 1:




```
def left_rot(n):
    y = n.right
    n.right = y.left
    y.left = n
return y
```

In the first case, B had height $k-1$, C had height $k-1$, and a node was inserted into C, making its current height k. We call a left rotation on n to make the y node the new root and shifting the B sub-tree over to be n's child. The order of the elements are preserved (In both trees, $A < n < B < y < C$), but after the rotation, we get a balanced tree.



In the second case, B had height $k-1$, C had height $k-1$, and a node was inserted into B, making its current height k. In this case, no single rotation on a node will result in a balanced tree, but if we make a right rotation on y and then a left rotation on x, we end up with a happy AVL tree.

If we insert a node into a heavy left child instead, the balancing solutions are flipped (i.e. right rotations instead of left rotations and vice versa), but the same concepts apply.

AVL insertions are binary search tree insertions plus at most two rotations. Since binary search tree insertions take $O(h)$ time, rotations are $O(1)$ time, and AVL trees have $h = O(\log n)$, AVL insertions take $O(\log n)$ time.

There are only a finite number of ways to imbalance an AVL tree after insertion. **AVL insertion is simply identifying whether or not the insertion will imbalance the tree, figuring out what imbalance case it causes, and making the rotations to fix that particular case.**

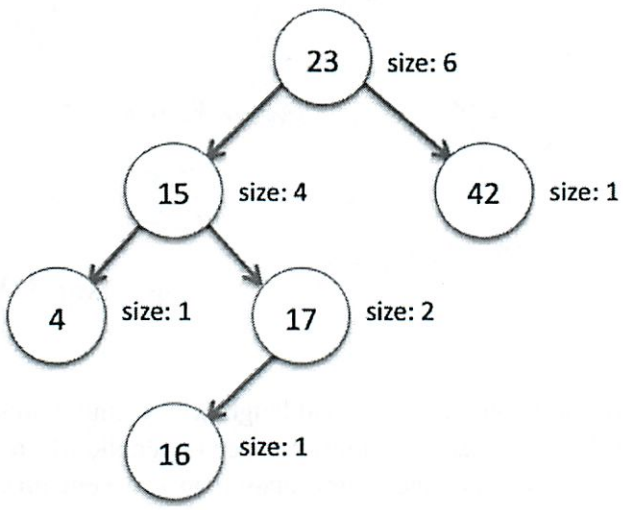
AVL Deletion

AVL deletion is not too different from insertion. The key difference here is that unlike insertion, which can only imbalance one node at a time, a single deletion of a node may imbalance several of its ancestors. Thus, when we delete a node and cause an imbalance of that node's parent, not only do we have to make the necessary rotation on the parent, but we have to traverse up the ancestry line, checking the balance, and possibly make some more rotations to fix the AVL tree.

Fixing the AVL tree after a deletion may require making $O(\log n)$ more rotations, but since rotations are $O(1)$ operations, the additional rotations does not affect the overall $O(\log n)$ runtime of a deletion.

Tree Augmentation

In AVL trees, we augmented each node to maintain the node's height and saw how that helped us maintain balance. Augmentation is a very useful tool to help us solve problems that a vanilla binary search tree cannot solve efficiently. We will learn about another useful augmentation, subtree size augmentation.



In subtree size augmentation, each node maintains the size of the subtree rooted at that node in a parameter `size`. The root of a tree containing n elements will have `size` $= n$ and each leaf of a tree will have `size` $= 1$.

The operations of this tree must maintain the `size` value of every node so that it is always correct.

For insertion, as we traverse down a branch to find where to insert a node, we need to increment `size` for every node that we visit. This is because going through a node during insertion means we will be inserting a node in the tree rooted at that node.

Deletion will also require some maintenance. Every time we remove a node, we must traverse up its ancestry and decrement `size` of all its ancestors.

If we wanted to augment an AVL tree with subtree size, we would also have to make sure that the rotation operations maintain `size` of all the nodes being moved around (Hint: the fact that x.size = x.left.size + x.right.size + 1 is useful here).

As you'll see in the problem set, using subtree augmentation can help speed up operations that normally would be slow on a regular binary search tree or AVL tree.

6.006 L5 Hashing

2/23

DNA matching

Given 2 strings S, T, finite alpha
Find longest string that appears in both
Used in Plagarism detection

Naive

for all substrings in S
  for all substrings in T
    Check

$\Omega(n^4)$
  Since comparing substrings

Pretty crappy

+ **Binary Search**
  - (has to be one of the 5 things we discussed in lecture)
  - if sucess → try larger L
  - if don't → try smaller L

$\Omega(n^3 \lg n)$ → better than $\Omega(n^4)$

(2)

Via BST

- fix an L
- put all the len L substrings - put into BST
  - takes log n

$\Omega(n^2 \lg n)$

but only for a given L

Need n Ls

$\Omega(n^2 \lg^2 n)$

Arrays

- indexed by substrings
- address for all possible strings - put yes where exists
- Then do for other array
- but substrings are not #s
- or is everything a #?
- So for every possible lenght L — n
- Insert all lenght L substrings of S into table — $O(nL)$
- For each L substrings of T - check if in table — $O(nL)$

$O(n^3)$

(3)

If binary search → $O(n^2 \lg n)$

Next time $O(n \lg n)$

Generalizing: Dictionaries (I know these!)

a set containing items
each item has a key
What are keys and items are quite flexible

Insert (item)
Delete (item)
Search (item)

Examples

spelling correction

[incorrect] → correct

Python interpreter

1. Dictionaries are everywhere
2. Anything in computer is a sea of bits
3. Dicts can be implemented by table

(4)

Can use array 2^{140} ↑ to fit longest word

But only 100,000 words

So can use hash functions

- exploits sparcity
- huge universe of possible keys
- want to store in table of size $m \sim n$

Define hash function $h: U \to \{1, \ldots, m\}$

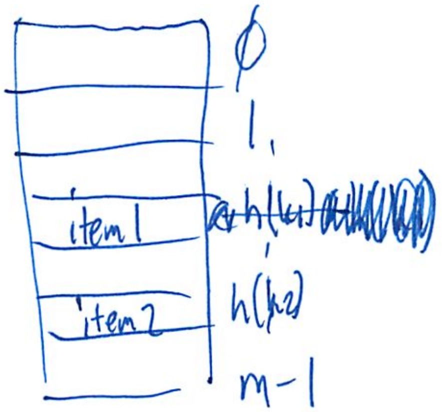
Filter key y through h() to find table position

Table entries are called buckets

Time to insert/find keys is

- Time to compute h
  - generally lenght of key
- Plus one time step to look in array

(5)



is no addr k1
But can hash h(k1)

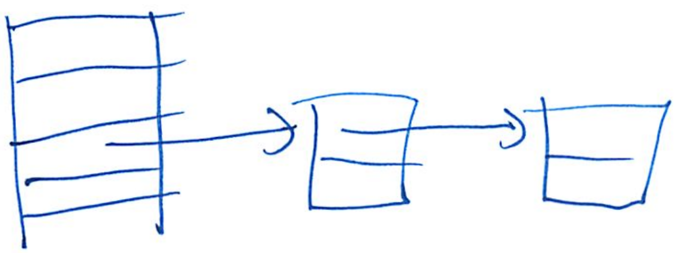
If hash collision $h(k_2) = h(k_4)$
└ distint keys k_2 k_4
- ~~that~~ but collided when taking hash function

If table is smaller than the range, some keys must collide
Pidgon hole principle

How to cope?
1. Change to a new uncolliding hash function
Remember $h(k_4) = h(k_1)$
↳ always hashes to same thing when same input
2. Chaining
Both things in budge
3. Open addressing (next time)

(6)

Chaining linked list



Problem solved?

Now must scan the lists entire bucket

↳ $\Theta(n)$ if they all hash to same thing

So be optimistic

Assume keys will be =ly distributed

"Simple Uniform Hashing" assumption

if use a good hashing function → yes

Theoretical: n items in table of m buckets

avg # $\alpha = \frac{n}{m}$

So expected time is $1 + \alpha$

$O(1)$ if $\alpha = O(m)$

Reality: keys have regularity

but we choose a random hash function

(7)

Ideal hash functions hard to implement

Pick a hash function whose values "look" random

Similar to pseudorandom generation

But always some set of keys are bad

__Division Hash Fn__

$h(k) = k \bmod m$

Collide if $k_1 \equiv k_2 \pmod{m}$

Unlikely if random

if m is power of 2 - just take low order bits (very fast)

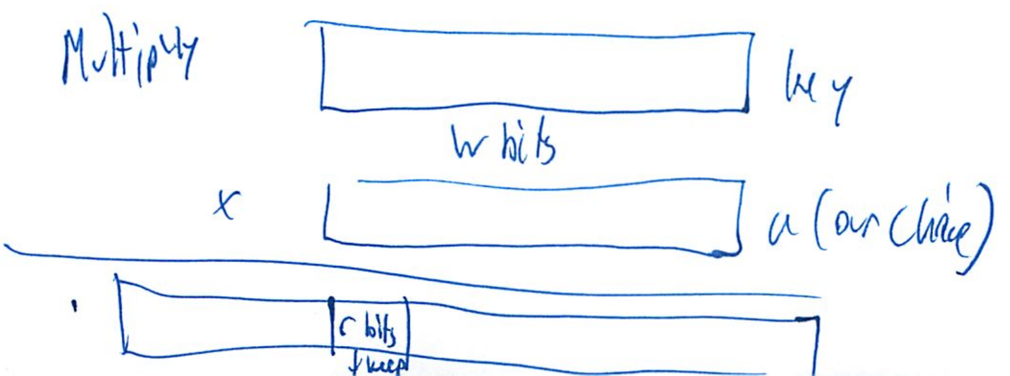
But regularity

Could make m a prime #

- hard to find
- regular

Suppose aiming for 2^r

Multiply



(8)

Python does this

prehashes to a large int

takes mod m to put in size m hash table

<u>Conclusion</u>

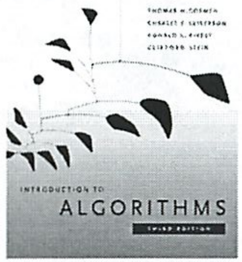
dictionaries are pervasive

hash tables implement them efficiently

Can beat by BST

- depends on which op you want to do

6.006- *Introduction to Algorithms*



Lecture 5

Prof. Silvio Micali

DNA matching

Given two strings S and T over same finite alphabet
Find largest substring that appears in both

If S=algorithm and T=arithmetic
Then return "rithm": algorithm arithmetic

- Also useful in plagiarism detection

- Say strings S and T of length n

Naïve Algorithm

- For L = n downto 1
- for all length-L substrings X1 of S
- for all length-L substrings X2 of T
- if X1=X2, return X1

Runtime analysis

– n candidate lengths L
– n substrings of that length in each of S and T
– L time to compare the strings
– Total runtime: $\Omega(n^4)$

+ Bynary search

- Start with L=n/2
- for all length L substrings X1 of S
- for all length L substrings X2 of T
- if X1=X2 (i.e., if success), then "try larger L"
  if failed, "try smaller L"

Runtime analysis

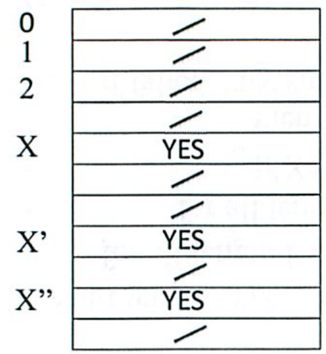
$\Omega(n^3 \log n)$ Better than $\Omega(n^4)$!

2/23

Via (Balanced) BSTs

Complexity?

~~BSTs~~ Array indexed by substrings



Wait1!

Substrings are no numbers!

OK, OK

For every possible length L=n,…,1 — n

Insert all length L substrings of S into table — O(nL)

For each length L substring of T, check if in table — O(nL)

Overall Complexity: $O(n^3)$

With binary search on length, total is $O(n^2 \log n)$

Next time: $O(n \log n)$

Generalizing: Dictionaries

- A **set** containing **items**; each item has a **key**
- what keys and items are is quite flexible
- Supported Operations:
  - **Insert(*item*):** add given *item* to set
  - **Delete(*item*):** delete given *item* to set
  - **Search(*key*):** return the item corresponding to the given *key,* if such an item exists

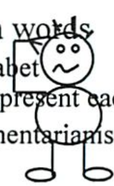
Other Examples

- Spelling correction
  - *Key* is a misspelled word, *item* is the correct spelling
- Python Interpreter
  - Executing program, see a variable name (*key*)
  - Need to look up its current assignment (*item*)
- …

Let me see if I understood…

(1) Dictionaries are everywhere
(2) Anything in the computer is a sequence of bits
(3) Dictionaries can be implemented by tables

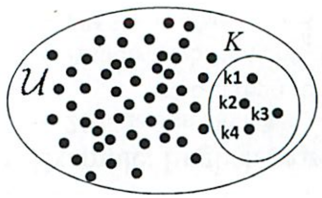
- Example: English words
  - 26 letters in alphabet
    ⇒ can represent each with 5 bits
  - Antidisestablishmentariamism has 28 letters
  - 28*5 = 140 bits
  - So, use array of size 2^{140}



- Isn't this too much space for 100,000 words?

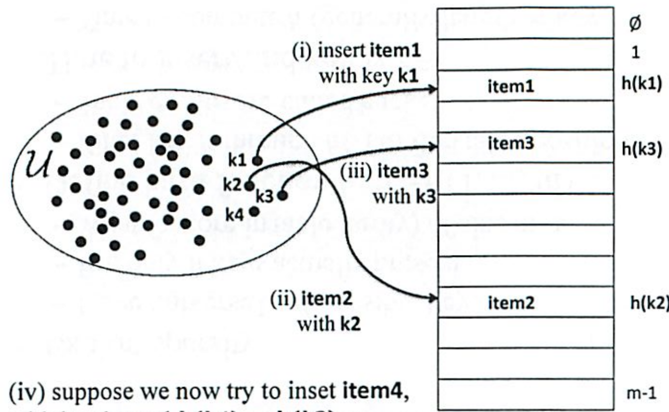
Hash Functions

- Exploit sparsity
  - Huge universe U of possible keys
  - But only n keys actually present
  - Want to store in table (array) of size m~n
- Define hash function $h: U \rightarrow \{1, \ldots, m\}$
  - Filter key k through $h(\,)$ to find table position
  - Table entries are called buckets
- Time to insert/find key is
  - Time to compute h (generally length of key)
  - Plus one time step to look in array

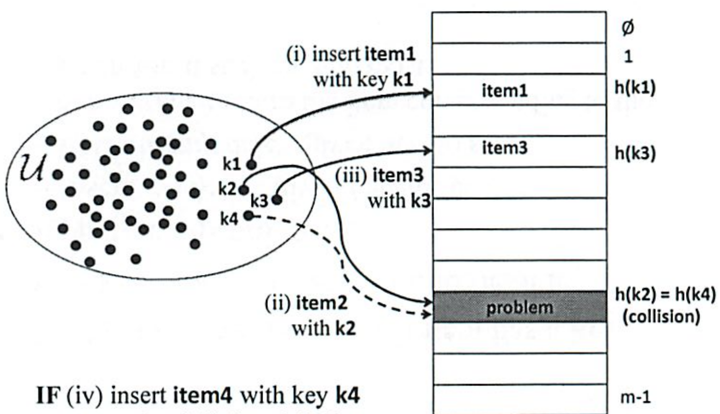


$\mathcal{U}$: *universe of all possible keys; huge set*

K : *actual keys; small set but not known in advance*



(iv) suppose we now try to inset **item4**, with key **k4** and $h(k4) = h(k2)$ ...



IF (iv) insert **item4** with key **k4** and $h(k4) = h(k2)$ ...

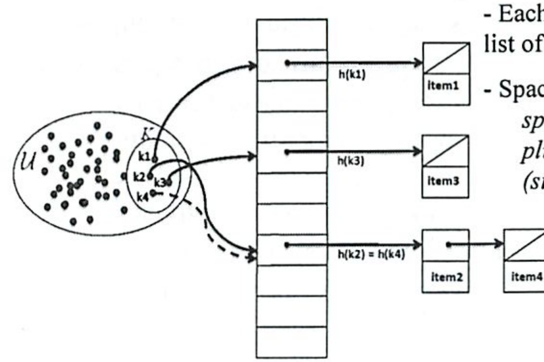
Collisions

- What went/can go wrong?
  - Distinct keys x and y
  - But h(x) = h(y)
  - Called a collision
- This is unavoidable: if table smaller than range, some keys must collide…
  - Pigeonhole principle
- What do you put in the bucket?

Coping with collisions

- **Idea1**: Change to a new "uncolliding" hash function
  - Hard to find, and takes time
- **Idea2: Chaining**
  - Put both items in same bucket (this lecture)
- **Idea3: Open addressing**
  - Find a different, empty bucket for y (next lecture)

Chaining



- Each bucket, linked list of contained items
- Space used is
  space of table
  plus one unit per item
  (size of key and item)

$\mathcal{U}$: *universe of all possible keys*

K : *actual keys, not known in advance*

Problem Solved?

- To find key, must scan whole list in key's bucket
- Length L list costs L key comparisons
- If all keys hash to same bucket, lookup cost $\Theta(n)$

Let's Be Optimistic !

- Assume keys are **equally likely** to land in every bucket, independently of where other keys land
- Call this
  the "Simple Uniform Hashing" assumption
  - (why/when can we make this assumption?)

Average Case Analysis under SUHA

- n items in table of m buckets
- Average number of items/bucket is $\alpha=n/m$
- So expected time to find some key x is $1+\alpha$
- O(1) if $\alpha=O(1)$, i.e. $m=\Omega(n)$

Reality

- Keys are often very nonrandom
  - Regularity (evenly spaced sequence of keys)
  - All sorts of mysterious patterns
- **Ideal** solution: **ideal** hash function: random

$$h: U \rightarrow \{1, \ldots, m\}$$

- Solution: pick a hash function whose values "look" random
- Similar to pseudorandom generators
- Whatever function, always some set of keys that is bad
  - but hopefully not your set

Division Hash Function

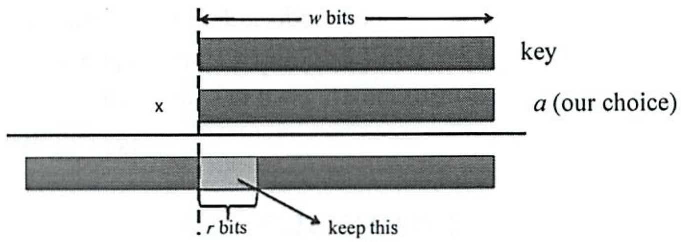
- $h(k) = k \bmod m$
- k_1 and k_2 collide when $k_1=k_2 \pmod m$
  - Unlikely if keys are random
- e.g. if m is a power of 2, just take low order bits of key
  - Very fast (a mask)
  - And people care about very fast in hashing

Problems

- Regularity
  - Suppose keys are x, 2x, 3x, 4x, ...
  - Suppose x and chosen m have common divisor d
  - Then (m/d)x is a multiple of m
    - so $i \cdot x = (i+m/d)x \bmod m$
  - Only use 1/d fraction of table
    - E.g, m power of 2 and all keys are even
- So make m a prime number
  - But finding a prime number is hard
  - And now you have to divide (slow)

Multiplication Hash Function

- Suppose we're aiming for table size 2^r
- and keys are w bits long, where $w>r$ is the machine word
- Multiply k with some a (fixed for the hash function)
- then keep certain bits of the result as follows



Python Implementation

- Python objects have a hash method
  - Number, string, tuple, any object implementing __hash__
- Maps object to (arbitarily large) integer
  - So really, should be called prehash
- Take mod m to put in a size-m hash table
- Peculiar details
  - Integers map to themselves
  - Strings that differ by one letter don't collide

Conclusion

- Dictionaries are pervasive
- Hash tables implement them efficiently
  - Under an optimistic assumption of random keys
  - Can be "made true" by choice of hash function
- How did we beat BSTs?
  - Used indexing
  - Sacrificed operations: previous, successor
- Next time: open addressing

Thank you!

Multiplication Hash Function

Bit shift

- **The formula:**

$$h(k) = [(a * k) \bmod 2^w] >> (w - r)$$

  - Multiply by a
  - When overflow machine word, wrap
  - Take high r bits of resulting machine word
  - (Assumes table size smaller than machine word)

Benefit: Multiplying and bit shifts faster than division
Good practice: Make a an odd integer (why?) > 2^{w-1}

Today's Topic

"Optimist pays off!"

a.k.a. The ubiquity and usefulness of ***dictionaries***

Implementation

- use BSTs!
  - can keep keys in a BST, keeping a pointer from each key to its value
  - O(log n) time per operation
- Often not fast enough for these applications!
- Can we beat BSTs?

  if only we could do all operations in O(1)…

[A parenthesis: DNA Matching

BSTs?

- For L=n downto 1
- Insert all length-L substrings of T into AVL tree
- For all length-L substring X2 of T,
  Try finding X2 in the tree

  if failed, try smaller L

- Runtime analysis

6.006 Recitation 2/24

Hashing

Hashing

- if have huge universe
- want to see if duplicates

U → Hash → map size n

Universe

↑ #s may be very large like multi GB files

↑ relatively few #s in n vs U

Allows us to test equality in constant time

Today: A very specific hash function → Polynomial hash

Strings How to find a random map

Bits

Data represented as bits/bytes

Bytes - smallest block of memory a computer can access into

(2)

Tuples + Strings can be broken into bytes

Smallest unit = word

32 bits

64 bits

<u>integer</u> size of largest # fitting into a single word

b/w -2^{31} and 2^{31}

Since we can do these operations in constant time

<u>Hash Fn</u>

treat string as an array of integers

$(a_1, a_2, \dots a_n)$

hash it down to 1

If any two lists are distinct, we want prob of collision to be $\frac{1}{2^{31}}$

It will take signed ints, but only output $0 \to 2^{31}$

(3)

Will think of a string as an array of #s

- └ how a string is actually represented

Convert characters to ASCII #s

It will involve mod (the remainder)

- └ to a prime #?
    - so that product is non-zero mod p if the 2 components are non zero mod p
- to 2^{31}?
    - so use full range
    - and is fast since can shift
- → but $2^{31}-1$ is prime
    - so use that
    - almost whole range (-1)
    - is prime
    - so fairly fast to do

Also $2^{19}-1$

(4)

(1.) To hash $(a_1, \ldots a_n)$ take $(a_1 + a_2 + \ldots a_n) \bmod p$

but $h(1,3) = h(3,1)$ ⊗ Not good enough

(2) $(\cancel{x}a_1 + \cancel{p}a_2 + p^2 a_3 + \cdots + p^{n-1} a_n) \bmod p$

(30,2)

(20,2)

But no fixed hash function can work

So chose h every time from a random family

Hash function must be chosen ind of the data

So will choose (2), but we will pick random b

- must keep b for entire table

Its easy to compute this polynomical hash

2b) $a_n + b^2 a_{n-1} + b^3 a_{n-2} + \cdots + b^{(n-1)} a_1$

(Just as easy to write both ways

(5)

Linear Time Code

```
                   ↓ array from index to n
                                  ↙ passing bounds instead
                                     of modifying array
def polyhash(array, base, index = 0):
    if index == len(array):
        return 0

    p = 2**31 - 1
    return array[index] + base * polyhash(array,
        base, index + 1)) % p
```

Why p should be prime

A =

$(a_1, a_2, \ldots, a_n) \Leftrightarrow A(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_n x^{n-1}$

$B = (b_1, \ldots b_n) \Leftrightarrow B(x)$

$A - B = (a_1 - b_1, \ldots, a_n - b_n)$

$A - B = \vec{0} = A == B$

↑ iff

$\vec{0} = A(x) - B(x) = 0$

(6)

Poly hash $(A, base) = A(base) \bmod p$

Two Polyhashes are = iff base is a root
of $A(x) - B(x) \bmod p$

How many roots can this polynomial have?
$n-1$ in most cases
except when they are = (when polynomial is 0)
└ then everything is a root

If $A(x) \neq B(x)$
$\deg(A(x) - B(x)) = n$ ← lenght of the array
→ Only n roots
→ prob of choosing a bad base is $\frac{n}{p}$

(any non 0 polynomial has a small # of roots)

(we wanted $\frac{1}{p}$, but got $\frac{n}{p}$. It does not matter since
$p \gg n$. We don't have to worry about my
arithmatic taking a while ...)

(7)

$p \doteq 2^{31}$ = 1 CPU op $\quad 2^{31} \sim 10^9$

$p \approx 2^{61}$ ~ 4 ops

So can grow p exponentially w/o taking much longer

So $\boxed{\frac{n}{p}}$ is exact collision prob

The important fact is each char chaning just changes
one calculation
- fairly easy to plug in
- log(p) time
  ~30 ops

So computing hash O(n)
If want to check all 1 char errors
- its $n \cdot \log(p) \cdot 26$

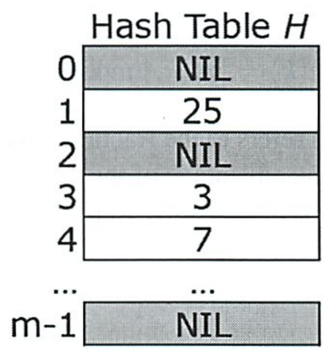
Overview of Hash Tables

A hash table is a data structure that supports the following operations:

- `insert(k)` - puts key k into the hash table
- `search(k)` - searches for key k in the hash table
- `remove(k)` - removes key k from the hash table

In a well formed hash table, each of these operations take on average $O(1)$ time, making hash tables a very useful data structure.

