Lecture 1: Introduction and Peak Finding

Lecture Overview

- Administrivia
- Course Overview
- "Peak finding" problem — 1D and 2D versions

Course Overview

This course covers:

- Efficient procedures for solving problems on large inputs (Ex: U.S. Highway Map, Human Genome)
- Scalability
- Classic data structures and elementary algorithms (CLRS text)
- Real implementations in Python
- Fun problem sets!

The course is divided into 8 modules — each of which has a motivating problem and problem set(s) (except for the last module). Tentative module topics and motivating problems are as described below:

1. Algorithmic Thinking: Peak Finding
2. Sorting & Trees: Event Simulation
3. Hashing: Genome Comparison
4. Numerics: RSA Encryption
5. Graphs: Rubik's Cube
6. Shortest Paths: Caltech $\rightarrow$ MIT
7. Dynamic Programming: Image Compression
8. Advanced Topics

Peak Finder

One-dimensional Version

Position 2 is a peak if and only if $b \geq a$ and $b \geq c$. Position 9 is a peak if $i \geq h$.

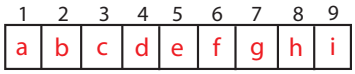
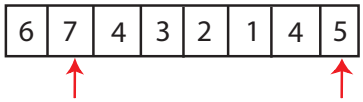


Figure 1: a-i are numbers

Problem: Find a peak if it exists (Does it always exist?)

Straightforward Algorithm



Start from left

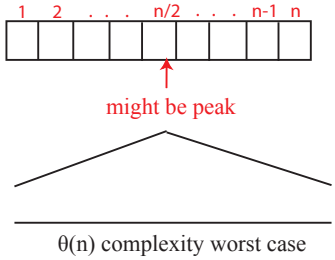
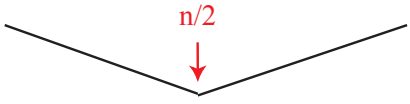


Figure 2: Look at $n/2$ elements on average, could look at n elements in the worst case

What if we start in the middle? For the configuration below, we would look at $n/2$ elements. Would we have to ever look at more than $n/2$ elements if we start in the middle, and choose a direction based on which neighboring element is larger that the middle element?



Can we do better?

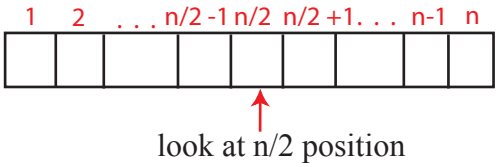


Figure 3: Divide & Conquer

- If $a[n/2] < a[n/2-1]$ then only look at left half $1 \ldots n/2 - - - 1$ to look for peak
- Else if $a[n/2] < a[n/2+1]$ then only look at right half $n/2+1 \ldots n$ to look for peak
- Else $n/2$ position is a peak: WHY?

$$\begin{aligned} a[n/2] &\geq a[n/2-1] \\ a[n/2] &\geq a[n/2+1] \end{aligned}$$

What is the complexity?

$$T(n) = T(n/2) + \underbrace{\Theta(1)}_{\text{to compare a[n/2] to neighbors}} = \Theta(1) + \ldots + \Theta(1) \ (\log_2(n) \ times) = \Theta(\log_2(n))$$

In order to sum up the $\Theta(i)$'s as we do here, we need to find a constant that works for all. If $n = 1000000, \Theta(n)$ algo needs 13 sec in python. If algo is $\Theta(\log n)$ we only need 0.001 sec. Argue that the algorithm is correct.

Two-dimensional Version

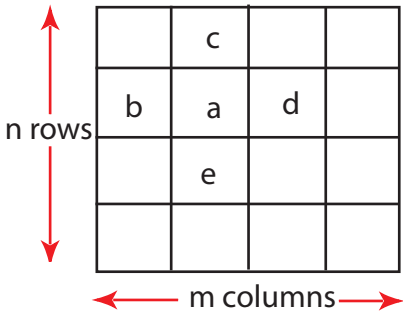


Figure 4: Greedy Ascent Algorithm: $\Theta(nm)$ complexity, $\Theta(n^2)$ algorithm if $m = n$

a is a 2D-peak iff $a \geq b, a \geq d, a \geq c, a \geq e$

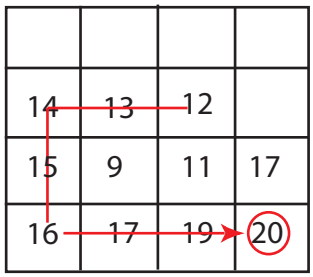
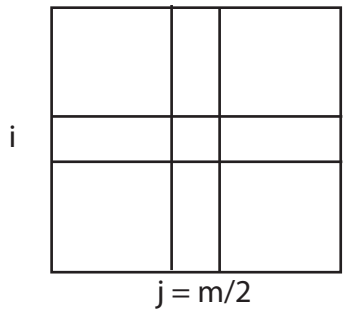


Figure 5: Circled value is peak.