You can think of a hash table as a list of m slots. Inserting a key puts it in one of the slots in the hash table, deleting a key removes it from the slot it was inserted in, and searching a key looks in the slot the key would have been inserted into to see if it is indeed there. Empty slots are designated with a NIL value. The big question is figuring out which slot should a key k be inserted into in order to maintain the $O(1)$ runtime of these operations.



Hash Functions

Consider a function $h(k)$ that maps the universe U of keys (specific to the hash table, keys could be integers, strings, etc. depending on the hash table) to some index 0 to m. We call this function a **hash function**. When inserting, searching, or deleting a key k, the hash table hashes k and looks at the $h(k)$th slot to add, look for, or remove the key.

A good hash function

- satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots. The hash function shouldn't bias towards particular slots
- does not hash similar keys to the same slot (e.g. compiler's symbol table shouldn't hash variables `i` and `j` to the same slot since they are used in conjunction a lot)
- is quick to calculate, should have $O(1)$ runtime
- is deterministic. $h(k)$ should always return the same value for a given k

Example 1: Division method

The division method is one way to create hash functions. The functions take the form

$$h(k) = k \bmod m \tag{1}$$

Since we're taking a value mod m, $h(k)$ does indeed map the universe of keys to a slot in the hash table. It's important to note that if we're using this method to create hash functions, m should not be a power of 2. If $m = 2^p$, then the $h(k)$ only looks at the p lower bits of k, completely ignoring the rest of the bits in k. A good choice for m with the division method is a prime number (why are composite numbers bad?).

Example 2: Multiplication method

The multiplication method is another way to create hash functions. The functions take the form

$$h(k) = \lfloor m(kA \bmod 1) \rfloor \tag{2}$$

where $0 < A < 1$ and $(kA \bmod 1)$ refers to the fractional part of kA. Since $0 < (kA \bmod 1) < 1$, the range of $h(k)$ is from 0 to m. The advantage of the multiplication method is it works equally well with any size m. A should be chosen carefully. Rational numbers should not be chosen for A (why?). An example of a good choice for A is $\frac{\sqrt{5}-1}{2}$.

Collisions

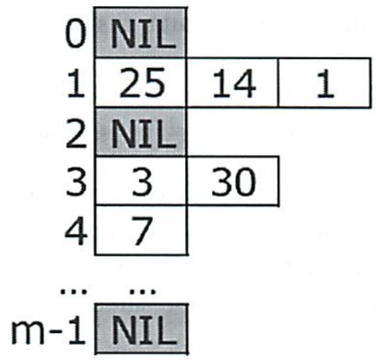
If all keys hash to different slots, then the hash table operations are as fast as computing the hash function and changing or inspecting the value of an array element, which is $O(1)$ runtime. However, this is not always possible. If the number of possible keys is greater than the number of slots in the hash table, then there must be some keys that hash into the same slot, in other words a **collision**. There are several ways to resolve a collision.

Chaining

In the chaining method of resolution, hash table slot j contains a linked list of every key whose hash value is j. The hash table operations now look like

- `insert(k)` - insert k into the linked list at slot $h(k)$
- `search(k)` - search for k in the linked list at slot $h(k)$ by iterating through the list
- `remove(k)` - search for k in the linked list at slot $h(k)$ and then remove it from the list

With chaining, if a key collides with another key, it gets inserted into the same linked list in the slot they hash into.



In the ideal case, all keys hash to different slots and every linked list has at most 1 element, keeping the runtimes of the operations at $O(1)$. In the worst case, all n keys inserted into the hash table hashes to the same slot. We then get a n size linked list which takes $O(n)$ to search through, resulting in $O(n)$ search and remove. This is why choosing a hash function that equally distributes keys to all slots is important.

If there are n keys in a hash table with m slots, we call the **load factor** α for the hash table to be $\frac{n}{m}$. Under the assumption of simple uniform hashing, the length of each linked list in the hash table is α. As long as the number of keys inserted is proportional to the size of the hash table, $\alpha = O(1)$, thus the operations on average are $O(1)$ as well.

Open Addressing Collisions

A hash table may use **open addressing**, which means that each slot in the hash table contains either a single key or NIL to indicate that no key has been hashed in that slot. Unlike chaining, we cannot fit more than one key in a single slot, so we must resolve collisions in a different way. We must have a method to determine which slot to try next in the case of a collision. We still try to put a key k into slot $h(k)$ first, but if that slot is occupied, we keep trying new slots until we find an empty one to put the key into.

Linear probing resolves collisions by simply checking the next slot, i.e. if a collision occurred in slot j, the next slot to check would be slot $j + 1$. More formally, linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m \tag{3}$$

Where $h'(k)$ is the hash function we try first. If $h(k, 0)$ results in a collision, we increment i until we find an empty slot. One drawback to linear probing is if keys hash to slots close to each other, a cluster of adjacent slots get filled up. When trying to insert future keys into this cluster, we must then traverse through the entire cluster in order to find an empty slot to insert into, which can slow down our hash table operations.

Quadratic probing resolves collisions in a similar fashion:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \tag{4}$$

for some constants c_1, c_2. Instead of linearly traversing through the hash table slots in the case of collisions, quadratic probing introduces more spacing between the slots we try in case of a collision, which reduces the clustering effect seen in linear probing. However, a milder form of clustering can still occur, since keys that hash to the same initial value will probe the exact same sequence of slots to find an empty slot.

Double hashing resolves collisions by using another hash function to determine which slot to try next:

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m \tag{5}$$

With double hashing, both the initial probe slot and the method to try other slots depend on the key k, which further reduces the clustering effect seen in linear and quadratic probing.

Searching for a key in a hash table using open addressing involves probing through slots until we find the key we want to find or NIL. If we encounter a slot with a NIL value before finding the key itself, that means that the key in question is not in the hash table.

Deleting for a key involves searching for the key first. Once the key to be deleted is found, we remove it by replacing the key in that slot with a dummy DELETED value. Note that we cannot replace the key with a NIL value, or else searching for keys further down in the probe sequence will falsely return NIL. We must replace it with a dummy value indicating that a key was once present in this slot, but not anymore.

Rolling Hash (Rabin-Karp Algorithm)

Objective

If we have text string S and pattern string P, we want to determine whether or not P is found in S, i.e. P is a substring of S.

Notes on Strings

Strings are arrays of characters. Characters however can be interpreted as integers, with their exact values depending on what type of encoding is being used (e.g. ASCII, Unicode). This means we can treat strings as arrays of integers. Finding a way to convert an array of integers into a single integer allows us to hash strings with hash functions that expect numbers as input.

Since strings are arrays and not single elements, comparing two strings for equality is not as straightforward as comparing two integers for equality. To check to see if string A and string B are equal, we would have to iterate through all of A's elements and all of B's elements, making sure that $A[i] = B[i]$ for all i. This means that string comparison depends on the length of the strings. Comparing two n-length strings takes $O(n)$ time. Also, since hashing a string usually involves iterating through the string's elements, hashing a string of length n also takes $O(n)$ time.

Method

Say P has length L and S has length n. One way to search for P in S:

1. Hash P to get $h(P)$ **O(L)**
2. Iterate through all length L substrings of S, hashing those substrings and comparing to $h(P)$ **O(nL)**
3. If a substring hash value does match $h(P)$, do a string comparison on that substring and P, stopping if they do match and continuing if they do not. **O(L)**

This method takes $O(nL)$ time. We can improve on this runtime by using a **rolling hash**. In step 2. we looked at $O(n)$ substrings independently and took $O(L)$ to hash them all. These substrings however have a lot of overlap. For example, looking at length 5 substrings of "algorithms", the first two substrings are "algor" and "lgori". Wouldn't it be nice if we could take advantage of the fact that the two substrings share "lgor", which takes up most of each substring, to save some computation? It turns out we can with rolling hashes.

"Numerical" Example

Let's step back from strings for a second. Say we have P and S be two integer arrays:

$$P = [9, 0, 2, 1, 0] \quad (1)$$
$$S = [4, 8, 9, 0, 2, 1, 0, 7] \quad (2)$$

The length 5 substrings of S will be denoted as such:

$$S_0 = [4, 8, 9, 0, 2] \quad (3)$$
$$S_1 = [8, 9, 0, 2, 1] \quad (4)$$
$$S_2 = [9, 0, 2, 1, 0] \quad (5)$$
$$\ldots \quad (6)$$

We want to see if P ever appears in S using the three steps in the method above. Our hash function will be:

$$h(k) = (k[0]10^4 + k[1]10^3 + k[2]10^2 + k[3]10^1 + k[4]10^0) \bmod m \quad (7)$$

Or in other words, we will take the length 5 array of integers and concatenate the integers into a 5 digit number, then take the number mod m. $h(P) = 90210 \bmod m$, $h(S_0) = 48902 \bmod m$, and $h(S_1) = 89021 \bmod m$. Note that with this hash function, we can use $h(S_0)$ to help calculate $h(S_1)$. We start with 48902, chop off the first digit to get 8902, multiply by 10 to get 89020, and then add the next digit to get 89021. More formally:

$$h(S_{i+1}) = [(h(S_i) - (10^5 * \text{ first digit of } S_i)) * 10 + \text{ next digit after } S_i] \bmod m \quad (8)$$

We can imagine a window sliding over all the substrings in S. Calculating the hash value of the next substring only inspects two elements: the element leaving the window and the element entering the window. This is a dramatic difference from before, where we calculated each substring's hash values independently and would have to look at L elements for each hash calculation. Finding the hash value of the next substring is now a $O(1)$ operation.

In this numerical example, we looked at single digit integers and set our base $b = 10$ so that we can interpret the arithmetic easier. To generalize for other base b and other substring length L, our hash function is

$$h(k) = (k[0]b^{L-1} + k[1]b^{L-2} + k[2]b^{L-3}...k[L-1]b^0) \bmod m \quad (9)$$

And calculating the next hash value is:

$$h(S_{i+1}) = b(h(S_i) - b^{L-1}S[i]) + S[i + L] \bmod m \quad (10)$$

Back to Strings

Since strings can be interpreted as an array of integers, we can apply the same method we used on numbers to the initial problem, improving the runtime. The algorithm steps are now:

1. Hash P to get $h(P)$ **O(L)**
2. Hash the first length L substring of S **O(L)**
3. Use the rolling hash method to calculate the subsequent $O(n)$ substrings in S, comparing the hash values to $h(P)$ **O(n)**
4. If a substring hash value does match $h(P)$, do a string comparison on that substring and P, stopping if they do match and continuing if they do not. **O(L)**

This speeds up the algorithm and as long as the total time spent doing string comparison is $O(n)$, then the whole algorithm is also $O(n)$. We can run into problems if we expect $O(n)$ collisions in our hash table, since then we spend $O(nL)$ in step 4. Thus we have to ensure that our table size is $O(n)$ so that we expect $O(1)$ total collisions and only have to go to step 4 $O(1)$ times. In this case, we will spend $O(L)$ time in step 4, which still keeps the whole running time at $O(n)$.

Common Substring Problem

The algorithm described above takes in a specific pattern P and looks for it in S. However, the problem we've dealt with in lecture is seeing if two long strings of length n, S and T, share a common substring of length L. This may seem like a harder problem but we can show that it too has a runtime of $O(n)$ using rolling hashes. We will have a similar strategy:

1. Hash the first length L substring of S **O(L)**
2. Use the rolling hash method to calculate the subsequent $O(n)$ substrings in S, adding each substring into a hash table **O(n)**
3. Hash the first length L substring of T **O(L)**
4. Use the rolling hash method to calculate the hash values subsequent $O(n)$ substrings in T. For each substring, check the hash table to see if there are any collisions with substrings from S. **O(n)**
5. If a substring of T does collide with a substring of S, do a string comparison on those substrings, stopping if they do match and continuing if they do not. **O(L)**

However, to keep the running time at $O(n)$, again we have to be careful with limiting the number of collisions we have in step 5 so that we don't have to call too many string comparisons. This time, if our table size if $O(n)$, we expect $O(1)$ substrings in each slot of the hash table so we expect $O(1)$ collisions for each substring of T. This results in a total of $O(n)$ string comparisons which takes $O(nL)$ time, making string comparison the performance bottleneck now. We can increase table size and modify our hash function so that the hash table has $O(n^2)$ slots, leading to an expectation of $O(\frac{1}{n})$ collisions for each substring of T. This solves our problem and returns the total runtime to $O(n)$ but we may not necessarily have the resources to create a large table like that.

Instead, we will take advantage of string **signatures**. In addition to inserting the actual substring into the hash table, we will also associate each substring with another hash value, $h_s(k)$. Note that this hash value is different from the one we used to insert the substring into the hash table. The $h_s k$ hash function actually maps strings to a range 0 to n^2 as opposed to 0 to n like $h(k)$. Now, when we have collisions inside the hash table, before we actually do the expensive string comparison operation, we first compare the signatures of the two strings. If the signatures of the two strings do not match, then we can skip the string comparison. For two substrings k_1 and k_2, only if $h(k_1) = h(k_2)$ and $h_s(k_1) = h_s(k_2)$ do we actually make the string comparison. For a well chosen $h_s(k)$ function, this will reduce the expected time spent doing string comparisons back to $O(n)$, keeping the common substring problem's runtime at $O(n)$.

6.006 L6

2/28

Alan Deckelbaum

CLRS Chap 17 + 32.2

PS 2 changed Q1 or Q2

PS1 was graded except 3a, 3b so max 65/75

Today: More hashing

Dictionary → insert, delete, find a key

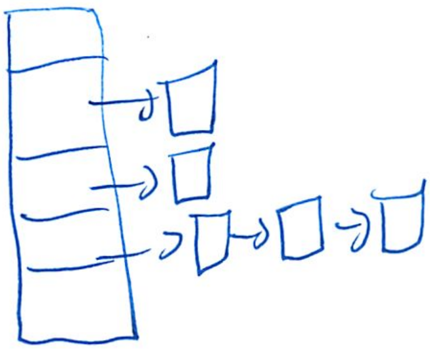
Hash table → implements dictionary
but spreads into over array
Uses hash function to map into smaller buckets
but can have collissions
- could build linked list chain

SUHA - hash fn behaves random, uniformally on bucket
expected load $\frac{n \in \text{items}}{m \in \text{buckets}}$ per bucket

Search $O(1 + \frac{n}{m})$

↑ hash + lookup ↑ scan linked list

(2)



Hash functions

- Division

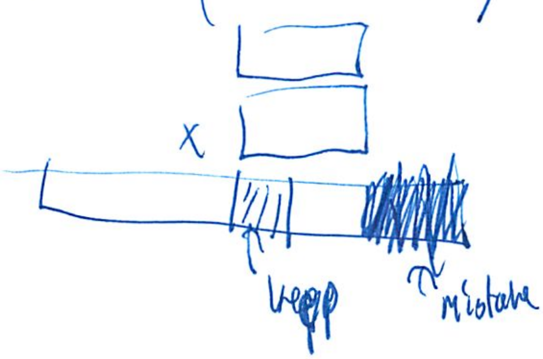
$h(k) = k \bmod m$

fast if m is a power of 2

bad if keys regular

- Multiplication

$h(k) = (a \cdot k \bmod 2^w) >> w - r$



Non #s

- String
  - stored as a series of bits
  - view as #
  - hash

(3)

$the = t(26)^2 + h(26) + e$

↑ "location" of string

but base 26

Note - hash time is lenght of String

↳ not $O(1)$

Back to longest common substring problem

Last time

$O(n^4) \rightarrow O(n^3 \log n) \rightarrow O(n^2 \log n)$

simple/naive — binary search — hashing

Today: even better

↳ binary search on max match L

insert all in hash table lenght L substrings of S

For each lenght L substring of T

Check the hash table

Binary Search $O(\log n)$

For each lenght L substrings

$n - L$ = # of L lenght substrings

Each hash takes L time

$O((n-L)L) = O(n^2)$

(4)

Hashing all L-lenght substrings of T

n-L hashes

another $\theta(n^2)$

Time for comparing substrings of T to substrings of S

Under SUHA $O(1)$ for each substring of S

Each comparison $O(L)$

So for all $O(nL) = O(n^2)$

So this is $O(n^2 \log n)$ inc. Binary Search

Faster?

Amdahl's law

- if code only takes 20% of the time you can only get max 20% speedup

And we have ~5 sections each ~20%

So must improve all asy worst parts

(5)

In our case
- must compute n hashes
- (missed)

Faster comparison

1st idea: When we find a match for some lenght we can stop and go to next value of lenght in our binary search.

But real problem is false positives
- strings in same buckets that don't match
- where our n^2 is coming from

Details
- n substrings to size-n table
  - ↳ avg load 1
- SUHA: for every substring x of T there is 1 other string in bucket
- Comparison L per string
- Total work $nL = \theta(n^2)$

(I have not been writing θ)

(6)

Sol: bigger table

So $m = n^2$

So avg load $\frac{n}{n^2} = \frac{1}{n}$

So comparison $\frac{L}{n}$

So total work $n\left(\frac{L}{n}\right) = L = O(n)$

But n^2 sized table might become too big

Solution: Signatures

n^2 ← size of table is not needed for fast look up

n would work

So don't make n^2 table

Just deal w/ false positives

So for each string, compute another hash value in the larger range $1 \ldots n^2$ called a signature

So only need to compare if

- same hash row AND same sig
- very small prob

(2)

P[same sig for 2 diff string] = $\frac{1}{n2}$

So hash substring to size n table

store sig w/ each substring

Check T string against its bucket

Comparing sigs is O(1)

^ would be O(log n); but if $n^2 < 2^{32}$

the sig fits inside a word so

it actually takes 1 op

Runtime

for each T-string

O(bucket size) = O(1) to compare sigs

So overall O(n) time in sig comparison

L × (Expected total # of False Sig Collisions)

- n out of the n^2 values in $[1, \ldots n^2]$

are used by S-strings

- So prob of T string sig collision

w/ S string $\frac{1}{n^2}$

- So total # collisions = 1

(8)

So total sting comparisons is L

(But we still need to compute sigs → so can $O(n)$)
see below

Faster Hashing

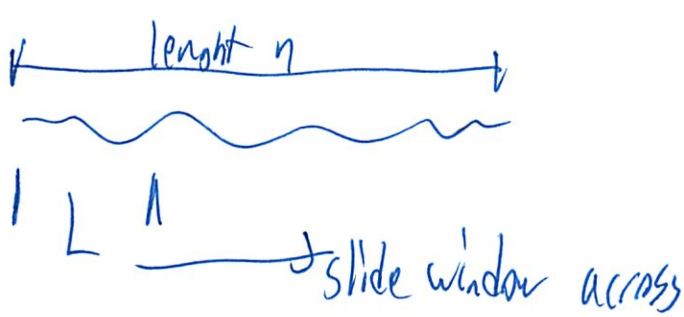
Rolling Hash

- seq of n substring hashes

$$O(nL) = O(n^2)$$

Better?

Yes - lots of redundancy



Example

There w/ $L = 3$ then

So old way

the
her
ere

| seperatly

(9)

But we can do some regrouping

$$= h(26)^2 + e(26) + r$$

$$= 26 \cdot (h(26) + e) + r$$

$$= 26 \cdot (t(26)^2 + h(26) + e - t(26^2)) + r$$

$$= 26(\text{"the"} - t(26)^2) + r$$

[Subtract 1st letter
Shift
add last letter]

Strings = base b #s

$$S[i] \cdot b^{L-1} + S[i+1] \cdot b^{L-2} + \cdots + S[i+L-1]$$

So constant work
Since did not need to go through entire string

(10)

But working w/ big #s

$S[i \dots i+L-1]$ may be so huge

Can't store on computer

We take mod m afterwards

So instead take things mod m at intermedia points

$(ab) \bmod m = (a \bmod m)(b \bmod m) \pmod m$

$(a+b) \bmod m = (a \bmod m) + b (\bmod m) \pmod m$

So now $O(L)$ to hash strings

Same for $S[i+1 \dots i+L]$

So done in $O(1)$ if we know $b^L \bmod m$

→ So computing $n-L$ hashes cost $O(n)$

⑪

$O(L)$ for 1st hash

$+ O(L)$ to compute 6-L

(missed)

Summary

Reduced compare costs $\frac{O(n)}{\text{length}}$

- By using big hash table

Or signs

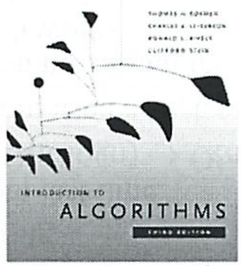
Reduced hash computation to $\frac{O(n)}{\text{lenght}}$

- Rolling hash

Total cost of phases $O(n \log n)$

Not the end; suffix trees achieve $O(n)$

6.006- *Introduction to Algorithms*



Lecture 6

Alan Deckelbaum

CLRS: Chapter 17 and 32.2.

LAST TIME...

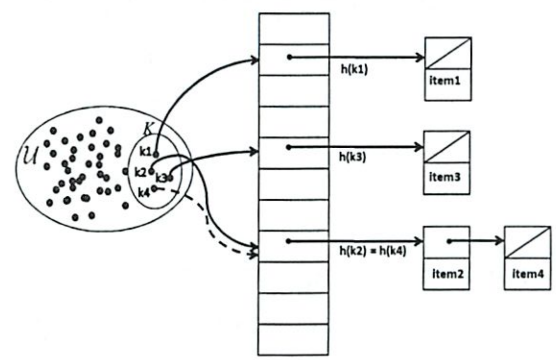
Dictionaries, Hash Tables

- **Dictionary**: Insert, Delete, Find a key
  - can associate a whole item with each key
- **Hash table**
  - implements a dictionary, by spreading items over an array
  - uses *hash function*

    h: Universe of keys (huge) → Buckets (small)
  - *Collisions*: Multiple items may fall in same bucket
  - *Chaining Solution*: Place colliding items in linked list, then scan to search
- **Simple Uniform Hashing** Assumption (SUHA):

  h is "random", uniform on buckets
  - Hashing n items into m buckets → expected "load" per bucket: n/m
  - If chaining used, expected search time O(1 + n/m)

Hash Table with Chaining



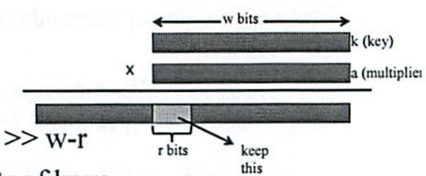
$\mathcal{U}$: *universe of all possible keys-huge set*

K : *actual keys-small set, but not known when designing data structure*

2/28

Hash Functions?

- Division hash
  - $h(k) = k \bmod m$
  - Fast if m is a power of2, slow otherwise
  - Bad if e.g. keys are regular
- Multiplication hash
  - a an odd integer
  - $h(k) = (a \cdot k \bmod 2^w) >> w\text{-}r$
  - Better on regular sets of keys



Non-numbers?

- What if we want to hash e.g. strings?
- Any data is bits, and bits are a number
- E.g., strings:
  - Letters a..z can be "digits" base 26.
  - "the" $= t\cdot(26)^2 + h\cdot(26) + e$
    $= 19\cdot(676) + 8\cdot(26) + 5$
    $= 334157$
- Note: hash time is length of string, not O(1) (wait a few slides)

Longest Common Substring

- Strings S,T of length n, want to find longest common substring
- Algorithms from last time:
  $O(n^4) \rightarrow O(n^3 \log n) \rightarrow O(n^2 \log n)$
- Winner algorithm used a hash table of size n:

> Binary search on maximum match length L; to check if a length works:
> - Insert all length-L substrings of S in hash table
> - For each length-L substring x of T
>   - Look in bucket h(x) to see if x is in S

Runtime Analysis

- Binary search cost: O(log n) length values L tested
- For each length value L, here are the costly operations:
  - Inserting all L-length substrings of S: n-L hashes
    - Each hash takes L time, so total work $\Theta((n-L)L)=O(n^2)$
  - Hashing all L-length substrings of T: n-L hashes
    - another $O(n^2)$
  - Time for comparing substrings of T to substrings of S:
    - How many comparisons?
    - Under SUHA, each substring of T is compared to an expected O(1) of substrings of S found in its bucket
    - Each comparison takes O(L)
    - Hence, time for all comparisons: $\Theta(nL)=O(n^2)$
- So $O(n^2)$ work for each length
- Hence $O(n^2 \log n)$ including binary search

Faster?

- Amdahl's law: if one part of the code takes 20% of the time, then no matter how much you improve it, you only get 20% speedup
- Corollary: must improve all asymptotically worst parts to change asymptotic runtime
- In our case
  - Must compute sequence of n hashes faster
  - Must reduce cost of comparing in bucket

FASTER COMPARISON

Faster Comparison

- **First Idea:** when we find a match for some length, we can stop and go to the next value of length in our binary search.
- **But,** the real problem is "false positives"
  - Strings in same bucket that don't match, but we waste time on
- **Analysis:**
  - n substrings to size-n table: average load **1**
  - SUHA: for every substring x of T, there is **1** other string in x's bucket (in expectation)
  - Comparison work: **L** per string (in expectation)
  - So total work for all strings of T: $\mathbf{nL = O(n^2)}$

Solution: Bigger table!

- What size?
- Table size $\mathbf{m = n^2}$
  - n substrings to size-m table: average load **1/n**
  - SUHA: for every substring x of T, there is **1/n** other strings in x's bucket (in expectation)
  - Comparison work: **L/n** per string (in expectation)
  - So total work for all strings of T: $\mathbf{n(L/n) = L = O(n)}$
- Downside?
  - Bigger table
  - (n^2 isn't realistic for large n)

Signatures

- Note n^2 table isn't needed for fast lookup
  - Size n enough for that
  - n^2 is to reduce cost of false positive compares
- So don't bother making the n^2 table
  - Just compute for each string another hash value in the larger range $1..n^2$
  - Called a signature
  - If two signatures differ, strings differ
  - Pr[same sig for two different strings] = $1/n^2$
    - (simple uniform hashing)

Application

- Hash substrings to size n table
- But store a signature with each substring
  - Using a second hash function to $[1..n^2]$
- Check each T-string against its bucket
  - First check signature, if match then compare strings
  - Signature is a small number, so comparing them is O(1)

strictly speaking O(logn); but if $n^2<2^{32}$ the signature fits inside a word of the computer; in this case, the comparison takes O(1)

Application

- Runtime Analysis:
  - for each T-string:
    O(bucket size)=O(1) work to compare signatures;
  - so overall O(n) time in signature comparisons
  - Time spent in string comparisons?

    L x (Expected Total Number of False-Signature Collisions)
    - n out of the n^2 values in $[1..n^2]$ are used by S-strings
    - so probability of a T-string signature-colliding with some S-string: n/n^2
    - hence total expected number of collisions 1

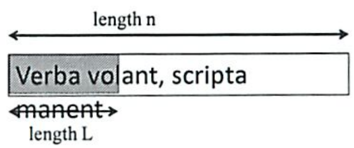
    so total time spent in String Comparisons is L

fine print: we didn't take into account the time needed to compute signatures; we can compute all signatures in O(n) time using trick described next…

FASTER HASHING

Rolling Hash

- We make a sequence of n substring hashes
  - Substring lengths L
  - Total time $O(nL) = O(n^2)$
- Can we do better?
  - For our particular application, yes!



Rolling Hash Idea

- e.g. hash all 3-substrings of "there"
- Recall division hash: x mod m
- Recall string to number:
  - First substring "the" = $t\cdot(26)^2 + h\cdot(26) + e$
- If we have "the", can we compute "her"?

"her" $= h\cdot(26)^2 + e\cdot(26) + r$

$= 26 \cdot \left(h\cdot(26) + e \right) + r$

$= 26 \cdot \left(t\cdot(26)^2 + h\cdot(26) + e - t\cdot(26)^2 \right) + r$

$= 26 \cdot \left(\text{"the"} - t\cdot(26)^2 \right) + r$

- i.e. subtract first letter's contribution to number, shift, and add last letter

General rule

- Strings = base-b numbers
- Current substring S[i … i+L-1]

$$S[i] \cdot b^{L-1} + S[i+1] \cdot b^{L-2} + S[i+2] \cdot b^{L-3} \ldots + S[i+L-1]$$
$$- S[i] \cdot b^{L-1}$$
$$S[i+1] \cdot b^{L-2} + S[i+2] \cdot b^{L-3} \ldots + S[i+L-1]$$
$$\times b$$
$$S[i+1] \cdot b^{L-1} + S[i+2] \cdot b^{L-2} \ldots + S[i+L-1] \cdot b$$
$$+ S[i+L]$$
$$S[i+1] \cdot b^{L-1} + S[i+2] \cdot b^{L-2} \ldots + S[i+L-1] \cdot b + S[i+L]$$
$$= S[i+1 \ldots i+L]$$

Mod Magic 1

- So: **S[i+1 … i+L] = b S[i … i+L-1] – b^L S[i] + S[i+L]**
- where

  S[i … i+L-1] = S[i] · b^{L-1} + S[i+1] · b^{L-2} + … + S[i+L-1] (*)
- **But** S[i … i+L-1] may be a huge number (so huge that we may not even be able to store in the computer, e.g. L=50, b=26)
- **Solution** only keep its ***division hash***: S[…] mod m
- This can be computed without computing S[…], using mod magic!
- Recall: **(ab) mod m = (a mod m) (b mod m) (mod m)**
  (a+b) mod m = (a mod m) + (b mod m) (mod m)
- With a clever parenthesization of (*): O(L) to hash string!

Mod Magic 2

- Recall: S[i+1 … i+L] = b S[i … i+L-1]– b^L S[i] + S[i+L]
- Say we have hash of S[i … i+L-1], can we still compute hash of S[i+1 … i+L] ?
- Still mod magic to the rescue!
- Job done in O(1) operations, if we know b^L mod m

➡ Computing n-L hashes costs O(n)

O(L) time for the first hash
+O(L) to compute b^L mod m
+ O(1) for each additional hash

Summary

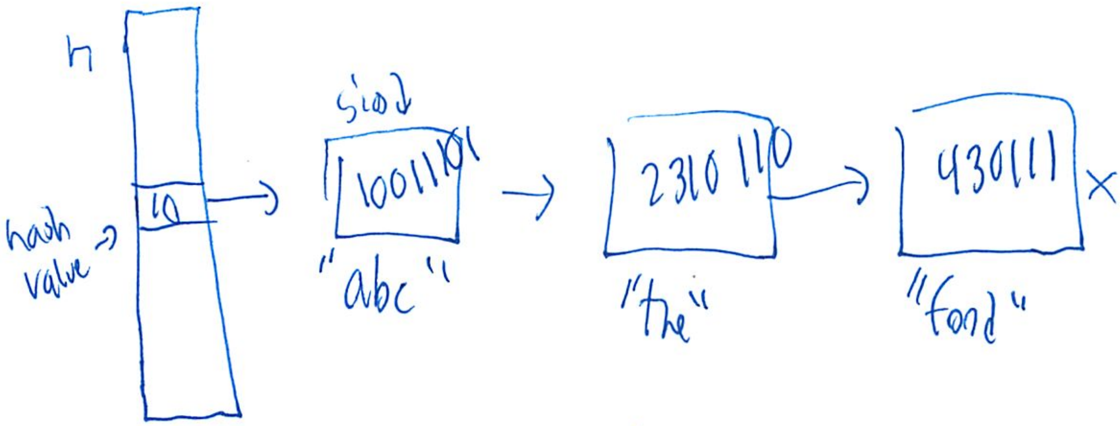
- Reduced compare cost to O(n)/length
  - By using a big hash table
  - Or signatures in a small table
- Reduced hash computation to O(n)/length
  - Rolling hash function
- Total cost of phases: O(n log n)
- Not the end: suffix tree achieves O(n)

6.006 Recitation

2/29
Leap Day!

(2-3 min late)

Signatures



Signature - 2nd hash fn
keep larger/ longer value

Compare sig O(1) since hash so much shorter

this is O(L) since L to hash
1 to find row
1 each to compare
↳ since are 1 word so 1 op
← as γ L

Python dictionaries already does this
for d['foot'] → computes h('foot') sig('foot') → goes to cell h → searches for entry w/ sig s

②

Still does a full comparison - in case both ~~ash~~ h and sig collission

2 ~~[illegible]~~ ways to use hasing

- key - value lookup (what we just saw)
  - dictionaries
- test equality
  - substring equality testing from lecture
  - or when download file, get checksum, hash, compare hash w/ checksum

- in key value

  $0 \le h \le n$

  * rel. small

  so no guarentee of no collissons since want to store in table

in test equality

no table so longer hash

$1 \ldots 2^{64}$

low - no chance of collission

Checksum = hash value w/ long range

or 2^{64} could be larger like in Dropbox - to better see uniqueness

(3)

Birthday Paradox

If ~~#~~ < 23 people, there is a > 50% change 2 have same birthday

Since $\binom{n}{2}$ "n choose 2" grows n^2

← # of pairs of people ~~← # of pairs of people~~

$\frac{23 \cdot 23}{2} \approx 260$

So are about $\approx n^2$ pairs of files that could collide

So hash value is $\{1, 2, \ldots, R\}$

Prob of collision $\sim \frac{n^2}{R}$ TA: ← sketchy math

So if 2^{64} files and 2^{256} possible hash values

$\frac{(2^{64})^2}{2^{256}} = \frac{1}{2^{128}}$ ← astronomically small

(4)

So we suggested range 1 → n^2
because of this calculation

Mail systems also do this
- only 1 copy of attachments or even message
- then just store the hash of the message

Hashing in Py

~~Class Foo (object):~~

ints → Python hashes the int

for object can override __hash__

Py can only store objects in dict that it knows how to hash

Need to make sure hash fn is <u>unique</u>
So for ordered pair

(self.x, self.y) tuple

self.x + self.y ⊗ No 1,1 would be same as 0,0

(5)

Random if you want each instance of Foo obj different

Could not test equality then since they all look different

Does NOT use hash to test equality

By default pointer equality (so must be same exact obj)

But can override hash

P-Set is about writing hash fns - but not a special class

Can't add members to class

Can't spend time to convert to class

Subtile Detail of Hashing

↓

(6)

SUHA → # of keys in cell $= \frac{n}{m}$

Expected time of lookup

$= O\left(1 + 1 + \frac{n}{m} + 1\right)$

- time to compute h and sig
- table lookup
- compare each item "load factor"
- final equality test

What do we say $m = \Theta(n)$

If keep inserting items → $\frac{n}{m}$ ↑, # collisions ↑ and horiz compare time ↑

~~If keep inserting~~

We can't just append rows at bottom

Since on resize $m \to m+1$

We need to rehash everything $O(n)$

This is disasterous if we resize on every insertion

So when reach $n = m$, double m to $2m$

So then can have m more inserts w/ current table

Will need to rehash everything $O(m)$

Then get $O(1)$ time for next inserts

①

- Half ~~table~~ table when $n = \frac{m}{2}$?? to save space when
  - load factor deleting items

So load factor too small?

But $\frac{m}{2}$ <u>not</u> where cut in half

But then add 1 obj - it will double it again

So just turn to $n = \frac{m}{4}$ → cut in <u>half</u>

- <u>Amortization</u>

- So amortized/avg case constant time

So if have enough space - just have large hash table

Save time, waste space

6.006 Lecture

3/1

Today: 3 new ideas

How to convey new items

↳ via Hashing

Dynamic Dictionaries

So far n items in m-size table

Now: arbitrary seq of insert, delete, find n:

How big table should we set up

too { ~~slow~~ small → load high - ops slow
large → waste space }

Want $m = \Theta(n)$ all times

Sol: resize
- start w/ table m
- make table bigger/smaller

(2)

Ignore hash size for simplicity

<u>Approach 1</u>

When $n > m$ $m \leftarrow m+1$

But each insert, we need to rebuild

$\Theta(1+2+\ldots+n) = \Theta(n^2)$

<u>Approach 2</u>

When $n \geq 2m$

Costly inserts in some cases

inserts 2^i for all i

$\Theta(1+2+4+\cdots+n) = \Theta(n)$

All other inserts take $O(1)$

↳ From the linked list will be ~1 item deep

keeps m a power of 2

(3)

Amarortized operation

If a seq of n operations takes time T
Then each op has amarized cost ~~[illegible]~~ $\frac{T}{n}$

Some ops are very slow
$\Theta(n)$ for insertion that causes last resize

But fast amortized cost per Op $O(1)$

Only case about total run time

Deletions

Don't rebuild just when $n < m$
↳ No $O(n^2)$

Rebuild $n < \frac{m}{2}$

But arbitrary add + deletions
- if you've just rebuilt $m = n$
- then if grow - will rebuild again

(4) need at least m items

Amarortized insert cost $O\left(\frac{2m}{m}\right) = O(1)$

So must have at $\frac{m}{2}$ items till __

(missed)

Summary

Arbitrary seq of insert / delete / find
$O(1)$ amortized time per op
* Decide when to rebuild carefully

OnLine Algorithm

- So before it was a process that took in all your data

Before

(5)

But if you ~~knew~~ had multiple inputs and knew them
(angle seeing in the future)

So make diff picks since can see future decision

So Ignorance vs Omniscience

But w/ timing sometimes we can do as well as the omnicent

Open Address

Remember chaining

Diff technique here

No linked list

- if bucket occupied, find other bucket $m \geq n$

For insert: probe a seq of buckets till find empty one

h specifies probe seq for key x

Technically h(x) seq visits all buckets

wants a mapping

(6)

Example

if occupied

go to anoter address $h(k,2)$

repeat check if occupied

Ops

Insert - ~~search~~ probe till find empty bucket

Search Probe till find item (sucess)

Or find empty bucket (failure)

└ thats where it would have been

Delete Probe to find it, remove item, leaving empty bucket

But

It doesn't work!

Insert X

Insert Y

└ passes through X

store Y in next slot

②

Delete x

Search y

ⓧ Returns $\emptyset$ - can't find y
but it is in table

<u>Instead</u>

Mark as deleted
└ RIP tombstone

If see x is deleted
└ Return not found

Insert z may hit x bucket
└ z can overwrite x since x deleted

<u>Linear Probing</u>

$h(k,i) \triangleq h'(k) + i$ for ordinary hash h'

But if lots of buckets are full
└ creates "clusters"

Big clusters hit by new items - gets longer

So bigger clusters get bigger

Double hashing

Totally ordinary $f(k)$, $g(k)$

Probe seq $h(k,i) \triangleq f(k) + i\, g(k) \bmod m$

If $g(k)$ rel prime to m

then probe seq for k can hit all buckets

(math)

Performance

hard to answer

$h(k,i)$ as before → then we can make UHA

- random permutation of $[1, \ldots, m]$

Note UHA ≠ SUHA

(9) UHA

~~Suppose~~ what is prob 1st probe sucessful

$$\text{Prob}\left(\frac{\text{empty}}{\text{all}}\right) = \frac{m-n}{m} \triangleq p$$

↑declared

Why? from UHA - probe seq must be random permutation

But if 1st probe fails, P(second probe sucessful)

$$\frac{m-n}{m-1} \geq \frac{m-n}{m} = p$$

Every trial succeeds w/ prob $\geq p$

Expected

(missed)

(10)

Open Addressing vs Chaining

Open addressing skips linked list

But as $\alpha \to 1$ the performance really deteriorates

What if

How do humans think?

How do they retrieve info from brain?

But our memory decays w/ time

Is our brain doing open addressing

Universal Hashing

Goal: Get rid of SUHA

Create a family of hash functions H

When you start picking at random $h \in H$

Unless you are unlucky - few collisions

- Adversary does not use your hash so can't force many collisions

(11)

Def: Universal hash fn

$$P[h(x) = h(y)] = \frac{1}{m}$$

Theorm UHF produces few expected collisions

$$E[\text{collisions w/ } x] = E[\#\text{ of } y \text{ s.t. } h(x) = h(y)]$$

$$= E[\sum_y 1_{h(x)=h(y)}]$$

$$= \sum_y E[1_{h(x)=h(y)}] \quad = \text{linearlity}$$

(missed)

Does this universal hashing family exists

Proof on slides

12

Probabilism

Crucial because

1. Adversary wants to harm you
2. To harm you, he must know what you'll be doing
3. But if you don't know what you are doing!
   - Since flip a coin

   ← Super Murphys Law
4. SM's law All sufficient complex systems are adversal

Crypto → Aversalial Computation

You pick h in Hash Family H

Adversary knows H

but not $h \in H$

Adversary Can pick the seq of keys you hash

A can learn if a collision was produced
- from electricity usage

Can learn values $h(k_1), h(k_2) \ldots h(k_i)$

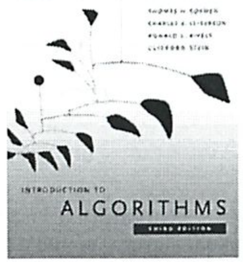
(13)

Adversary can choose $h(k_{i+1})$ adaptively

And thus crypto never sleeps

Think about 6.875

6.006- *Introduction to Algorithms*



Lecture 7

Prof. Silvio Micali

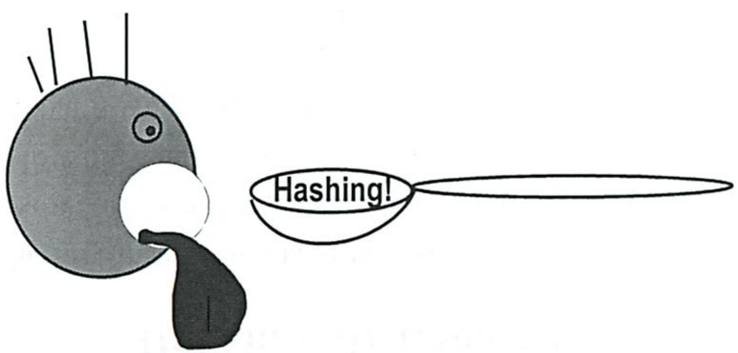
Plan for Today:
3 new Ideas, 2 of which GREAT!

Congratulations!

Sit down
Focus
Enjoy

Vote at the end...

How to convey these
new cool iDEAS?



Idea 1

VIA: DYNAMIC DICTIONARIES

Dynamic Dictionaries

- **So far:** Insert n items in m-size table
- **Now:** *arbitrary* sequence of insert, delete, find n?
- How big a table should we set up?
- What if we guess wrong?

too small → load high, operations slow
too large → high initialization cost, wasted space

Wanted: $m=\Theta(n)$ at all times

Solution: Resize

- Start with small constant m
- When table too full, make it bigger
- When table too empty, make it smaller

How?

Build a whole new hash table and reinsert items
(Recompute all hashes, Recreate new linked lists)

Time to rebuild: NewSize + #hashes × HashTime

(For simplicity: ignore HashTime)

When to resize?

Approach 1: whenever $n > m$, $m \leftarrow m+1$
Sequence of n inserts:

- Each insert increases n past m causing rebuild
- Total work: $\Theta(1 + 2 + \ldots + n) = \Theta(n^2)$

Approach 2: Whenever $n \geq 2m$

- ***Costly inserts***: insert 2^i for all i:

These cost: $\Theta(1 + 2 + 4 + \ldots + n) = \Theta(n)$

- All other inserts take $O(1)$ time – ***why?***
- Inserting n items takes $O(n)$ time
- Keeps m a power of 2

Amortized Analysis

- If a sequence of n operations takes time T, then each operation has amortized cost T/n
- Some ops are very slow: $\Theta(n)$ for insertion that causes last resize
- But fast amortized cost per operation: $O(1)$
- Often only care about total runtime, so low amortized time is great

Deletions?

- Rebuild table to new size when $n < m$? No: $O(n^2)$
- Rebuild when $n < \frac{m}{2}$

Arbitrary Insertions + Deletions?

Suppose "just rebuilt": $m = n$

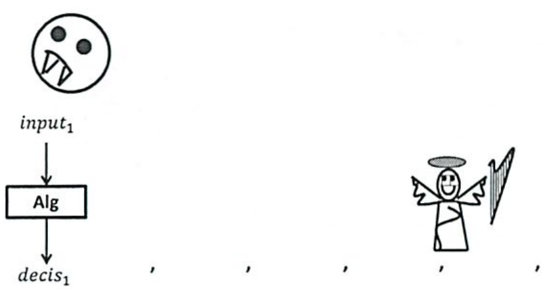
- Next rebuild a "grow" ⇒ at least m more inserts before growing table
  Amortized **insert** cost O(2m / m)) = O(1)

- Next rebuild a "shrink" ⇒ at least m/2 more deletes before shrinking
  Amortized **delete** cost O(m/2 / (m/2)) = O(1)

Summary

- Arbitrary sequence of insert/delete/find
- O(1) amortized time per operation

Welcome to:
On-Line Algorithms!

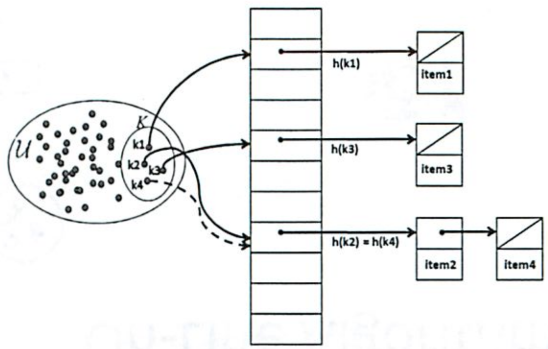


Ignorance vs. Omniscience

Idea 2

OPEN ADDRESSING

Recall Chaining…



$\mathcal{U}$: *universe of all possible keys-huge set*

K : *actual keys-small set, but not known when designing data structure*

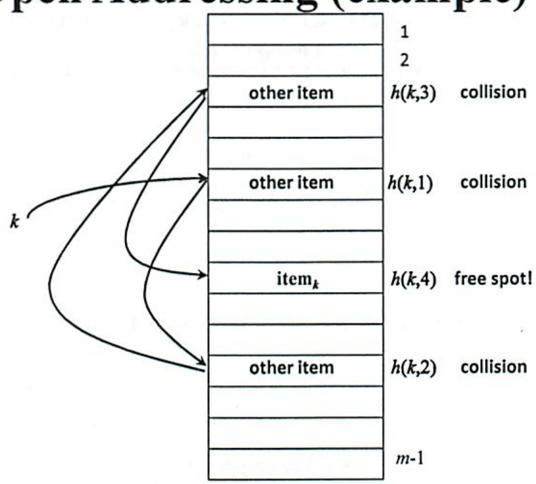
Different technique for dealing with collisions:

No linked lists: if bucket occupied, find other bucket (need m≥n)

- For insert: probe a sequence of buckets until find empty one!
- h specifies probe sequence for key x
  - Ideally, h(x) sequence "visits all buckets"
  - Technically, h: U × [1..m] → [1..m]

Universe of keys   Probe number   Bucket

Open Addressing (example)



Operations

Insert:

- Probe till find empty bucket, put item there

Search:

- Probe till find item (return with success)
- Or find empty bucket (return with failure)
  - Because if item inserted, would use that empty bucket

Delete:

- Probe till find item
- ~~Remove, leaving empty bucket~~

Problem with Deletion

Consider the following sequence:

- Insert x
- Insert y
  - suppose probe sequence for y passes x bucket
  - store y elsewhere
- Delete x (leaving hole)
- Search for y
  - Probe sequence hits x bucket
  - Bucket now empty
  - Conclude y not in table (else y would be there)

Solution for deletion

- When delete x
  - Leave it in bucket, but mark it deleted
- Future search for x sees x is deleted
  - Returns "x not found"
- "Insert z" probes may hit x bucket
  - Since x is deleted, overwrite with z

  (So keeping deleted items doesn't waste space)

What probe sequence?

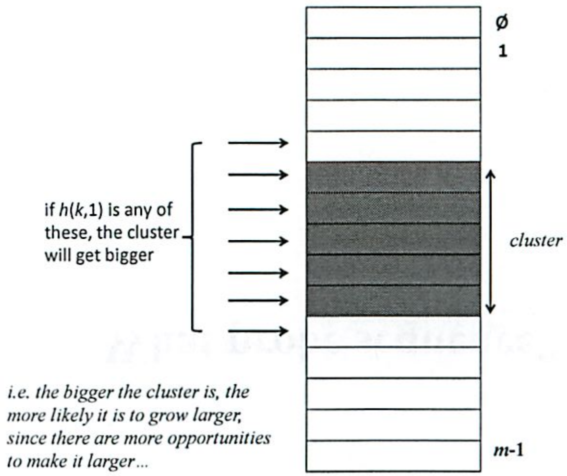
Linear probing

$h(k,i) \triangleq h'(k) + i$ for ordinary hash h'

Problem: creates "clusters",

i.e. sequences of full buckets

- exactly like parking
- Big clusters are hit by lots of new items
- They get put at end of cluster
- Big cluster gets bigger: "rich get richer" phenomenon



- E.g., $0.1 < \alpha < 0.99$, cluster size $\Theta(\log n)$
- Wrecks our constant-time operations

Double Hashing

- Two ordinary hash functions f(k), g(k)
- Probe sequence $h(k,i) \triangleq f(k) + i \cdot g(k) \bmod m$
- If g(k) always relatively prime to m, E.g., $m=2^r$ g(k) odd
  Then probe sequence for k can hit all buckets

Proof: The same bucket is hit twice if for some i,j:

$f(k) + i \cdot g(k) = f(k) + j \cdot g(k) \bmod m$
→ $i \cdot g(k) = j \cdot g(k) \pmod m$
→ $(i-j) \cdot g(k) = 0 \pmod m$
→ m and g(k) not relatively prime
(otherwise m should divide i-j, which is not possible for i, j<m)

Performance of Open Addressing

- Operation time is length of probe sequence
- How long is it?
- In general, hard to answer.
- If h(k,i) **as before**, then we "can" make the
  Uniform Hashing Assumption (UHA):
  - Probe sequence=h(k,1) h(k,2) … h(k,m) is a uniform random permutation of [1..m]

Note: this is different to the simple uniform hashing assumption (SUHA))

Analysis under UHA

Suppose:

- a size-m table contains n items
- we are using open addressing
- we are about to insert new item

Q: Probability **first** prob successful?

$$Prob\left(\frac{empty\ buckets}{all\ buckets}\right) = \frac{m-n}{m} \triangleq p$$

Why? **From UHA, probe sequence random permutation**
Hence, first position probed randomly
m-n out of the m slots are unoccupied

Analysis (II)

Q: If first probe unsuccessful, probability second prob successful?

$$\frac{m-n}{m-1} \geq \frac{m-n}{m} = p$$

Why?

- From UHA, probe sequence random permutation
- Hence, first probed slot is random; the second probed slot is random among the remaining slots, etc.
- Since first probe unsuccessful, it probed an occupied slot
- Hence, the second probe is choosing uniformly from m-1 slots, among which m-n are still clean

Analysis (III)

- If first two probes unsuccessful, probability third prob successful?

$$\frac{m-n}{m-2} \geq \frac{m-n}{m} = p$$

- …

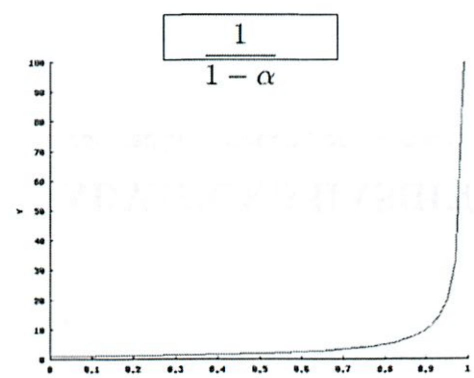
➔ every trial succeeds with probability ≥p

expected number of probes till success? $\leq \frac{1}{p} = \frac{1}{1-\alpha}$

e.g. if α=90%, expected number of probes is at most 10

Open Addressing vs. Chaining

- Open addressing skips linked lists
  - Saves space (of list pointers)
  - Better locality of reference
    - Array concentrated in m space
    - So fewer main-memory accesses bring it to cache
    - Linked list can wander all over memory
- Open addressing sensitive to load α
  - As α → 1, access time shoots up



What IF?

ADVANCED HASHING ?

covered in recitation (for those who care)

Idea 3

VIA UNIVERSAL HASHING

Goal

Get rid of simple uniform hashing assumption

- Create a family of hash functions H
- When you start, pick at random $h \in H$
- Unless you are unlucky, few collisions
  ~Adversary doesn't know what hash you will use
  So cannot pick keys that collide too much

DEF: Universal Hash Family

...is a family (set) of hashfunctions such that, for any keys x and y, if you choose a random h from the family,

$$Pr[h(x) = h(y)] = 1/m$$

Thm: UHF produces few expected collisions

Proof:

$$\begin{aligned} E[collisions\ with\ x] &= E[number\ of\ y\ s.t.\ h(x) = h(y)] \\ &= E[\textstyle\sum_y 1_{h(x)=h(y)}] \\ &= \textstyle\sum_y E[1_{h(x)=h(y)}]\ (linearity\ of\ E) \\ &= \textstyle\sum_y \Pr[\,h(x) = h(y)] \\ &= n/m \end{aligned}$$

THM: ∃Universal Hashing Families!

Proof:

- Suppose table size= p prime
- Define $h_{ab}(x) = a \cdot x + b\ (mod\ p)$
- If a and b are random elements in $\{0, \dots, p-1\}$, then $h_{ab}(x)$ is a UHF
- $mod\ p$ is a field, so you can divide/substract as well
- Pick two keys x and y. What is the probability (over the choice of a, b) that the hashes of x and y collide?
- Must be $a \cdot x + b = q\ (mod\ p)$ and $a \cdot y + b = q\ (mod\ p)$, for some q in $\{0, \dots, p-1\}$
- For fixed q, this is a linear system in a, b
- Two variables, two equations, Unique solution: that is, unique h_{ab} makes this happen
- Probability of choosing this h_{ab} is $1/p^2$
- Collision if $h_{ab}(x) = h_{ab}(y) = q$ for some q
- There are p possible values for q, hence overall probability of collision= $p/p^2 = 1/p = 1/m$

Welcome to Probabilism!

Crucial because:

1. The Adversary wants to harm you

2. To harm you he must know what you'll be doing

3. He cannot know if **you yourself** do not know!

And

4. **SM**'s Law: All sufficient complex systems are adversarial!

Cryptography

Secret writing → *Adversarial Computation*

You pick h in a hash family H (but not which h you picked)

Adversary knows H (but not which $h \in H$ you picked!)

Adversary picks the sequence of keys you must hash

Adversary learns when he has caused a **collision**

Adversary learns the **values** $h(k_1)$, $h(k_2)$, ..., $h(k_i)$

Adversary can choose $h(k_{i+1})$ **adaptively!**

And yet...

"Cryptographers never sleep"

SM

Happy 6:006 ⇒ Happy 6.875!

Credits

Teenagegirlsvslife.blogspot.com
Goldenstateofmind.com
SMgraphics.home

Vote!

Next Week: Sorting

Summary

- Hashing maps a large universe to a small range
- But avoids collisions
- Result:
  - Fast dictionary data structure
  - Fingerprints to save comparison time

- Next week: sorting

Better? Perfect Hashing!

- Hash table with zero collisions
- So don't need linked lists
- Can't guarantee for arbitrary keys
- But if you know keys in advance, can quickly find a hash function that works
  - E.g. for a fixed dictionary

NOT COVERED IN CLASS

Fingerprinting

- File backup service
  - Major cost in time and money: bandwidth
- How decide whether a file has changed?
  - And thus needs new backup
- Send whole file?
  - Too expensive
- Send hash of file (treating file as big number)
  - Only send file if hash differs
  - Might make a mistake, if hashsame

What signature?

- File x and backup y, length n bits
- Treat as n-bit numbers
- Pick random prime number p in [2..n]
- Hash/compare x (mod p) vs. y (mod p)
  - Send log n bits
- False negative if
  - x and y different
  - but x (mod p) = y (mod p)
  - i.e. (x-y) (mod p) = 0
  - i.e. p is a factor of x-y

What are the odds?

- How many prime factors does x-y have?
  - It's an n-bit number
  - It's the produce of its factors $p_1 .. p_k$
  - Each $p_i \geq 2$
  - So $(x-y) = p_1 p_2 .. p_k \geq 2^k$
  - So $k \leq \log_2 n$ prime factors
- How many primes in range [1..n] ?
  - Prime number theorem says about n/ln n
  - So, Pr[pick wrong factor] = (log n)/(n/ log n)
  - For better safety, pick bigger prime

Randomized Algorithms

- Hashing/Fingerprinting make random choices
- Then you prove they probably work
- Prevent adversary from giving you a bad input
- Lot of applications in algorithms design
  - Take 6.856 some day

Another Approach

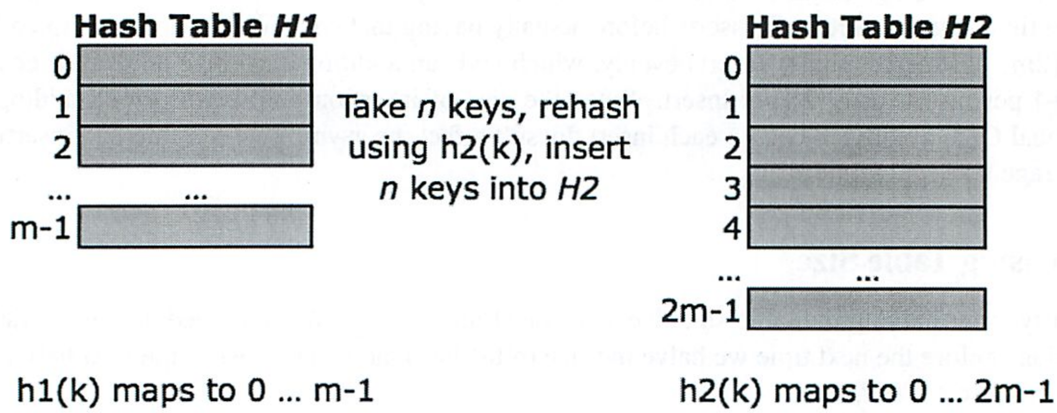
- Algorithm
  - Keep m a power of 2 (for faster computation)
  - Grow (double m) when $n \geq m$
  - Shrink (halve m) when $n \leq m/4$
- Analysis
  - Just after rebuild: $n=m/2$
  - Next rebuild a grow → at least m/2 more inserts
    - Amortized cost $O(2m / (m/2)) = O(1)$
  - Next rebuild a shrink → at least m/4 more deletes
    - Amortized cost $O(m/2 / (m/4)) = O(1)$

Resizing Hash Tables

Hash tables perform well if the number of elements in the table remain proportional to the size of the table. If we know exactly how many inserts/deletes are going to be performed on a table, we would be able to set the table size appropriately at initialization. However, it is often the case that we won't know what series of operations will be performed on a table. We must have a strategy to deal a various number of elements in the hash table while preserving an average $O(1)$ access, insertion, and removal operations.

To restrict the load balance so that it does not get too large (slow search, insert, delete) or too small (waste of memory), we will increase the size of the hash table if it gets too full and decrease the size of the hash table if it gets too empty.

Resizing a hash table consists of choosing a new hash function to map to the new size, creating a hash table of the new size, iterating through the elements of the old table, and inserting them into the new table.



Consider a hash table that resolves collisions using the chaining method. We will double the size of the hash table whenever we make an insert operation that results in the load balance exceeding 1, i.e. $n > m$. We will halve the size of the hash table whenever we make a delete operation that results in the load balance falling beneath $\frac{1}{4}$, i.e. $n < \frac{m}{4}$. In the next sections, we will analyze this approach and show that the average runtime of each insertion and deletion is still $O(1)$, even factoring in the time it takes to resize the table.

Increasing Table Size

After doubling the table size due to an insert, $n = \frac{m}{2}$ and the load balance is $\frac{1}{2}$.We will need at least $\frac{m}{2}$ insert operations before the next time we double the size of the hash table. The next resizing will take $O(2m)$ time, as that's how long it takes to create a table of size $2m$.

	0.5m insertions before resize				resize
Operation	insert	insert	...	insert	insert + resize
Runtime	O(1)	O(1)	...	O(1)	O(2m)

Redistribute O(2m) resize cost over 0.5m insertions

	0.5m insertions before resize				resize
Operation	insert	insert	...	insert	insert + resize
Runtime	O(1)	O(1)	...	O(1)	O(1)
Amortized Cost	O(2m/0.5m)	O(2m/0.5m)	...	O(2m/0.5m)	-

On average, since the number of elements is proportional to the size of the table at all times, each of the $\frac{m}{2}$ inserts before resizing will still take $O(1)$ time. The last insert will take $O(2m)$ time as we need to factor in the time it takes to resize the table. We can use amortized analysis to argue that the average runtime of all the insertions is $O(1)$. The last insert before resizing costs $O(2m)$ time, but we needed $\frac{m}{2}$ inserts before actually paying that cost. We can imagine spreading the $O(2m)$ cost across the $\frac{m}{2}$ inserts evenly, which adds an additional average amortized cost of $O(\frac{2m}{0.5m})$ per insert, or $O(1)$ per insert. Since the cost of insertion before was $O(1)$, adding an additional $O(1)$ amortized cost to each insert doesn't affect the asymptotic runtime and insertions on average take $O(1)$ time still.

Decreasing Table Size

Similarly, after halving the table size due to an deletion, $n = \frac{m}{2}$. We will need at least $\frac{m}{4}$ delete operations before the next time we halve the size of the hash table. The cost of the next halving is $O(\frac{m}{2})$ to make a size $\frac{m}{2}$ table.

The $\frac{m}{4}$ deletes take $O(1)$ time and the resizing cost of $O(\frac{m}{2})$ can be split evenly across those $\frac{m}{4}$ deletes. Each deletion has an additional average amortized cost of $O(\frac{0.5m}{0.25m})$ or $O(1)$. This results in maintaining the $O(1)$ average cost per deletion.

Performance of Open Addressing

Recall that searching, inserting, and deleting an element using open addressing required a probe sequence (e.g. linear probing, quadratic probing, double hashing). To analyze the performance of operations in open addressing, we must determine on average how many probes does it take before we execute the operation. Before, we made the **simple uniform hashing assumption (SUHA)**, which meant a hash function mapped to any slot from 0 to $m - 1$ with equal probability. Now, we make the **uniform hashing assumption (UHA)**, which is a slight extension from SUHA. UHA assumes that the probe sequence is a random permutation of the slots 0 to $m - 1$. In other words, each probe looks likes we're examining a random slot that we havent examined before.

If the table has load balance α, that means there is a $p = 1 - \alpha$ probability that the first probe will find an empty slot under UHA. If the first probe is a collision, note that the probability that the second probe will find an empty slot is greater than p, since there are an equal number of empty slots that we could insert in, but were choosing randomly from a pool of fewer slots. In general, after each collision, there is a probability of at least p that we will probe into an empty slot.

Using principles of probability, if there is exactly a probability of p that we will find an empty slot at each probe, then we expect to probe $\frac{1}{p}$ times before we succeed. For example, if $p = \frac{1}{4}$, we expect to probe 4 times before we find an empty slot. Since in our case, our probability of success is actually increasing after each probe, $\frac{1}{p}$ is a high estimate on how many times we probe before we succeed. Since $p = 1 - \alpha$, we expect to probe at most $\frac{1}{p}$ times. Looking at the behavior of the $\frac{1}{1-\alpha}$ graph, it is clear that with open addressing, performance is fairly good until α approaches too close to 1.

Universal Hashing

With a fixed hashing function, an adversary could select a series of keys to insert into the hash table that all collide, giving the hash table worst case performance. Universal hashing is the idea that we select the hash function randomly from a group of hash functions. This means an adversary cannot choose keys that he knows will give worst case performance anymore, since the adversary doesn't even know what hash function will be chosen for the table. If we form the group of hash functions carefully, we can assure that the expected time for each operations is $O(1)$, even if there is an adversary who is trying to achieve worst case performance.

For universal hashing to work, the group of hash functions H must be **universal**. This means that for each pair of distinct keys k, l, in the universe of keys, the number of hash functions in the group for which $h(k) = h(l)$ is at most $\frac{|H|}{m}$. This means, for each pair of distinct keys, the chances of picking a hash function in which they collide is at most $\frac{1}{m}$, which is the same probability given by the simple uniform hashing assumption.

6.006 Recitation

3/2

Class experimental this semester

100 ~~more~~ more students than last year

Today: PS2 #4

- lots of ways to approach

Count # of bits that are 1 in an ~~binary~~ integer

- could just increment through the digits

```
while num > 0
    if num % 2 = 1
        count += 1
    num /= 2
      ↑ shift left
```

So we are inputting a regular intiger

Can we rewrite it in a way that is far faster

②

Test will be 20% of grade
- if correct on all test cases

Will take a minute or two

If faster - get speed points
- so wait to see the results
- is posible to do in < minute

Words are 64 bits
4 bytes

We could bitshift >> instead of /2
No logic, so fast
C auto optimizes
But not Python - since does not know will be an int
20% speedup
* Must multiply or divide by ~~the~~ power of 2

Can get rid of , num % 2 == 1 ,
→ do we need? No

Would it make a difference → no

③

Anything w/ conditional is slow
- Since pipelining
- when conditional, it ~~xxxx~~ guesses what dir
- if wrong, needs to flush pipeline

So instead cont += num % 2

Recursion is <u>very</u> slow

Do some languages optimize ^tail recursion

Not python

% is same as division

if power of 2 - use bit operators

So mod 2 = AND 1

Gave us another 20% ~6

So thats as fast as we can get cleaning up this algorithm

(4)

So we are going to build a lookup table

Not for entire function, of course

But for a core set of common operations

So needs to be just right - not too big or small

So precompute 1 → 256

So then process by byte

And shift by 8, instead of 1

precompute

```
lookup = []
for i in range(256):
    lookup.append(count_bits(i))
```

Then

```
count += lookup[num & 255]
                  ^AND
num >>= 8
```

⑤

Table [creation] does not count in analysis

Since run it once, before timing

This is 2-3 times faster!

What if look up table was bigger?

8 bits
16 bits → Do it dividing the word size

↳ 65535

then shift 16

⊗ Takes longer since lookup

- lookup is constant time
- but constant time is getting better

Computers do use virtual memory to HDD

And there are caches underneath main ram [multiple]

(e) L1 cache 32 kb - might be ~ 8kb usable
1st table 1 kb
2nd table 256 kb

But if optimizing multiplication - larger table still might be better

Problem Set 2

This problem set is due **Wednesday, March 7** at **11:59PM.**

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice and true/false questions, no explanations are necessary: your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to correct solutions which are described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **Del or no del? (35 points, 5 points per part)**

   Consider the following correct Python implementation for deleting a node from a binary search tree. This function is analogous to the `delete` method of the `BSTNode` class found on the website, except that it assumes that all `BSTNode` instances have a parent pointer. (Since the implementation on the website does not include parent pointers, you will not be able to test this code by replacing the `delete` method in that class.)

   This `delete` function takes a node, `self`, and a value, `val`. It deletes the node with that value from the subtree rooted at `self`, if it exists and if the tree has at least one other node. The function returns `True` if some node was deleted.

   Assume each node has five properties: its `val`, `count`, `left`, `right`, and `parent`. The `left`, `right`, and `parent` pointers are either other instances of this class or `None`. Also, assume that the `search` method is implemented exactly as in the `BSTNode` class.

↓ does not have node

```
1 def delete(self, val):
2     # Find the node to delete.
3     node = self.search(val)
4     if node.val != val:
5         return False
6
7     # If there were multiple occurrences of this value, we're done.
8     node.count -= 1
9     if node.count > 0:
10        return True
11
12    if node.right is None:
13        if node.left is None:
14            # This node is a leaf. Delete its reference from its parent.
15            if node.parent is not None:
16                if node.parent.left == node:
17                    node.parent.left = None
18                else:
19                    node.parent.right = None
20                return True
21            else:
22                # We are the only node. Deletion is not allowed.
23                return False
24        else:
25            # Move the old left child to our place.
26            node.val = node.left.val
27            node.count = node.left.count
28            node.right = node.left.right
29            node.right.parent = node
30            node.left = node.left.left
31            node.left.parent = node
32            return True
33    else:
34        # We have a right child. Replace this node with its successor
35        # in the right subtree.
36        next = node.right.search(val)
37        if next is not None:
38            node.val = next.val
39            node.count = next.count
40        node.right.delete(next.val)
41        return True
```

has a left child (lines 27–30)

has a right child (lines 37–39)

Correction ← next.count=1 line 40 (lines 40, 41 renumbered by hand as 41, 42)

Answer the following questions with `True` or `False`.

a. The code is correct if lines 25 – 32 are replaced with the lines

```
25          # Move the old left child to our place.
26          node.left.parent = node.parent
27          if node.parent is not None:
28              if node.parent.left == node:
29                  node.parent.left = node.left
30              else:
31                  node.parent.right = node.left
32          return True
```

b. The code is correct if line 36 is replaced with

```
36      next = node.right
```

c. The code is correct if line 37 is removed (and lines 38 – 39 are unindented).

d. The code is correct if all instances of `left` and `right` are interchanged.

e. Lines 34 – 41 can be replaced with the lines

```
34      # We have a right child. Replace this node with its successor
35      # in the right subtree.
36      next = node.right.search(val)
37      if next.right is not None:
38          next.right.parent = next.parent
39      if next.parent.left == next:
40          next.parent.left = next.right
41      else:
42          next.parent.right = next.right
43      return True
```

f. The code is correct if line ~~40~~ 41 (Correction) is moved to just before line 37.

g. The code is correct if lines 40 and 41 are combined into

```
40      return node.right.delete(next.val)
```

Solution Format:

You should answer this problem with a boolean value for each part. For example, if you thought the answer to part y) was `True` and the answer to part z) was `False`, then your answer should be:

```
answer_for_problem_1_part_y = True
answer_for_problem_1_part_z = False
```

2. **Binary search tree sort (20 points)**

Consider the following code for a sorting algorithm. Here, the `BST` class is an implementation of a self-balancing binary search tree. This class supports the `insert`, `get_min`, and `delete` operations in $O(\log n)$ time, where n is the number of elements in the tree.

```
def bst_sort(list):
    bst = BST()
    for val in list:
        bst.insert(val)
    ans = []
    for i in range(len(list)):
        min = bst.get_min()
        ans.append(min)
        bst.delete(min)
    return ans
```

a. **(5 points)** This function sorts the list: `True` or `False`?

b. **(5 points)** On a list of n elements, the runtime of this algorithm is:

   1. $O(n)$
   2. $O(n \log n)$
   3. $O(n \log^2 n)$
   4. $O(n^2)$
   5. $O(n^2 \log n)$
   6. $O(n^2 \log^2 n)$

c. **(10 points)** Assuming that (comparison) sort is impossible in better than $\Theta(n \log n)$, give a short argument that it is impossible to construct a data structure which stores arbitrary ordered values and supports `insert`, `get_min` and `delete`, each in $o(\log n)$.

Solution Format:

Your answer for part a) should be a boolean. Your answer for part b) should be an integer between 1 and 6, and your answer for part c) should be a (short) string.

3. **An awkward sort of party (20 points)**

   There are n people who attend a party, labeled 1 through n. Person i arrives at time a_i and departs at time d_i. The $2n$ arrival / departure times are all distinct.

   None of the partygoers knew each other before the event. Afterwards, each person goes on Twitter and follows the people who were there when they arrived at the party, but who left before they did.

   Find an efficient algorithm to determine the total number of new Twitter followings formed, given the the n pairs of the arrival and departure times of each person. Prove that your algorithm is correct and find its running time. For full credit, your algorithm should run in $O(n \log n)$ time.

   Solution Format:

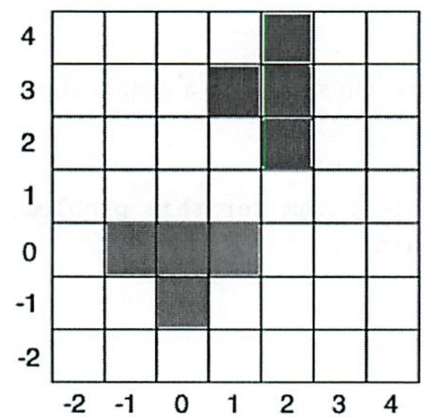
   Your answer for this problem should be a string, such as:

```
answer_to_problem_3 = """
I have a beautiful algorithm for this problem, but this tweet is not
long enough to contain it.
"""
```

4. **The thinly-veiled ntris problem (50 points)**

Two biologists have independently documented the proteins found in two strains of bacteria, *E. foo* and *E. bar*. Each protein is a *polyomino*: a two-dimensional shape formed by attaching a number of unit squares along their edges. Two proteins are the same if one polyomino can be transformed into the other by a rotation and translation.

Each scientist represents a protein as a list of ordered pairs of integers, one pair for each unit square in the protein. Each pair represents the coordinates of the center of its unit square. For example, the T-protein (which looks much like the T piece in Tetris) might be represented by the list `[(0,0),(1,0),(-1,0),(0,-1)]` or by the list `[(2,3),(2,4),(2,2),(1,3)]`:



Two representations of the T-protein.

As a computer scientist working with the biology department, your job is to determine the number of proteins in common between the two strains of bacteria. Write a function `num_proteins_in_common` that efficiently computes the number of proteins in common, given two lists of proteins. You may assume that the proteins in each list are distinct. However, you may not assume a bound on the number of proteins or on the number of squares in a protein.

We have attached some code to help you get started with this problem. Specifically, we have provided three functions for your use:

- a `translate` function that translates a polyomino by a fixed offset
- a `rotate` function that rotates a polyomino by a quarter-turn counterclockwise
- and a `compare` function that determines if two polyominoes are equivalent after rotations and translations

(To see more examples of polyominoes, you may want to visit `ntris.mit.edu`. However, a high score will not get you any credit for this class.)

Code on Website

No parent pointer

```
class BST(object):
  class BSTNode(object):
    # This node has two pieces of data: a value and a count. The count is the
    # number of times that this value has been inserted.
    def __init__(self, val):
      self.val = val
      self.count = 1
      self.left = None
      self.right = None

    # Replaces this node's count and value with the other's. Used in delete().
    def replace_data(self, other):
      self.val = other.val
      self.count = other.count

    # search() returns the last node in this node's subtree on the path taken
    # when searching for val.
    def search(self, val):
      if self.val < val:
        if self.right is not None:
          return self.right.search(val)
      elif self.val > val:
        if self.left is not None:
          return self.left.search(val)
      return self

    # Inserts val into this node's subtree.
    def insert(self, val):
      result = self.search(val)
      if result.val < val:
        result.right = BST.BSTNode(val)
      elif result.val > val:
        result.left = BST.BSTNode(val)
      else:
        result.count += 1

    # If val is in this node's subtree, delete() removes one occurrence of val
    # from this subtree. This function may change the root of this subtree, so
    # it returns the new root.
    def delete(self, val):
      if self.val == val:
        # The only case when actual deletion happens is when count becomes 0.
        self.count -= 1
        if self.count <= 0:
          if self.right is not None:
            # In this case, we search the right subtree to find the node of
            # smallest value greater than val. We move that next node's value to
            # this node, and then delete next from the right subtree.
            next = self.right.search(val)
            self.replace_data(next)
            next.count = 1
            self.right = self.right.delete(next.val)
```

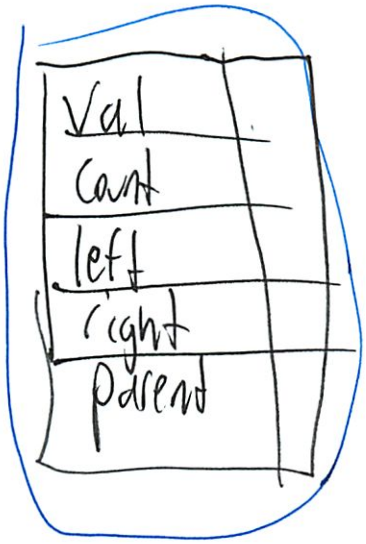
```
      else:
        # Since this node has no right subtree, we can delete it by moving
        # its left child into its position. The left child is the new root
        # of this subtree.
        return self.left
    # The easy cases: recurse on the right or left subtree and return self.
    elif self.val < val:
      if self.right is not None:
        self.right = self.right.delete(val)
    elif self.val > val:
      if self.left is not None:
        self.left = self.left.delete(val)
    return self

  # Performs an in-order traversal of the subtree rooted at this node and
  # appends the elements to the result list.
  def in_order_traversal(self, result):
    if self.left is not None:
      self.left.in_order_traversal(result)
    for i in range(self.count):
      result.append(self.val)
    if self.right is not None:
      self.right.in_order_traversal(result)

  # DO NOT BOTHER TO READ THIS CODE! Used to pretty-print small trees. Do
  # not call on large trees.
  def __str__(self):
    if self.left is None:
      if self.right is None:
        return str(self.val)
      right_strs = str(self.right).split('\n')
      left_strs = len(right_strs)*['']
    elif self.right is None:
      left_strs = str(self.left).split('\n')
      right_strs = len(left_strs)*['']
    else:
      left_strs = str(self.left).split('\n')
      right_strs = str(self.right).split('\n')
      left_rows = len(left_strs)
      right_rows = len(right_strs)
      if left_rows < right_rows:
        left_strs.extend((right_rows - left_rows)*[len(left_strs[0])*' '])
      else:
        right_strs.extend((left_rows - right_rows)*[len(right_strs[0])*' '])

    left_index = 0
    for i in range(len(left_strs[0])):
      if left_strs[0][i] != ' ':
        left_index = i
        break
    right_index = len(right_strs[0])
    for i in range(len(right_strs[0])):
```

```
        if right_strs[0][i] != ' ':
          right_index = i + 1

      top_str = len(left_strs[0])*' ' + str(self.val) + len(right_strs[0])*' '
      mid_len = len(str(self.val))
      second_str = left_index*' ' + (len(left_strs[0]) - left_index + mid_len + right_index
)*'-' + (len(right_strs[0]) - right_index)*' '
      return '\n'.join([top_str, second_str] + [left_strs[i] + mid_len*' ' + right_strs[i]
for i in range(len(left_strs))])

  def __init__(self):
    self.root = None

  def clear(self):
    self.__init__()

  def count(self, val):
    if self.root is None:
      return 0
    result = self.root.search(val)
    return result.count if result.val == val else 0

  def insert(self, val):
    if self.root is None:
      self.root = BST.BSTNode(val)
    else:
      self.root.insert(val)

  def delete(self, val):
    if self.root is None:
      return
    self.root = self.root.delete(val)

  def in_order_traversal(self):
    result = []
    if self.root is not None:
      self.root.in_order_traversal(result)
    return result

  def __str__(self):
    if self.root is None:
      return ''
    return str(self.root)
```

Doing PS2

Deleting nodes from a BST

- assumes all instances has a parent pointer



delete(node, tree, val)

- must have at least one other node

is node duplicate with val?

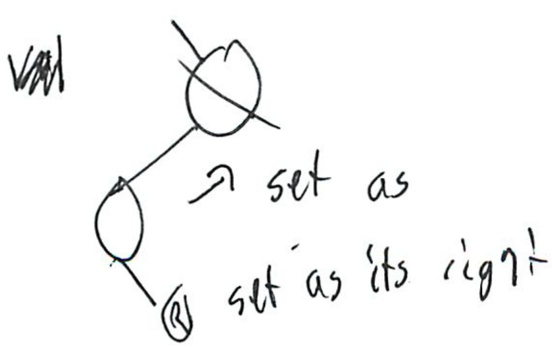
returns tree if deleted

Their delete does not take in a node...
So lets just ignore for now and answer the questions

So base code

if leaf → if parent not node

Cache it at



②

Read book's delete code

1 child - elevate

2 chidren - take right and sub it in

If z no left chidren → replace z w/ right child

If z has one (left) Child → replace it

If z has left + right

- find z's sucessor y which lies in z's right subtree + has no left child

So basically does our code follow this?

If node left is not None

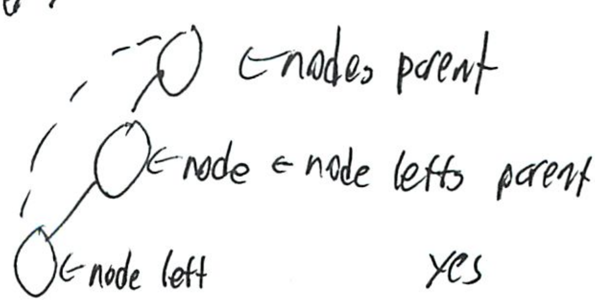
- replace

* but should only do if only child

So our current code is not correct

Are proposed fixes better?

a) Moves old left + in



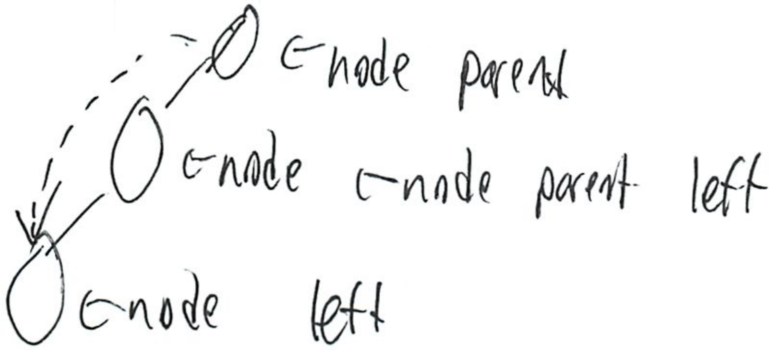
yes

if this is not None

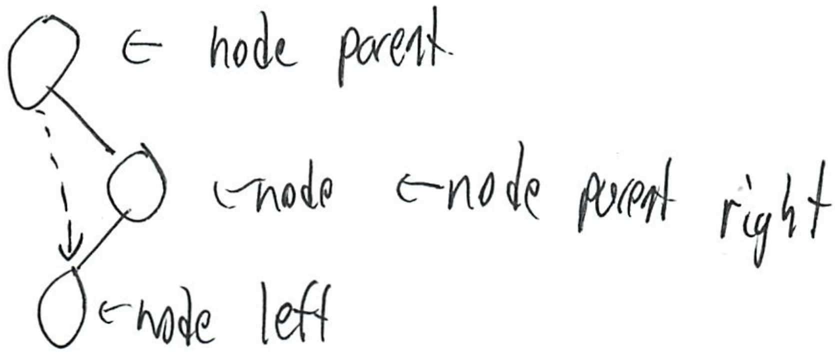
③

Why would node parent be not none?

? ie node left's new parent



Or



? So was ~~the previous~~ the original right?

So I think it is wrong when both
l and r ~~[illegible]~~ children
Since it moves left into place
when it should be on right

Ohh wait → it goes straight there if right
child!!!!

(4)

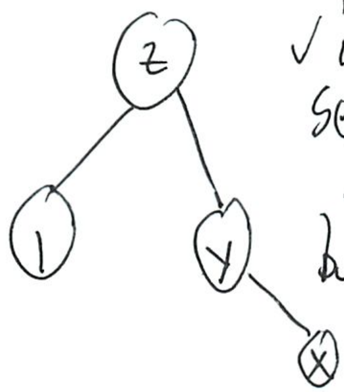
So if right node

- search for next
- set it to that

(but don't update all the parent stuff?)

- and does not check for nil left subtree...
- is that required - posted code does not do that

- Or do we set parents stuff by calling delete
  - So for delete z

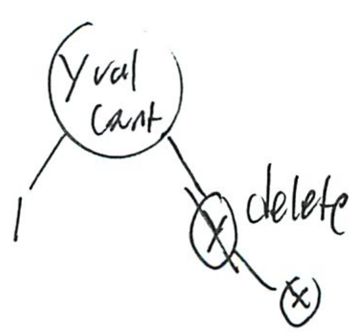


✓ is right tree
search for next → y
- has no sub child - but we never searched that

but replace cont, val in z

Supposed to, delete y

Actual

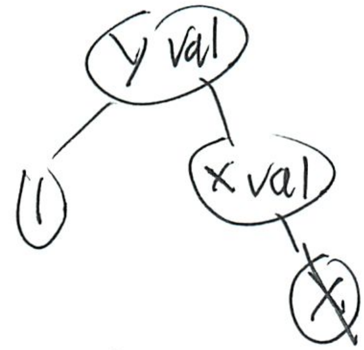


(5)

Delete y does what?

Y has ~~[illegible]~~ right child

~~[illegible]~~ Sucessor x



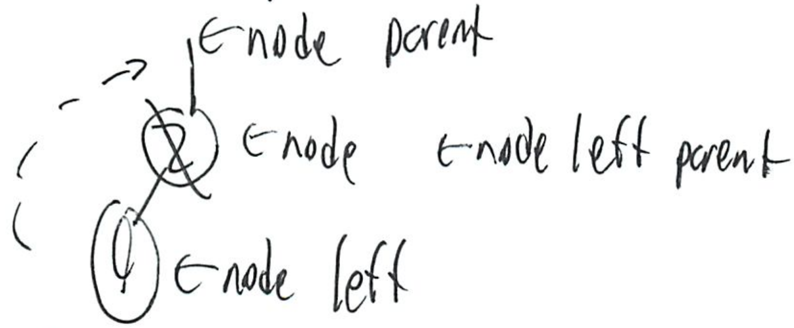
x is leaf

So set x vals parent left to none

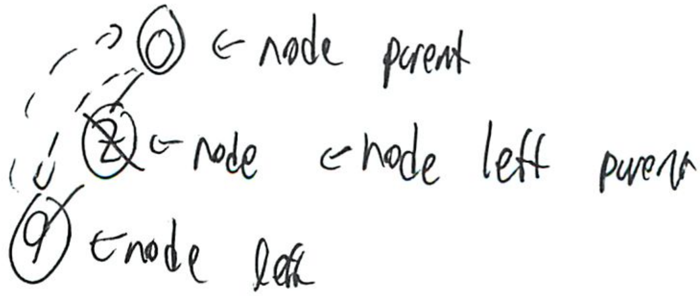
but y vals parent never updated

Oh back to qu

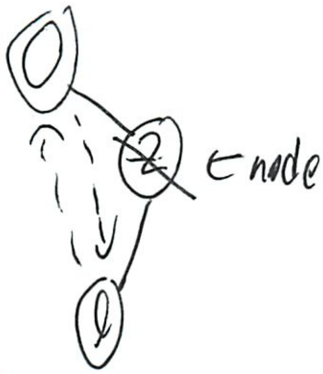
About if only left child



So if



(6)



(✓) Seems to work

b) No must serch

c) No - if no results fail

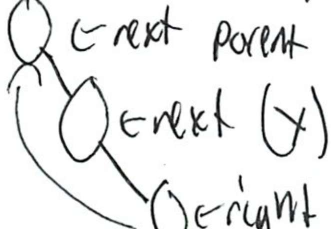
(What did we decide on if original was correct → No)

d) No - left and right matter

(But then a) still wrong
- but that section is right...)
- but that was never critical → ignore for now)
So was prob correct originally

e) A new right code

- find next
if it has a right
- set parent

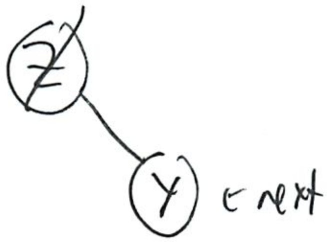


? Why are we doing that ?

(Are we replacing y by its own right child) - yes

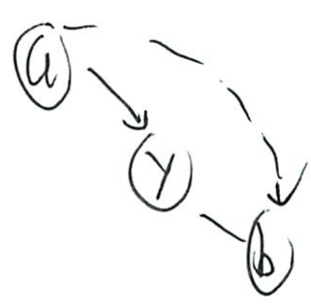
(7)

Then if y is z's right child



Then set a

Else



(8)

It doesn't seem to work - but I have no proof
- Though false has not been tried

No in both cases we are replacing y w/ its right ~~child~~ child

Say true

(Why can't I think this through
- too many test cases)

Went through ones in book (1) Passes 3rd
(2) Matches 4th

(I over complicated this)

f) No delete value before reassigning - read error

g) Will node right always delete?

I keep going back to is original correct?

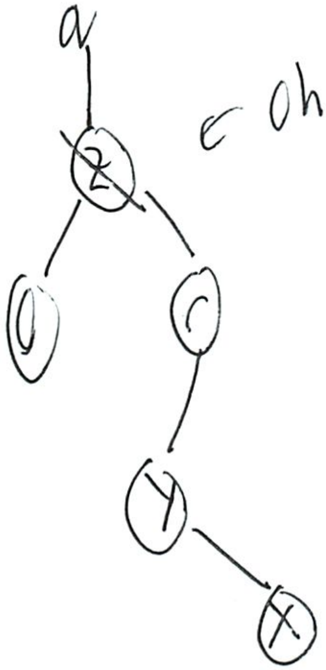
~~Yeah~~ Yeah original was correct - since in d we keep r where it is

p 297

But deleting next
Next is always found?
- could be None?

⑨

When does that happen



← oh but if z is top-most always be one?
will be higher if something on right

: so can remove if None test...

So here True
and c is revised as True

#2 Binary Tree Sort

Sorting algo in self-balancing tree
└ so log n
↑ # of els

Will this sort the list
Actually its pretty clever...

(10)

So does it work?

Yes - but not really efficient

Runtime

lg n for insert w/ balanced tree

insert: $n \lg n$

Get out

$$n(\underset{\substack{\uparrow \\ \text{tree height} \\ \lg n}}{\overset{\text{min}}{h}} + \underset{\substack{\uparrow \\ \text{delete}}}{\overset{\text{delete}}{\lg n}})$$

$n \lg n + n \cdot 2 \lg n$

So $n \lg n$

Would like to confirm that's right

I looked back in book - seems right

(ii)

c) If comparison sort is ~~[illegible]~~ $\leq \Theta(n \lg n)$

argue ~~[illegible]~~ impossible to construct a data structure which supports theses in $o(\lg n)$

? upper bound that is not asy tight

So like max

~~Basically~~ Comparison sort

$a \leq b$ and $b \leq c \rightarrow a \leq c$

for all $a, b \rightarrow$ either $a \leq b$ or $b < a$

WP: $\log_2(n!)$ - Shannon Entrophy

It's just not possible

Oh we can use $\Theta(n \log n)$

Basically inserting each item (and allowing get min) means it must be sorted - which is not possible

(12)

Book 8.1 p 193

$\Omega(n \lg n)$ is best

Uses tree - height h w/ ℓ reachable leaves

each of the $n!$ permutations appears as some leaf

we have $n! \le \ell$

A tree has $\le 2^h$ leaves

$n! \le \ell \le 2^h$

take lng

$h \ge \lg(n!)$

$= \Omega(n \lg n)$

---

3. Akward Party

n people 1 .. n

Arrive a_i depart a_i

All times distinct

(13)

Follow people who were there where arrived but ~~th~~ left before they did

i follows j if $a_i < a_j$ AND $d_i > d_j$

Find an efficient ~~a~~ algo to find the # of people in $O(n \lg n)$

So we are not trying to find the # of people who will be followed on avg (that is 6.041)
~~(Or perhaps we can~~
[for each arrival - the possible departures
I was never real good at that either

But just the for loop to check
So ~~a~~ $\Theta(n^2)$ is the naive ~~algo~~ way
Something w/ enter + leave

← I don't have Θ vs O intuition

Re read - better...

(14)

So people arrive

1 2 3 4 5

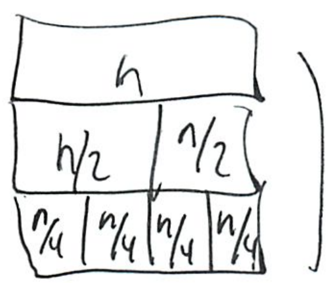
Can only follow people who were there when arrived

— 1 1,2 1,2,3 1,2,3,4

So this is upper triangular

So < half

log n means cut in half for a each



log n times

ie log 5 = 1.6

$5 \cdot 1.6 = 8$

5
2
1) = 8

Yeah cut in half each time

15

But here its ~~ex~~ avg ~~n/2~~ n

n people · $\frac{n}{2}$ avg people condition !

But leaving

Can be - everyone there same lenght of time

1 2 3 4 5

1 | 1,2 | 1,2,3 | 1,2,3,4

So thats full - needed

Everyone other way

5 4 3 2 1

No one

But we have worst case - so actually leave order does not work

So is the triangle thing $O(n \log n)$

Was 10 in above example

(16)

Yeah I get more

- $\frac{n}{2}$ is wrong

I'm so bad at this recognize pattern

Its $\sum_{i=1}^{5} (n-i) \overset{\text{Wolfram}}{=} \frac{1}{2}(n^2-n)$

? that is $\sim n^2$

But if worst case people leaving -

Then its the arrival of people is by that

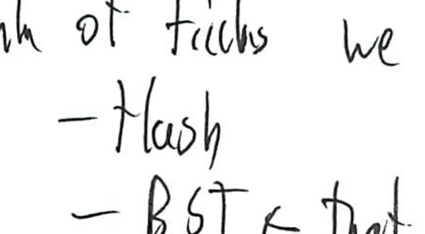
Piazza has people more care about lists

- which are not ness. sorted

Think of tricks we used

- Hash
- BST ← that air plane thing

So when someone arrives, push onto tree

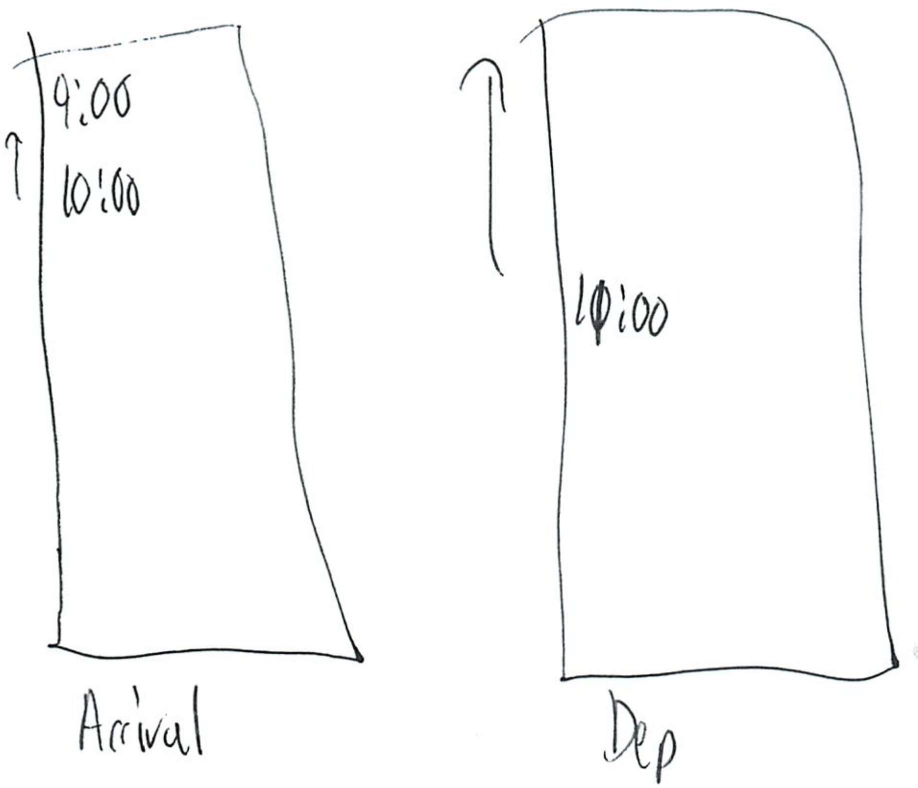


↑ copy in a list(?) of who there when arrived

Then when leave compare the list of leavers

Or this is like a 2 column match

— but no discrete values ...



So basically filled ↓ and everyone above

but that does not seem to work

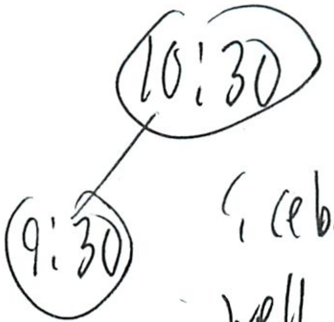
And overlap table — have to rebuild each time...

Piazza hint: Insert elements in some order

Interleave Computation w/ insertions

I was kinda thinking

So



(i.e. rebalance tree around that one
well that is only time

* Remember have array → not getting into in time

Or insert all into tree $O(h)$ for h=height

Or balance $O(\lg n)$ for each item

~~Or Hash Table~~ ?

(19)

#4 Protien shaps

- list of entries

- want # in comon
  - rotation + translation allowed

Want # in common
The point here is hashing

Have translate + rotate features
- prob takes too much time to try each

But basically have it hash to same value
  ↳ how does that?

What is core about shape?

Could base rotate to something standard
In not much time
So a line of 3,1 but  is also

(20)

Do we have a distinct limit on them → no

~~some~~

Key is efficent hash function

But what ~~must~~ we have to be identical

Oh translate is just move
- not flip as I thought

Could rotate all 4 ways
But I really think trick → need no rotations

So 3 long 1 short
but 

Or blank, 1, blank

Or rotate so longest row is on bottom?
└ rotate is prob long

Hint: dictionaries
hash() function

(21)

But its question of what to hash

Need some shape language

~~Or gotta~~ Or rotate till long flat row

Or 3,1 and then compare individually?

? or # of pieces

In code they do lenght of list (# of blocks)

· And get canociacc set

? So can we use those

- but get canonical shape does not rotate

Same # shapes - not unique entropy

6.006 PS2

3/5

Arianna + Shri

#2 Append - might be n

So n^2 since bad append

#1

C - Can remove

D - Could you swap?

- surface argument
- or think deeply

#3

- insert based on Hs
- as doing check Ds?

Oh. d_i is same i

Binary tree w/ a_i balanced

Remove - how many els to the left

②

Make balances BST a_i and d_i ~~N~~ $2n \log n$

Remove min ele

Check if a_i or d_i 1

If a_i - append to new BST

~~Find~~ If d_i - find corresponding a_i + remove

~~Remove~~ and add to counter # of

eles > ~~M~~ a_i in 2nd tree

~~? length of BST~~

A_1 A_2 D_2 D_1

1 ~~[illegible]~~

A_1 A_2 A_3 D_1 D_2 D_3

— 1 11 = 3

Don't need ~~2nd~~ 1st balanced tree - but its the same

Actually - counter that added later

to the right

(3)

Sort a_i d_i together

pop 1 el from start

if a_i insert into Bal BST
  using a_i value

else if d_i
  remmove a_i

  Find all $a_j > a_i$ they will follow you
  and ~~you~~ they ~~have~~ arrived after you are leaving before ~~you~~ them

(Shri good at tricks)

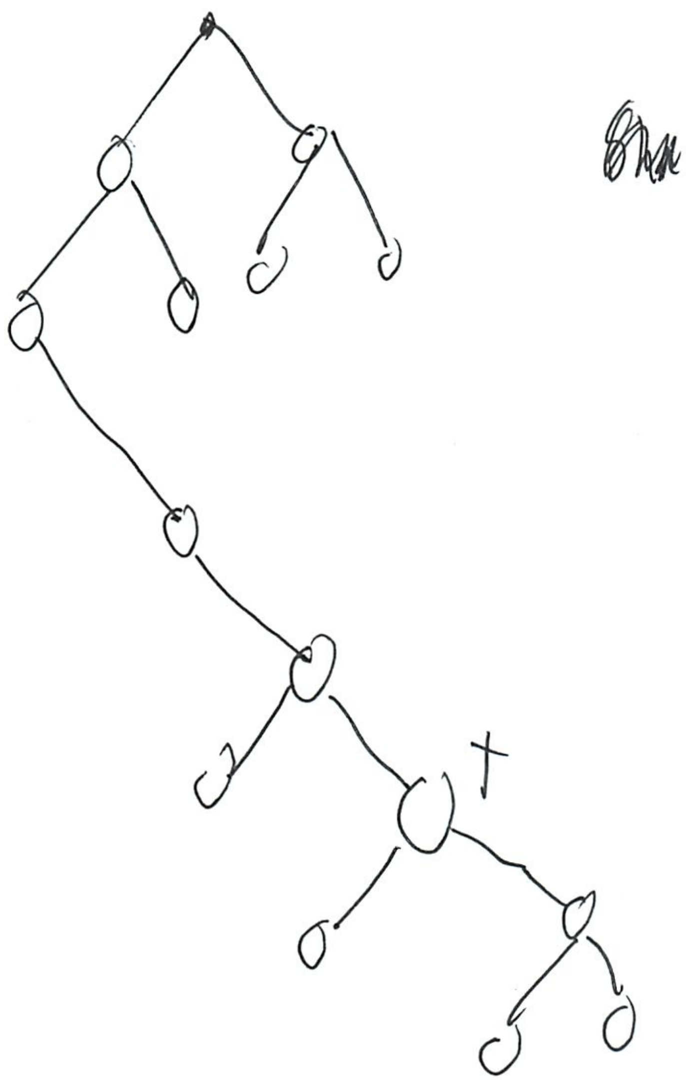
But how do everything greater than you
  - augment?
  - like in lecture - but want greater than t
    $\leq t$ $\geq t$

(4) Find T

+1 to counter

and # of els to right

which is stored in there



Should not be more time then delete → log n

Same time

---

basically n people and log n for almost all

free ops

(5)

Essentially always translate

Rotate is ~~always~~ constant - so not much work

Bottom Left most is 0,0

Is the compare ~~same~~ shape

- Checks size
- if same rotates 4 x
- then save

So the shape stuff is done

So hash

~~Gen~~ Simple is n^2

~~Protien~~ for each protein

~~A~~ Get canonical

Hash

Protien 2

For each

get can
hash
Check

} rotate 4 times = done

(6)

Could we do a compare on length?

Hash el

$0, 0, 2, 2$

$0^1 + 0^2 + 2^3 + 2^4$

Ohh if come in diff order

Just add coords

- hash fail cord

$\sum x \cdot \sum y$

$\sum x + \_\_ \sum y$

$\sum x + \sum x \cdot y$

↑ invarient on order

$\sum x + \sum x \cdot y$

Code

Python

- try it

Instructor #2 append is $\Theta(1)$

(7)

Confusion 1d - if l and r interchanged
- does it ~~[illegible]~~ work?

1f ~~[illegible]~~ - just changing ~~parent~~ pointer
Can still use so Yes
No node.count changed line 27 so No

1e - nothing puts y in for z -
Convinced ∃ 1e False

(✓) Small tests pass 1st try w/o syntax errors

(✗) Medium test cases not work
Why?
Hash overlaps?
Do linked list thing?
Better hash
Undercounting ~~[illegible]~~

~~[illegible]~~ Ton of failing compares
Linked list part not as needed
Implement linked list

⑧

⊗ Now even less!
✓ Fixed linked list — Correct ans

✓ Pass all small + medium test cases
(very happy - first try no issues!)

Hash: based on 2000

"speed up"

3.22 sec before
3.322 sec after

1024 3.28 sec

So run large tests

So ✓ small ✓ medium = 25/50 pts
✓ large in 6 sec = 29/50 pts
need < 3 sec for 50/50

So just speed up

So it ~~finds~~ duplicates a lot!
91 times of ~~[illegible]~~ 199 on a medium test
Need a super cool hash

(7)

To me that seems like the problem

~~Hash~~ Better hash ?

Implement mod %

- are we going over a word

No still ~~8~~6 digit #

But all the #s approx the same

Squared is way slower

<< 2 instead ? ← pow (2,2) = 2^2

So $x \cdot 2^2$

0 out of 290 hash fails ← just 4

(V) 3.38 sec for 47 pts

So bit shift

arithmetic

<< n is multiplying by 2^n

>> n is dividing by 2^n

Remove sum 1 (✓) 3.2 sec 48 pts

(10)

So	x << 3	y << 2	(✓) 47 pts (worse)
	x << 3	y << 3	(✓) 47 pts
	x << 1	y << 1	(✗)(✓) 49 pts
	x << 1	y << 0	(✓) 49 pts slighty faster
	x << 0	y << 1	(✗) 6.6 sec
	x << 0	y << 0	(✗) 6.5

We are at 3.02 sec

So now just write up #3 + done

Solutions

```
collaborators = 'Your collaborators here'

# Answer True or False for each part of problem 1.

answer_for_problem_1_part_a = True
# This also correctly moves the left child into the node's place

answer_for_problem_1_part_b = False
# This may cause violation of the BST property

answer_for_problem_1_part_c = True
# This node cannot be None, since the right node exists

answer_for_problem_1_part_d = True
# Everything is symmetric.

answer_for_problem_1_part_e = False
# Replacing next with his right child works, but we forgot to update val and count

answer_for_problem_1_part_f = False
# next.val will change if we recursively delete, causing our updated node.val to be wrong

answer_for_problem_1_part_g = True
# At this point, this function must turn True

# On problem 2, your answer to part a) should be a boolean, your to part b)
# should be an integer, and your answer to part c) should be a string.
answer_for_problem_2_part_a = True
answer_for_problem_2_part_b = 2
answer_for_problem_2_part_c = """
Suppose some data structure supported insert, get_min, and delete of arbitrary ordered
values, in o(log n).
Then, the above algorithm would run in o(n log n) and also correctly sort.  This is a
contradiction.
"""


# Enter your answer to problem 3 here.
answer_for_problem_3 = '''
    Sort all arrival and departure times together in an array A, putting a
    pointer indicating whether the time in the sorted array is an arrival or
    a departure time of a person in the party.
    Let T be an augmented BST, where at each node X we keep the number of
    elements of the subtree rooted at X. Initially, T is empty. Each node will
    be a person that went to the party, where the value of the key is their
    departure time. Hence, T will be a BST in which each node is put with value
    being its departure time and each node has the extra information described
    above. (we are doing this so that we can compute the rank in O(log n)).
    Let ANS be the total number of follows we will have in the end. Initially,
    ANS = 0.
    Loop through the entries of A as follows: (i ranging from 1 to 2n)
        If A[i] is an arrival time, then insert the corresponding element E in T and check
its rank in T. Let K_e be the rank of E in T. Then, ANS += K_e - 1. (because we don't
```

```
follow ourselves on twitter :-p)
        If A[i] is a departure time, remove element E from T.

    Correctness: when we insert E in T, we have that all elements in T are the
    people who were at the party when E arrived, since we are inserting E by
    arrival time. Moreover, T does not contain people who left the party by the
    time we insert E, since they will be removed from T as they leave. Hence,
    K_e - 1 is exactly the number of people that arrived at the party by the
    time the E arrived and that will leave the party before E leaves, since the
    tree is constructed by departure time. Hence, ANS will be the number that we
    want in the end.

    Runtime analysis: it takes O(n log n) to sort the array A. Then, to insert/
    delete/compute the rank in T, each of these operations take O(log n). Since
    we perform operations in T 2n times, the total time of going thorugh and
    performing operations in the tree is O(n log n).
    This gives a total running time of O(n log n).
'''

# -- Problem 4 code begins here. You need to finish the last function. --

# Rotates the polyomino 90 degrees counterclockwise. Returns a new list.
def rotate(polyomino):
  return [(-y, x) for (x, y) in polyomino]

# Translates the polyomino by the offset. Returns a new list.
def translate(polyomino, offset):
  return [(x + offset[0], y + offset[1]) for (x, y) in polyomino]

# Checks if two polyominoes are equivalent under rotation and translation. Runs
# in linear time in the size of the two polyominoes.
def compare(poly1, poly2):
  if len(poly1) != len(poly2):
    return False

  # Translates a polyomino so that it just touches the x and y axes. Returns the
  # set of squares in the translated shape.
  def get_canonical_set(polyomino):
    offset = (-min(x for (x, y) in polyomino), -min(y for (x, y) in polyomino))
    return set(translate(polyomino, offset))

  poly1_set = get_canonical_set(poly1)
  for i in range(4):
    poly2_set = get_canonical_set(poly2)
    if poly1_set == poly2_set:
      return True
    poly2 = rotate(poly2)

  return False


def get_canonical_frozenset(polyomino):
```

```
  offset = (-min(x for (x, y) in polyomino), -min(y for (x, y) in polyomino))
  return frozenset(translate(polyomino, offset))

def add_protein_to_dictionary(protein, d):
  d[get_canonical_frozenset(protein)] = True

# Fill in the body of this function for Problem 4.
def num_proteins_in_common(protein_list1, protein_list2):
  proteindict = {}
  count = 0

  for protein in protein_list1:
    proteindict[get_canonical_frozenset(protein)] = 1

  for protein in protein_list2:
    for i in range(4):
      if get_canonical_frozenset(protein) in proteindict:
        count += 1
        break
      protein = rotate(protein)

  return common_count
```

6.006 L8

3/6

Sorting

Today: Sort
- Insertion Sort
- Merge Sort

Recurrences
- Master Theorm

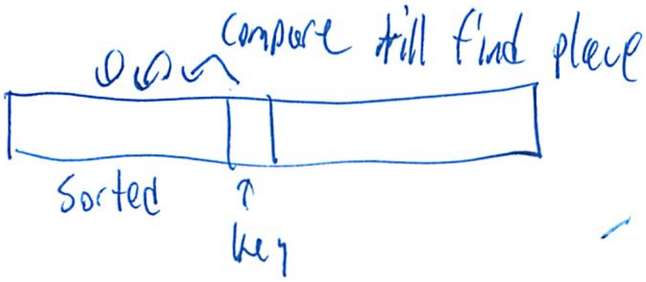
Sort

Want ascending order

$A[1] \le A[2] \le \ldots \le A[n]$

Insertion sort

- Comparison + swapping if needed
  ~~(was in book as the cards)~~



②

Slow!

$O(n^2)$

since you do all n swaps (worst case) for n items

worse case is reverse order

so $\Theta(n^2)$

Early computers just sorted mostly

Merge Sort

steps
1. if n=1 done
2. otherwise recursively sort A[1 .. n/2]
   A[n/2+1 ... n]
3. Merge the 2 sub arrays
   (this was in the book)

(I'm starting to recognize big ideas / getting good at this class)

(3)

Merging

Look for smallest element from the 2 smallest arrays
Put into array

Complexity ~~$\Theta(n)$~~ $O(n)$
$\Theta(n)$ ← usually synonymous

But the recursive sort

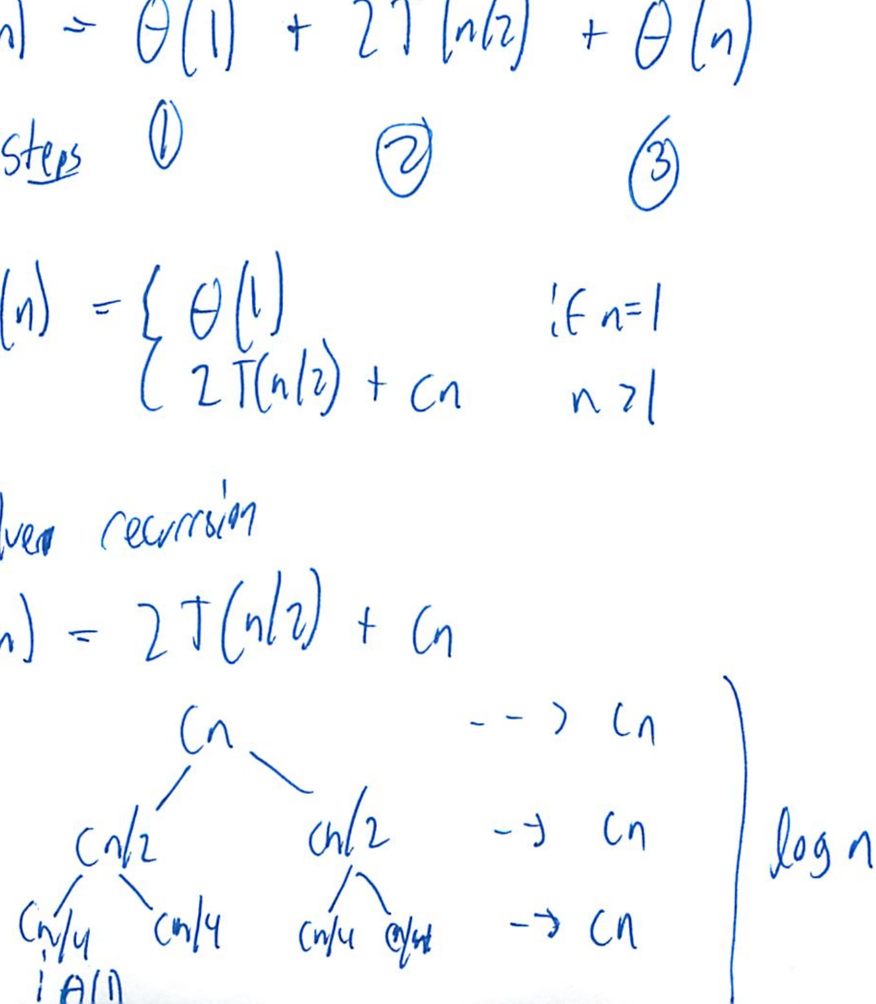
So $T(n) = \Theta(1) + 2T(n/2) + \Theta(n)$

steps (1) (2) (3)

* $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$

Now solve recursion

$T(n) = 2T(n/2) + cn$



(4)

So total = $\Theta(n\log n)$

Master Theorem

One theorem for all recurrences (sort of)
Useful on exam

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$, f is positive

a = # subproblems
b = size of each subproblem
$f(n)$ = time to split ~~prob~~ into sub problems and recombine

Merge sort

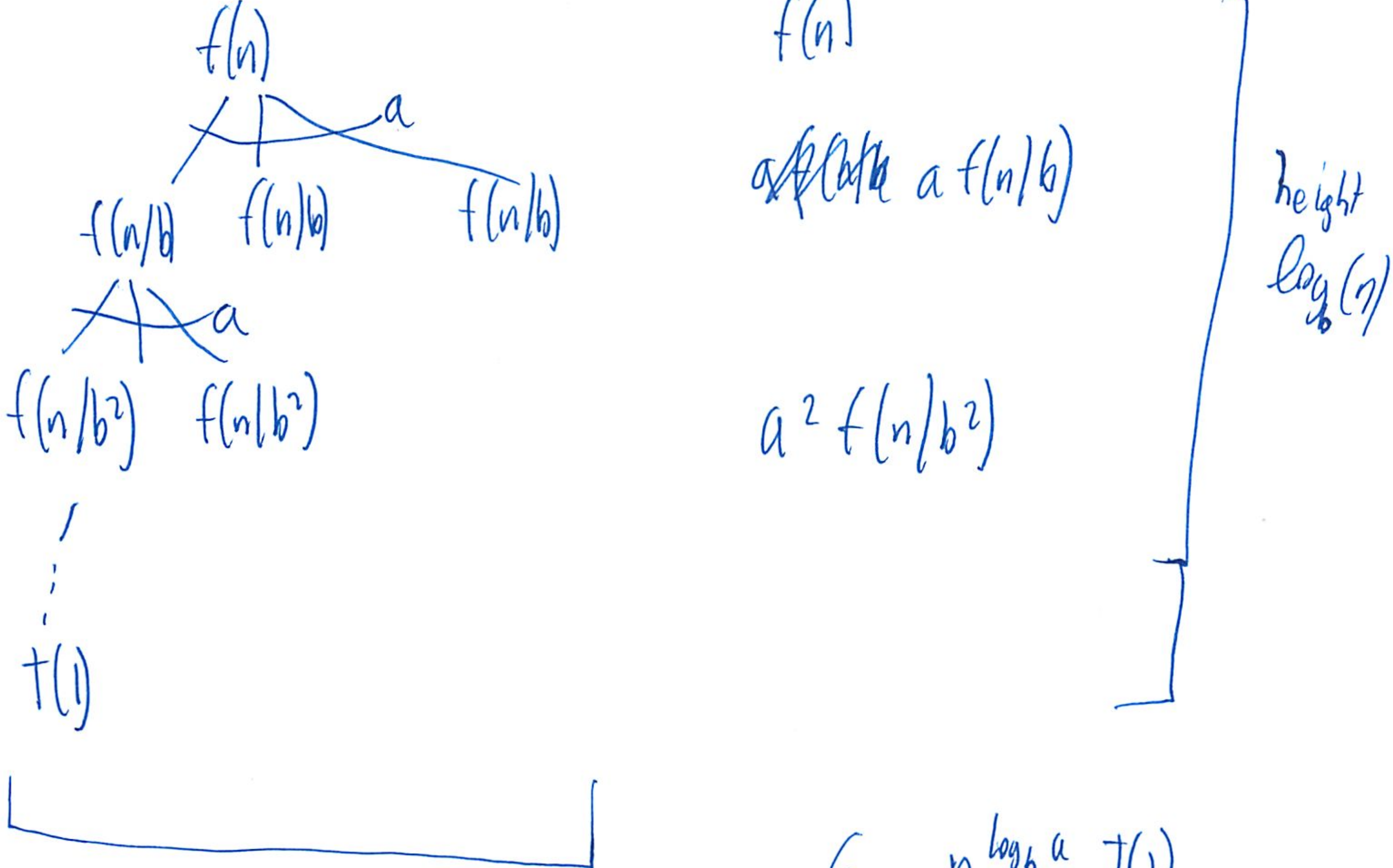
$a = 2$
$b = 2$
$f(n) = O(n)$

Binary Search

$a = 1$
$b = 2$
$f(n) = O(1)$

(5)

So compare $f(n)$ w/ $n^{\log_b a}$



width $= a^h$ = # leaves

$= a^{\log_b n}$

$= n^{\log_b a}$

So $n^{\log_b a}\, T(1)$

= ?

Not clear

<u>Case 1</u> The weight ↑ geometrically from roots to leaves

weight of leaves hold a <u>constant fraction</u>

of total weight → $\Theta(n^{\log_b a})$

(6)

Case 2 Weight approx same on each level

Total weight ≈ # levels · leaves weight

↓

$\Theta(n^{\log_b a} \lg_b n)$

($\lg_b n$ = # of levels)

Case 3 Weight ↓ geometrically from root to leaves

So root holds constant fraction of weight

$\Theta(f(n))$

3 Common Cases

Compare $f(n)$ w/ $n^{\log_b a}$

1. $f(n) = \Theta(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$

$f(n)$ grows polynomially slower than $n^{\log_b a}$ (by a n^ϵ factor)

Cost of level $i = a^i f(n/b^i)$

$= \Theta(n^{\log_b a - \epsilon} \cdot b^{i\epsilon})$

$= \Theta(n^{\log_b a - \epsilon} \cdot (b^\epsilon)^i)$

(i)

This is exactly a geometric increasing series

So leave level dominates

$T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$

$f(n)$ and $n^{\log_b a}$ grow at similar rates

(This class about difference polynomial and exponential)

polynomial much better than exponential

Cost of level $i = a^i f(n/b^i) = \Theta(n^{\log_b a} \cdot \log^k(n/b^i))$

So all levels ~ same cost

$T(n) =$

3. $f(n) = \Theta(n^{\log_b a + \epsilon})$ for some constant $\epsilon \geq 0$

$f(n)$ grows polynomically faster than $n^{\log_b a}$

by n^ϵ factor

Cost of level $i = a^i f(n/b^i) =$

(8)

So root cost dominates

$$T(n) = \Theta\, f(n)$$

Example: Solve recurrence

$$T(n) = 2T(n/2) + 1$$

Use Master Theorm

$a = 2$
$b = 2$

$n^{\log_2 2} = n$

$f(n) = 1$

Case 1 so $f(n) = O(n^{1-\epsilon})$ for $\epsilon = 1$

$$T(n) = \Theta(n)$$

Example 2

$T(n) = 2T(n/2) + n$

$a = 2$ $\quad n^{\log_b a} = n$

$b = 2$ $\quad f(n) = n$

Case 2 $\quad f(n) = \Theta(n \lg^0 n)$

that is $k = 0$

Example 3

$T(n) = 4T(n/2) + n^3$

$a = 4$ $\quad n^{\log_b a} = n^2$

$b = 2$ $\quad f(n) = n^3$

$f(n) = \Omega(n^{2+\epsilon})$ for $\epsilon = 1$

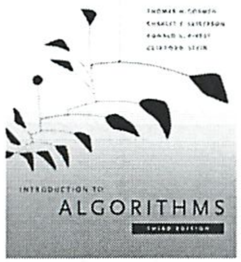
$T(n) = \Theta(n^3)$

[We did in recitation - the cases were presented eaiserly]

(10)

Prof: practice doing recurrences

6.006- *Introduction to Algorithms*



Lecture 8

Prof. Silvio Micali

CLRS: chapter 4.

Menu

- Sorting!
  - Insertion Sort
  - Merge Sort
- Recurrences
  - Master theorem

The problem of sorting

Input: array A[1…n] of numbers.

Output: permutation B[1…n] of A such that
$B[1] \leq B[2] \leq \cdots \leq B[n]$.

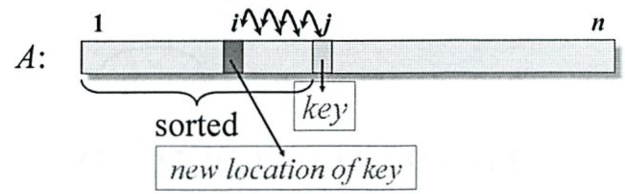
e.g. A = [7, 2, 5, 5, 9.6] → B = [2, 5, 5, 7, 9.6]

How can we do it efficiently ?

Insertion sort

INSERTION-SORT (A, n) ⊳ $A[1 \,..\, n]$
for $j \leftarrow 2$ **to** n
insert key A[j] into the (already sorted) sub-array A[1 ..j-1]
by pairwise key-swaps down to its right position

Illustration of iteration *j*



Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6 *1 swap*

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6 *1 swap*

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6 *0 swap*

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6 *3 swaps*

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6 *2 swaps*

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6
2 3 4 6 8 9 *done*

Running time? $O(n^2)$
e.g. when input is $A = [n, n-1, n-2, \ldots, 2, 1]$

Meet Merge Sort

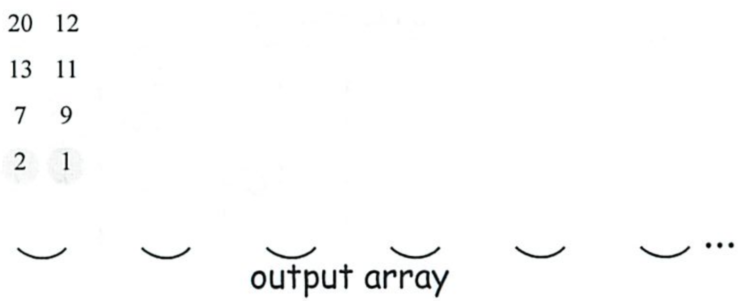
divide and conquer:

MERGE-SORT $A[1 \,..\, n]$

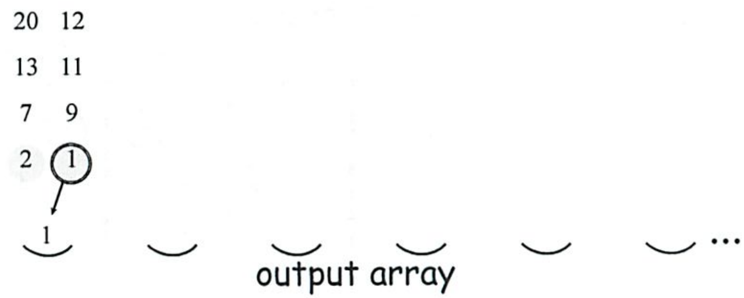
1. If $n = 1$, done (nothing to sort).
2. Otherwise, recursively sort $A[1 \,..\, n/2]$ and $A[n/2+1 \,..\, n]$.
3. "***Merge***" the two sorted sub-arrays.

Key subroutine: **MERGE**

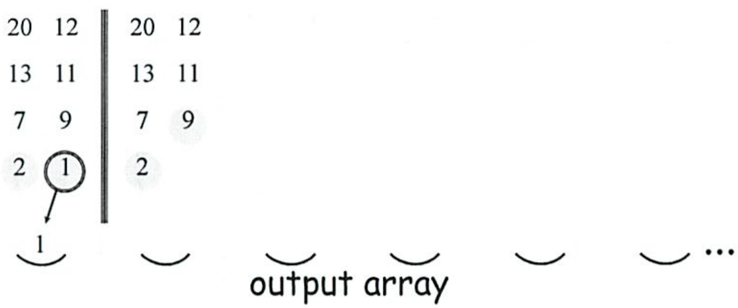
Merging two sorted arrays



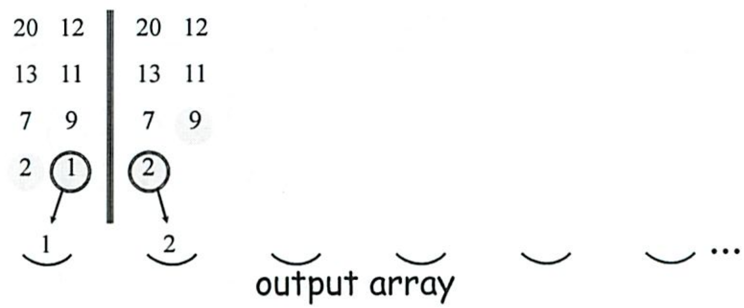
Merging two sorted arrays



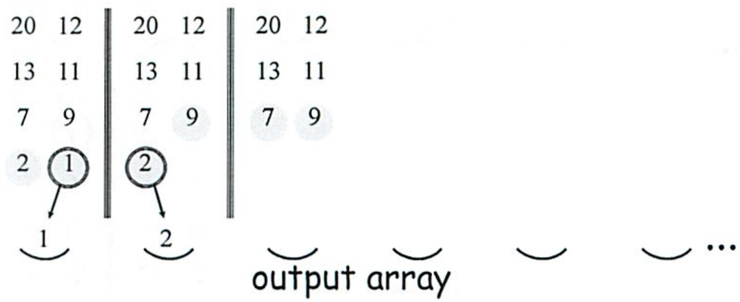
Merging two sorted arrays



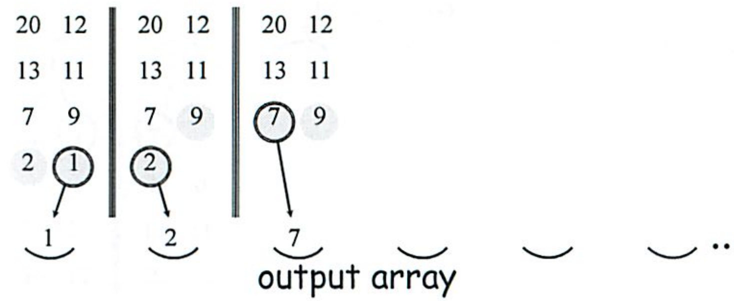
Merging two sorted arrays



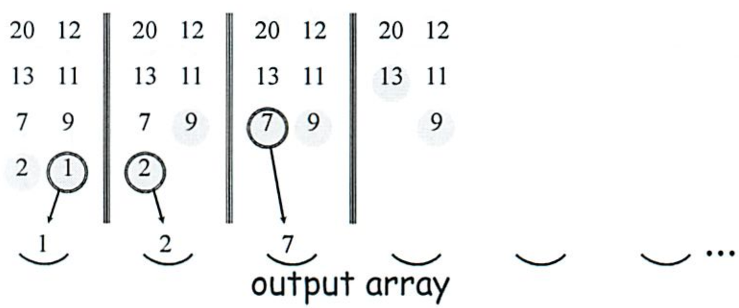
Merging two sorted arrays



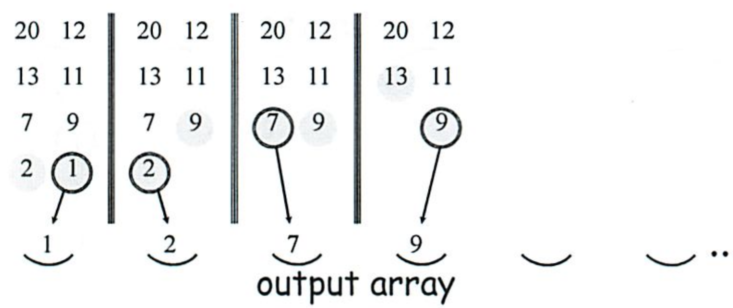
Merging two sorted arrays



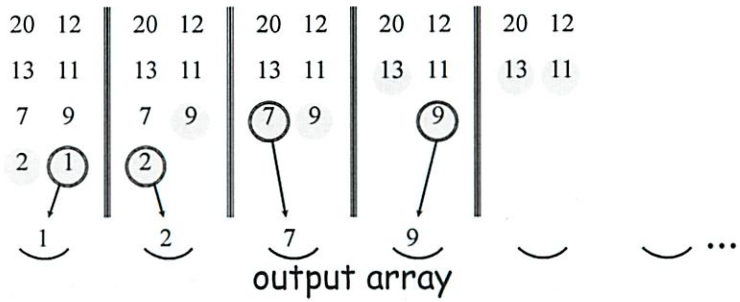
Merging two sorted arrays



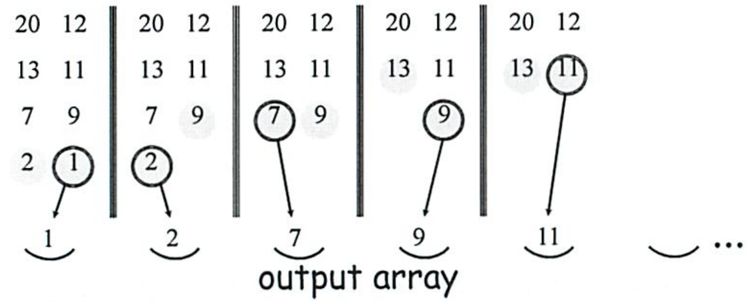
Merging two sorted arrays



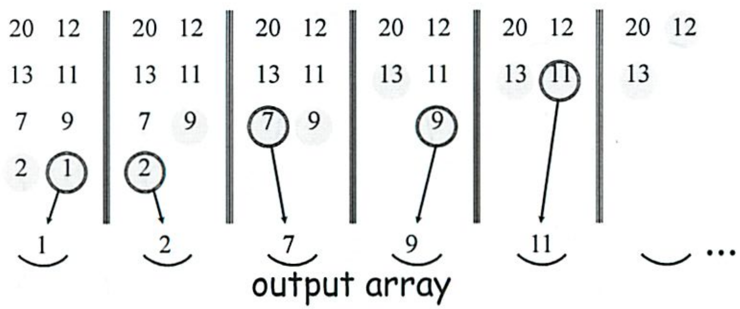
Merging two sorted arrays



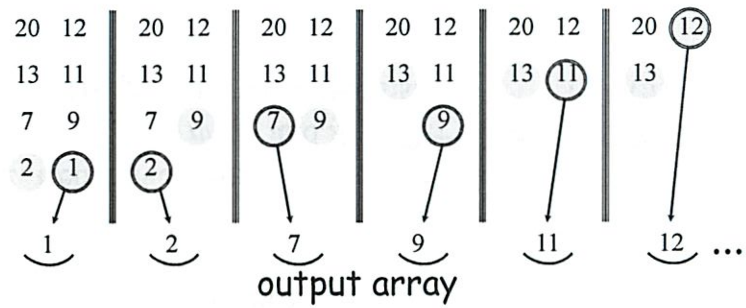
Merging two sorted arrays



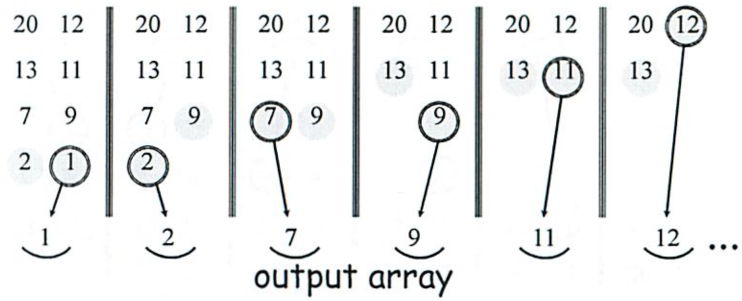
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

Analyzing merge sort

MERGE-SORT $A[1 \,..\, n]$	$T(n)$
1. If $n = 1$, done	$\Theta(1)$
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$	$2T(n/2)$
3. "Merge" the two sorted lists	$\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Recurrence solving

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

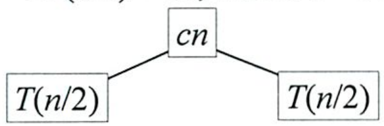
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

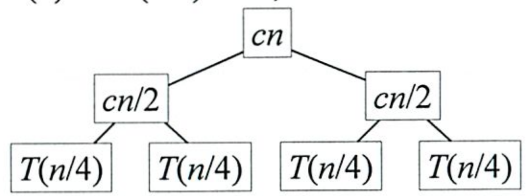
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



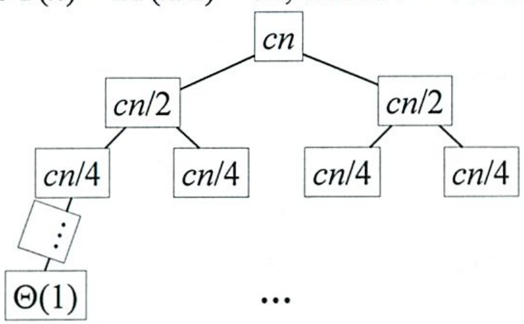
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



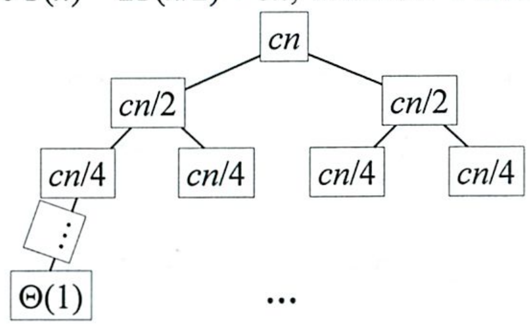
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



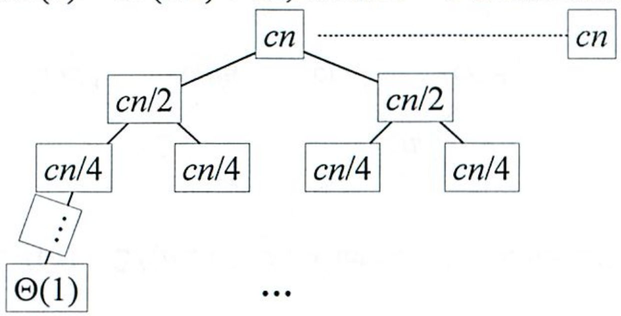
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



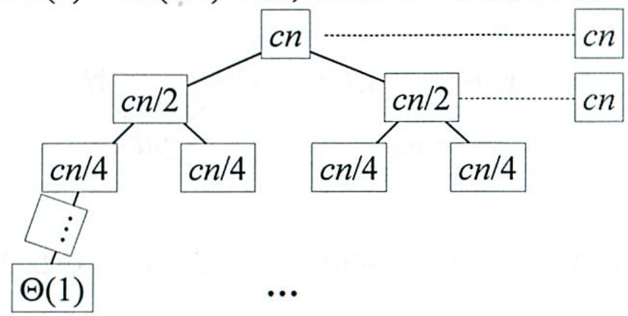
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



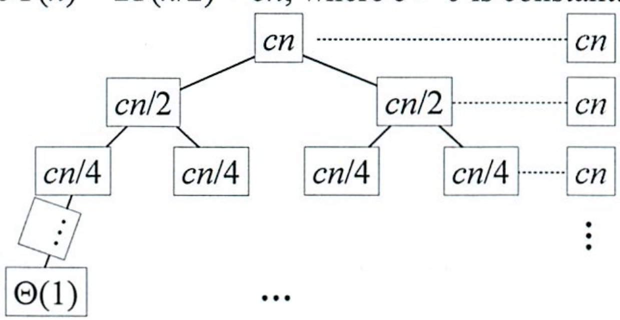
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



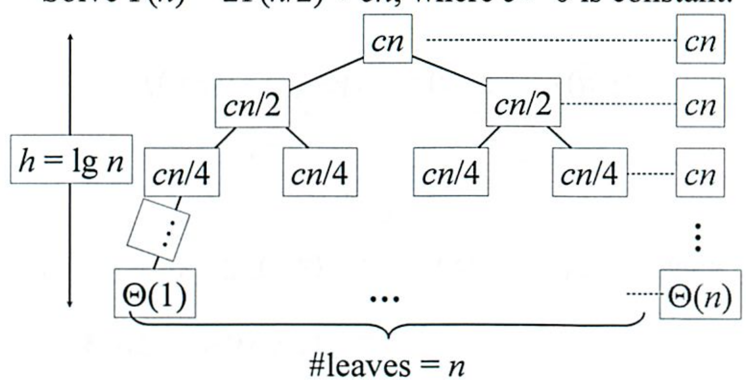
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



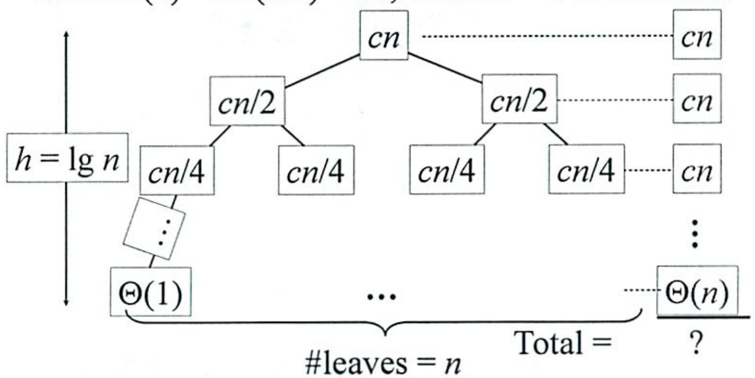
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



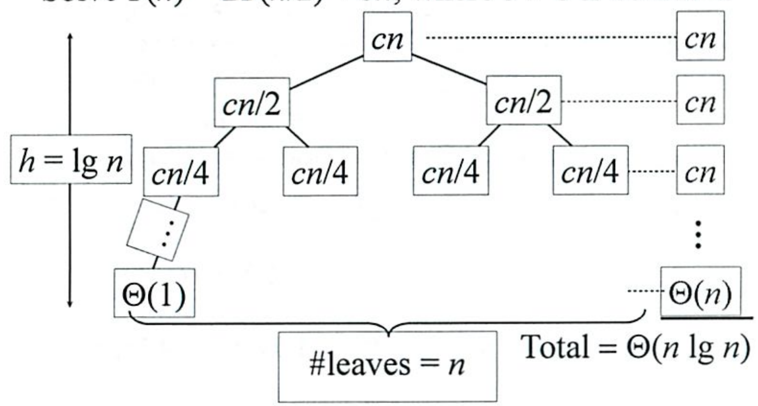
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



The master method

"One theorem for all recurrences" (sort of)

It applies to recurrences of the form

#subproblems; size of each subproblem; time to split into subproblems and combine results

$$T(n) = a\,T(n/b) + f(n),$$

where $a \geq 1$, $b > 1$, and f is positive.

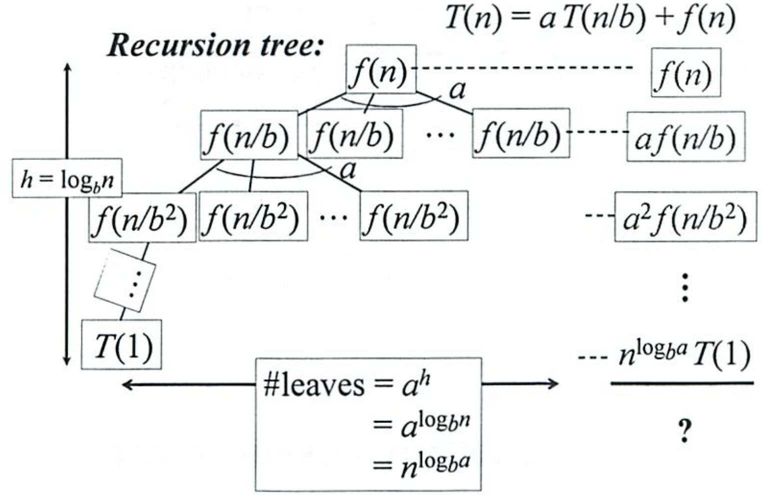
e.g. Mergesort: $a=$

e.g.2 Binary Search: $a=$

Basic Idea: Compare $f(n)$ with $n^{\log_b a}$.

Idea of master theorem

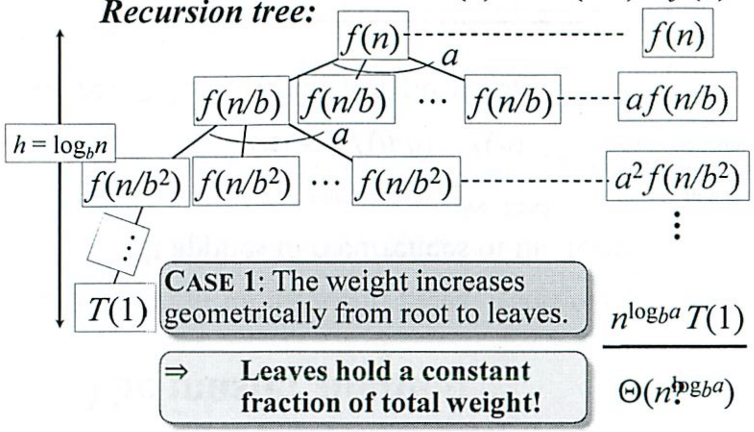
$T(n) = a\,T(n/b) + f(n)$



Idea of master theorem

$T(n) = a\,T(n/b) + f(n)$

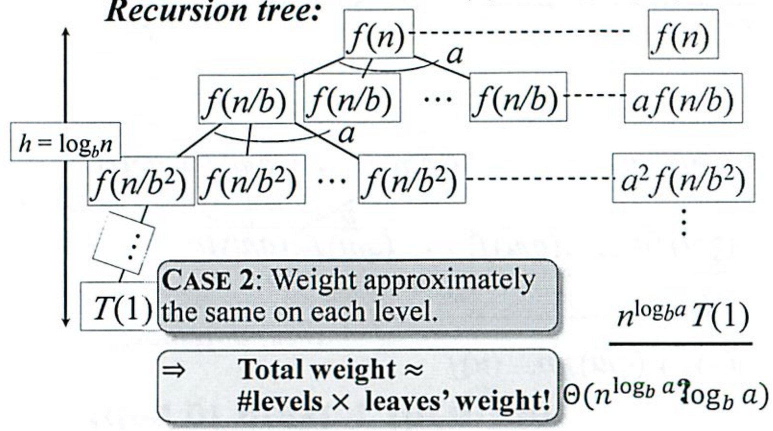
Recursion tree:



Idea of master theorem

$T(n) = a\,T(n/b) + f(n)$

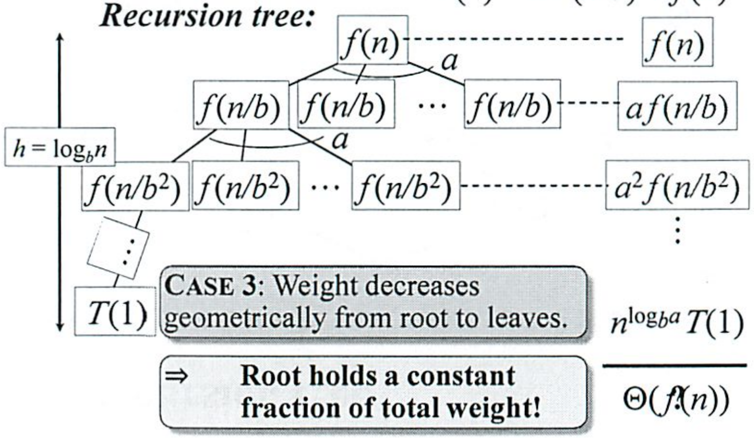
Recursion tree:



Idea of master theorem

$T(n) = a\,T(n/b) + f(n)$

Recursion tree:



Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = \Theta(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

I.e., $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

cost of level $i = a^i f(n/b^i) = \Theta(n^{\log_b a-\varepsilon} \cdot (b^{\varepsilon})^i)$

so geometric increase of cost as we go deeper in the tree

hence, leaf level cost dominates!

Solution: $T(n) = \Theta(n^{\log_b a})$.

Three common cases (cont.)

Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Theta(n^{\log_b a} \log_b^k n)$ for some constant $k \geq 0$.

I.e., $f(n)$ and $n^{\log_b a}$ grow at similar rates.

(cost of level i) = $a^i f(n/b^i) = \Theta(n^{\log_b a} \cdot \log\_b^k(n/b^i))$
so all levels have about the same cost

Solution: $T(n) = \Theta(n^{\log_b a} \log_b^{k+1} n)$

Three common cases (cont.)

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Theta(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.

I.e., $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor).

(cost of level i) = $a^i f(n/b^i) = \Theta(n^{\log_b a+\varepsilon} \cdot b^{-i\varepsilon})$
so geometric decrease of cost as we go deeper in the tree
hence, root cost dominates!

Solution: $T(n) = \Theta(f(n))$.

Example1

$$T(n) = 2T(n/2) + 1$$

Please don't!

$a = 2, b = 2 \Rightarrow n^{\log_b a} = n, \quad f(n) = 1$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon = 1$

$\therefore T(n) = \Theta(n)$.

Use Master Theorem:

1. Compute a and b
2. Compute $n^{\log_b a}$ and $f(n)$
3. Compare

Example 2

$$T(n) = 2T(n/2) + n$$

$a = 2, b = 2 \Rightarrow n^{\log_b a} = n \quad f(n) = n$

CASE 2: $f(n) = \Theta(n \lg^0 n)$, that is, $k = 0$

$\therefore T(n) = \Theta(n \lg n)$.

Example 3

$$T(n) = 4T(n/2) + n^3$$

$a = 4, b = 2 \;\Rightarrow\; n^{\log_b a} = n^2 \quad f(n) = n^3$

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

$\therefore\; T(n) = \Theta(n^3)$.

Let's go master the rest of the day!

6.006 Recitation

3/07

Shi: i Change += to answer = answer + check()

Sorting appears linear - since C offsets the log portion

Frozen set fast - again C

Search trees → order

Hashing does not do order

We did # nodes ≤ value in class

↳ So range queries good in AVL

(Office hrs in Recitation)

6.006 Lecture 8 3/8

Quiz 1 7:30 - 9:30 3/14

Last time Merge Sort $O(n \log n)$

Today Organizing

Heapsort $O(n \log n)$

└ Special type : priority queue

Priority Queue

Stores set S elements

insert (S, x) ← inserts x into S

max (S)

extract_max(S)

increase_key (S, x, k) ← changes x → k

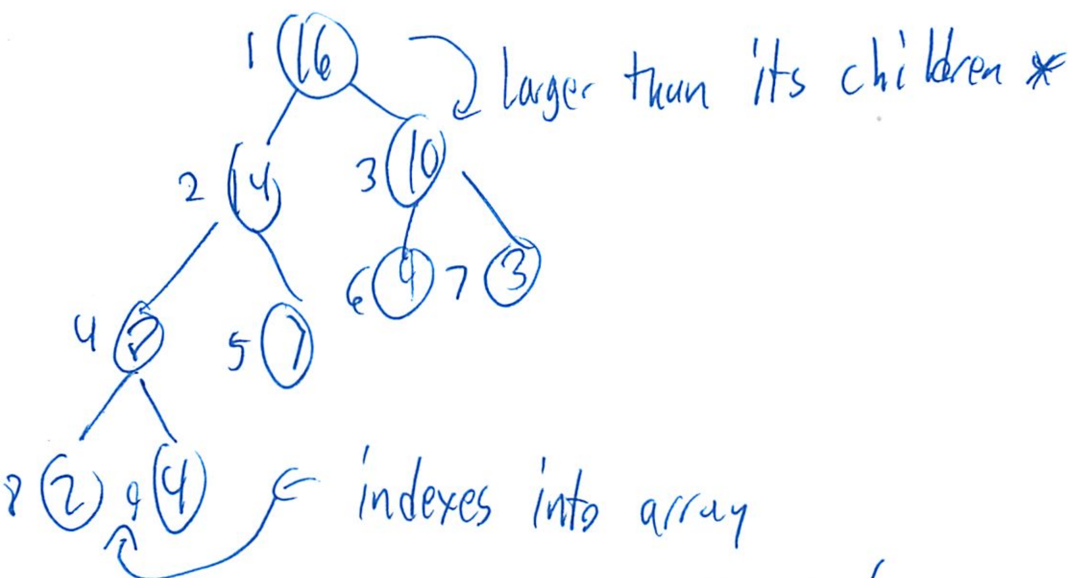
(2)

Heap

Implementation of Priority queue

Array A visualized of complete binary tree

Max-Heap Property key of a node ≥ keys of children



	1	2	3	4	5	6	
A =	16	14	10	8	7	9	...

↑ is legal for a heap

Root of tree: index: 1

If index is i, father is $\lfloor \frac{i}{2} \rfloor$

left(i) is $2i$ ← node i's left child

right(i) = $2i+1$ ← " right "

③

Heap Size Variable

For flexibility may only need 1st few els

So only keep 1st (heap size) elements

$A[1] \dots A[\text{heap size}]$

Max_Heapify(A, i)

Corrects a single violation of heap property at root i

Assumes left(i) right(i) are max heapes

ie $A[i] < A[\text{left}(i)]$ or $A[\text{right}(i)]$

Goal: fix subtree at i

How: Swap values

(Example in slides)

- Swap 4 and 14
- Swap 4 and 8

(4)

Code is in the slides

Remember subtree given must be almost perfect
↑ only solves a very particular problem
$O(\log n)$

~~[illegible]~~

Build Max Heap(A)

Produce a max heap from <u>unordered</u> array

Convert $A[1, \ldots, n]$ to a max heap

<u>Observation</u> Els $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are leaves of the tree


← Start at bottom right


work right to left
upwards

(5)

(Example in slides)

First row is good

Then move upwards swapping items when needed

Induction ← can prove ~~from~~ by

node $\frac{n}{2} \to 1$

left

$O(n \log n)$ since Heapify $O(n)$ times

But not paying log n each time you call it

even at 1st level worst you can pay is 1

Half pay 1

$\frac{1}{4}$ 2

$\frac{1}{8}$ 3

} So $O(1)$ So $O(n)$ time overall

Heap sort (A)

Sort array using heap

(6)

Nieve Sorting

Find largest, put last

Find next largest, put next to lass

etc

$O(n^2)$

Heap sorting

1. Build Max Heap from unordered array $O(n)$

   Note → actually array
   not tree pointers

2. Find max el
3. Swap w/ 1st item
   - violated heap property
   - but at the root is a violation
4. Discard that last item
   └ by ↑ heap-size variable
5. Max heapify at root
6. Go to step 2

(7)

So this is nieve but w/ heap and heapify to fix last el

Etc -> in slides watch rest of it

So heap algorithm works
- we are just always fixing root

<u>Running time</u>

O(n) to build heap

For n iterations

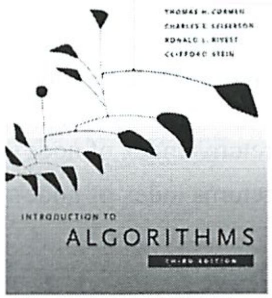
O(log n) to heapify (we saw before)

O(n log n)

Some other operations supported as well

Insert, Extract_max O(log n)

6.006- *Introduction to Algorithms*



Lecture 9

Prof. Constantinos Daskalakis

CLRS: 2.1, 2.2, 2.3, 6.1, 6.2, 6.3 and 6.4.

Lecture Overview

- Priority Queue
  - Heaps
    - Heapsort: a new O(n log n) sorting algorithm

Last time:

- Mergesort for sorting n elements in O(n log n)

This time:

vs

Priority Queue

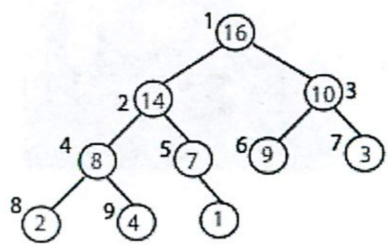
Any data structure storing a set S of elements, each associated with a key, which supports the following operations:

insert(S, x) :	insert element x into set S
max(S) :	return element of S with largest key
extract_max(S) :	return element of S with largest key and remove it from S
increase_key(S, x, k) :	change the key-value of element x to k, if k is larger than current value

Heap

An implementation of a priority queue. It is an array A, visualized as a nearly complete binary tree.

Max-Heap Property: The key of a node is $\geq$ than the keys of its children.



Heap-Size Variable

For flexibility we may only need to consider the first few elements of an array as part of the heap.

The variable heap-size denotes what prefix of the array is part of the heap:

A[1],…, A[heap-size];

Visualizing an Array as a Tree

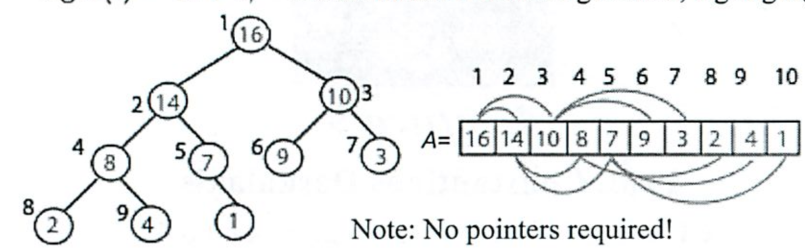
Root of tree: first element in the array, corresponding to index = 1

If a node's index is i then:

$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$; returns index of node's parent, e.g. parent(5)=2

$\text{left}(i) = 2i;$ returns index of node's left child, e.g. left(4)=8

$\text{right}(i) = 2i + 1;$ returns index of node's right child, e.g. right(4)=9



Note: No pointers required!

Operations with Heaps

- *Max_Heapify (A , i)*

Correct a single violation of the heap property occurring at the root i of an otherwise perfect subtree…

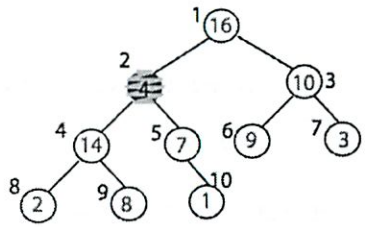
Setting: Assume that the trees rooted at left(i) and right(i) are max-heaps, but element A[i] violates the max-heap property;

i.e. A[i] is smaller than at least one of A[left(i)] or A[right(i)].

Goal: fix the subtree rooted at i.

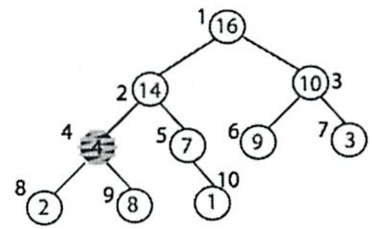
How? Trickle element A[i] down the tree to its right place.

Max_Heapify (Example)



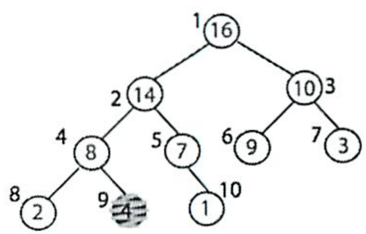
MAX_HEAPIFY (A,2)
heap_size[A] = 10

Max_Heapify (Example)



Exchange A[2] with A[4]
Call MAX_HEAPIFY(A,4)
because max_heap property is violated

Max_Heapify (Example)



Exchange A[4] with A[9]
No more calls

Max_Heapify (Pseudocode)

Max_heapify (A, *i*)

Find the index of the largest element among A[*i*], A[left(*i*)] *and* A[right(*i*)]

```
l ← left(i)
r ← right(i)
if l ≤ heap-size(A) and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size(A) and A[r] > A[largest]
   then largest ← r
```

If this index is different than i, exchange A[*i*] *with largest element; then recurse on subtree*

```
if largest ≠ i
   then exchange A[i] and A[largest]
        MAX_HEAPIFY(A, largest)
```

If A[i] is smaller than both A[left(*i*)] *and* A[right(*i*)] why do I insist on swapping with largest and not with any one of them, arbitrarily?

Operations with Heaps

- *Max_Heapify* (*A* , *i*)

Correct a single violation of the heap property occurring at the root i of an otherwise perfect subtree.
Time O(log n).

- *Build_Max_Heap* (*A*)

Produce a max-heap from an unordered array A.

Build_Max_Heap(A)

Convert $A[1 \ldots n]$ to a max heap.

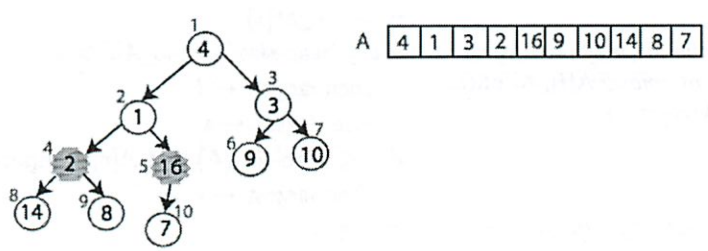
Observation: Elements $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are leaves of the tree

because $2i > n$, for all $i \geq \lfloor n/2 \rfloor + 1$

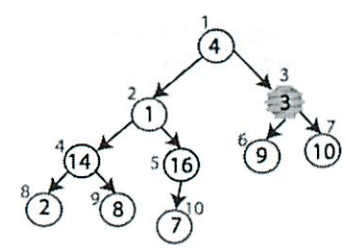
so heap property may only be violated at nodes $1 \ldots \lfloor n/2 \rfloor$ of the tree

```
Build_Max_Heap(A):
    heap_size(A) = length(A)
    for i ← ⌊ length[A]/2⌋ downto 1
        do Max_Heapify(A, i)
```

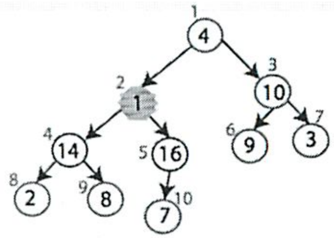
Build_Max_Heap (Example Execution)



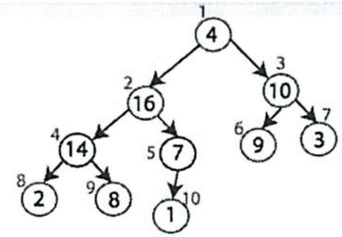
Build_Max_Heap (Example Execution)



Build_Max_Heap (Example Execution)



Build_Max_Heap (Example Execution)



Running Time: O(n log n), since I need to Heapify O(n) times.

Observe, however, that Heapify only pays O(1) time for the nodes that are one level above the leaves, and in general O(ℓ) for the nodes that are ℓ levels above the leaves. ⟶ O(n) time overall!

Operations with Heaps

- *Max_Heapify* (*A* , *i*)

- Correct a single violation of the heap property occurring at the root *i* of an otherwise perfect subtree.
- Time O(log n).

- *Build_Max_Heap* (*A*)

- Produce a max-heap from an unordered array *A*.
- Time O(n)

- *Heapsort* (*A*)

- Sort an array *A* using heaps.

The Naïve Algorithm…

Sorting Strategy:

Find largest element of array, place it in last position; then find the largest among the remaining elements, and place it next to the largest, etc…

In notation:

O(n^2)

1. last_element = n;
2. Find maximum element A[i] of array A[1…last_element];
3. Swap A[i] and A[last_element];
4. last_element = last_element - 1;
5. Go to step 2

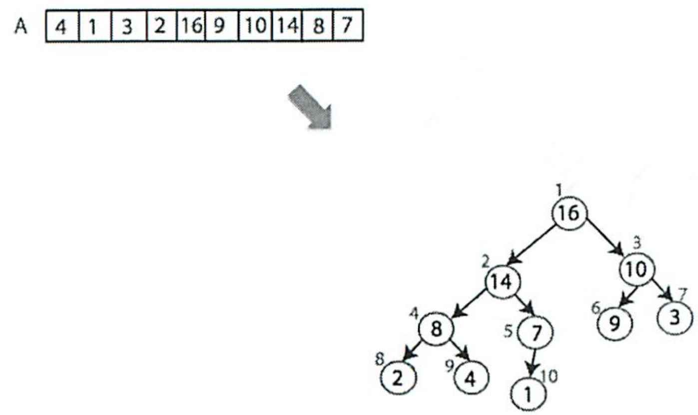
We have a fast data structure for step 2!
(which is also the most costly)

Heapsort

Sorting Strategy:

1. Build Max Heap from unordered array;

Heapsort

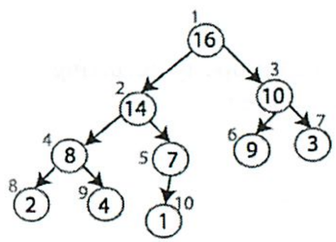


Heapsort

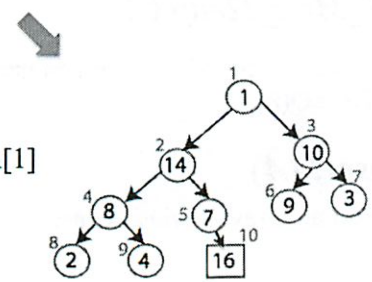
Sorting Strategy:

1. Build Max Heap from unordered array;
2. Find maximum element; this is A[1];
3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!

Heapsort



Swap elements A[10] and A[1]

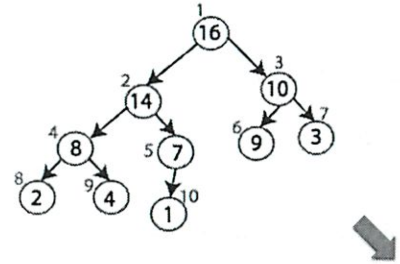


Heapsort

Sorting Strategy:

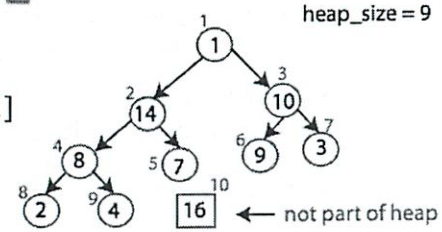
1. Build Max Heap from unordered array;
2. Find maximum element A[1];
3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!
4. Discard node n from heap
   (by decrementing heap-size variable)

Heapsort



Swap elements A[10] and A[1]

heap_size = heap_size-1

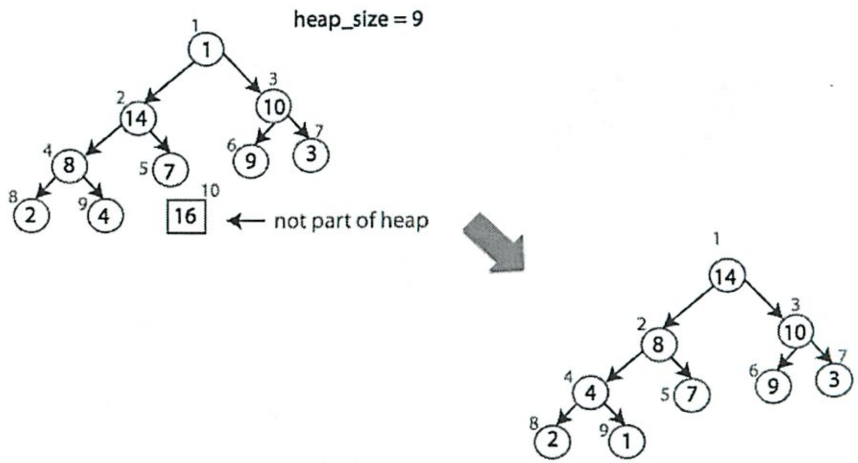


Heapsort

Sorting Strategy:

1. Build Max Heap from unordered array;
2. Find maximum element A[1];
3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!
4. Discard node n from heap
   (by decrementing heap-size variable)
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

Heapsort

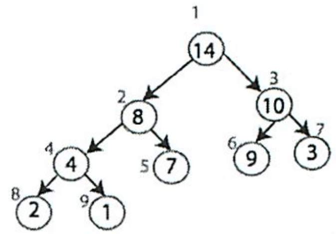


Heapsort

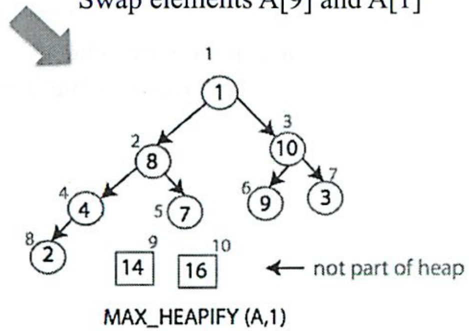
Sorting Strategy:

1. Build Max Heap from unordered array;
2. Find maximum element A[1];
3. Swap elements A[*n*] and A[1]:
   now max element is at the end of the array!
4. Discard node *n* from heap
   (by decrementing heap-size variable)
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.
6. Go to step 2.

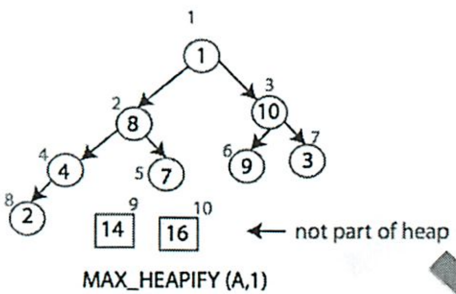
Heapsort



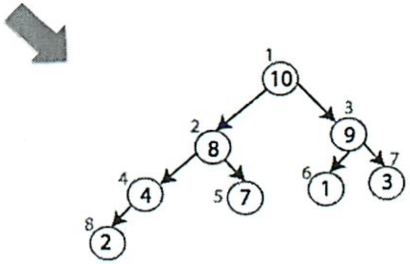
Swap elements A[9] and A[1]



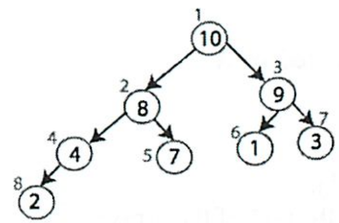
Heapsort



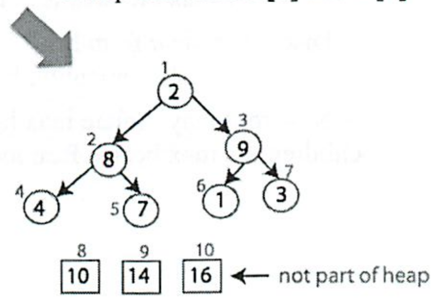
Max_Heapify(A,1)



Heapsort



Swap elements A[8] and A[1]



and so on…

Heapsort Running Time

- O(n) to build heap
- followed by n iterations:

  in each iteration a swap and a heapify is made;
  so O(log n) time spent in each iteration.

Overall O(n log n)

Operations with Heaps Summary

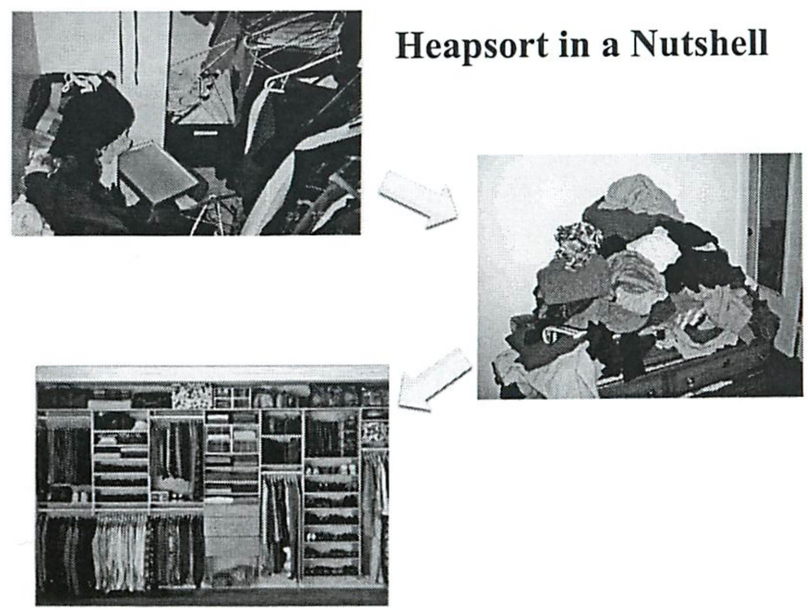
Max_Heapify : correct a single violation of the heap property occurring at the root of a subtree in O(log n);

Build_Max_Heap : produce a max-heap from an unordered array in O(n);

Heapsort : sort an array of size n in O(n log n) using heaps

Insert, Extract_Max ? O(log n)

Heapsort in a Nutshell



6.006 Recitation

3/9

Sorting

Quick + Merge $O(n \log n)$

↑ somewhat extra curricular — ↑ Python w/ optimization

But some can be faster depending on data

Today Quick Sort + Radix Sort

↑ C standard (STL)

Quicksort

Randomized $O(n \log n)$

list $x_1, x_2, \ldots x_n$

Step1 Swap x_1 w/ a random x_i

P (pivot) $= x_1$

→ Pivot around p

$$\underbrace{a_1 \; a_2 \; a_3}_{<p} \quad \underbrace{p \; p \; p}_{=p} \quad \underbrace{b_1 \; b_2 \; b_3}_{>p}$$

(2)

Step 2

recurse on as and bs

Correct? Yes

Running Time? [illegible]

How to pivot?

You swap in place
 ↳ so no extra memory at all (except 3 pointers)

↓ pointer (before first p), ↓ pointer (before x_1), ↓ pointer (before b_1)

$a_1\ a_2 \ldots a_k\ p\ p \ldots p\ \ x_1\ x_2 \ldots x_n\ \ b_1\ b_2 \ldots b_n$

- last p: current location of pivot
- $x_1 \ldots x_n$: not sorted
- $b_1 \ldots b_n$: > pivot

Move next x_i into place w/ $O(1)$ swaps

1. Swap x_1 to x_n
2. Move pointer 1 back (x_n becomes b_1)

Memory has big impact on running time

$O(n)$

(3)

$T(n) = O(n) + T(\sim\frac{n}{2}) + T(\sim\frac{n}{2})$

^ avg case

Worst case n^2

↳ ie if list already sorted!

but the approximations are true if p is random
(lots of math - not covered here)

Better than most sorting algorithms

- 2x fast as merge sort
- Complicated reasons why
- # comparisons less by a factor of 2
- how the swap happens:

if have int a, b can swap w/o a c

$a = a + b$

$b = a - b$

$a = a - b$

could overflow

? intermediate Place

Could say

$c = a$

$a = b$

$b = c$

(4)

XOR together

- fast since logic easy for computer

$a = a \oplus b$ like +=

a ^= b → a b

b ^= a → $a \oplus b$ b

a ^= b → $a \oplus b$ b

$a \oplus a \oplus b = b$

$b = a$

$x \oplus x = 0$

↑ xor

adding involved XORs in the first place

Merge sort has a bunch of copying

So Quicksort is faster than mergesort

This is also why heaps good

- no extra memory
- just do swaps

(5)

Grading Sorting Algos

Not really time → O(n lg n) (all are)

Deterministic - does not use randomness

In-place - does not use extra memory

Stable
- if multiple keys are =, order originally in list are preserved
- actually matters a lot - persere order from 2nd digit
- all can be stable if you try

Cache performance
- linear scans - not jump randomly ← can do a constant # of item at a time
- stay in cache: page as long as need

	MS	QS	CS	RS	
Time	$n \log n$	$n \log n$	ints ≤ n → n	nk	← integers $\leq n^k$
Deterministic	✓	X	✓	✓	
In-place	X	✓	X	X	
Stable	Can be	Can be	(can be)	has to be	
Cache performance (B is cache block size)	$\frac{n \log n}{B}$ ↑ Only 1 extra array; Scanning linearly through 3 lists	$\frac{n \log n}{B}$	n ↑ freq list is getting thrashed	nk	← often a deal breaker

open problem to do (Deterministic, In-place, Stable)

---

binary search log n cache performance ← bad lots of jumping around

(6)

← Radix is a generalization

Counting Sort

list $x_1, \ldots, x_n$ only

Contains #s in $\{0, 1, \ldots, n-1\}$

how to sort?

need ~~[illegible]~~ not be unique

allocate a freq. array of size n

```
for x in list:
    f[x] += 1          ← how many times i appears in the frequency
for i in range n:
    for j in range f[i]:   ← pull those out into the count
        result.append(i)
```

Runtime $O(n)$ ← easy faster

But fails on 2 important real world metrics

- in-place sorting
- cache performance

In real life any of these can work

(7)

Fix stability problem

```
for x in list
    f[x].append(x)
```

4 3 1 1 4

f: [ ] [1,1] [ ] [3] [4,4]

0 1 2 3 4

Could also f[k(x)]

? x is more complex
but get a key for it

like if ordered pairs

(4,0) (3,2) (1,1) (1,2) (4,5)

mixture

f: [(1,1) (1,2)] [ ] [ ] [(3,2)] [(4,0), (4,5)]

(8)

Radix Sort

Removes $\{0, 1, \ldots n-1\}$

Adds $\{0, 1, \ldots, n^k - 1\}$

Based on place value

Runtime $O(n^k)$
↑ gets worse as k↑

Write all numbers in base n

$d_1 \ d_2 \ \ldots \ d_k$

↑ most sig digit — ↑ least significant digit

$0 \leq d_i \leq n-1$ k digits (n^k)

Sort w/ stable counting → key $= d_k$

then key $= d_{k-1}$

⋮

key $= d_1$

(4)

So sort by least sig digit 1st

each takes O(n), do k times = O(nk)

Last sort is what determines order

So preserves order (stable)