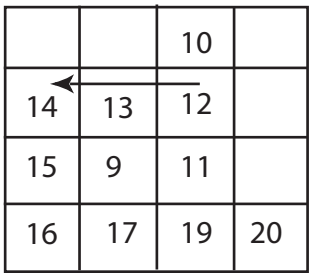
Attempt # 1: Extend 1D Divide and Conquer to 2D



- Pick middle column $j = m/2$.
- Find a 1D-peak at i, j.
- Use (i, j) as a start point on row i to find 1D-peak on row i.

Attempt #1 fails

Problem: 2D-peak may not exist on row i

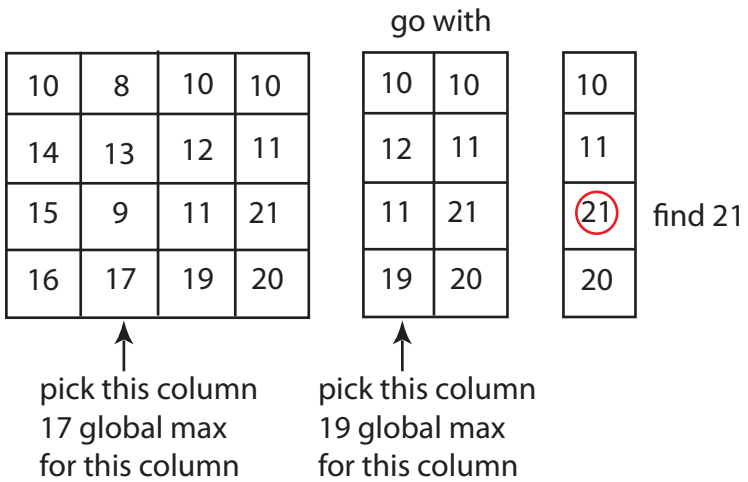


End up with 14 which is not a 2D-peak.

Attempt # 2

- Pick middle column $j = m/2$
- Find global maximum on column j at (i, j)
- Compare $(i, j-1), (i, j), (i, j+1)$
- Pick left columns of $(i, j-1) > (i, j)$
- Similarly for right
- (i, j) is a 2D-peak if neither condition holds ← WHY?
- Solve the new problem with half the number of columns.
- When you have a single column, find global maximum and you're done.

Example of Attempt #2



Complexity of Attempt #2

If $T(n, m)$ denotes work required to solve problem with n rows and m columns

$$\begin{aligned} T(n,m) &= T(n, m/2) + \Theta(n) \text{ (to find global maximum on a column — (n rows))} \\ T(n,m) &= \underbrace{\Theta(n) + \ldots + \Theta(n)}_{\log m} \\ &= \Theta(n \log m) = \Theta(n \log n) \text{ if m = n} \end{aligned}$$

Question: What if we replaced global maximum with 1D-peak in Attempt #2? Would that work?

Lecture 2: Models of Computation

Lecture Overview

- What is an algorithm? What is time?
- Random access machine
- Pointer machine
- Python model
- Document distance: problem & algorithms

History

Al-Khwārizmī "al-kha-raz-mi" (c. 780-850)

- "father of algebra" with his book "The Compendious Book on Calculation by Completion & Balancing"
- linear & quadratic equation solving: some of the first algorithms

What is an Algorithm?

- Mathematical abstraction of computer program
- Computational procedure to solve a problem

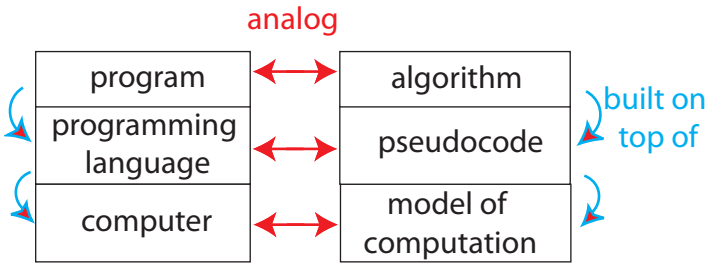
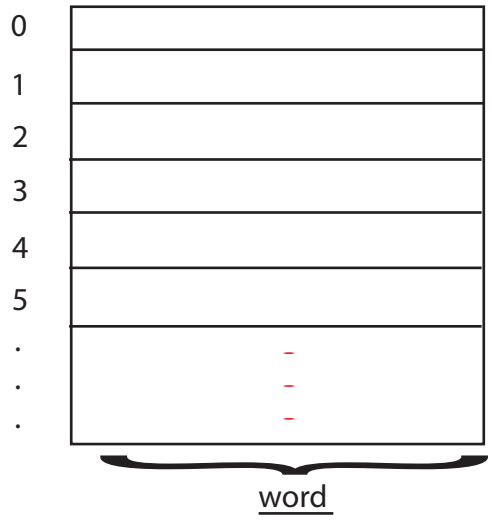


Figure 1: Algorithm

Model of computation specifies

- what operations an algorithm is allowed
- cost (time, space, ...) of each operation
- cost of algorithm = sum of operation costs

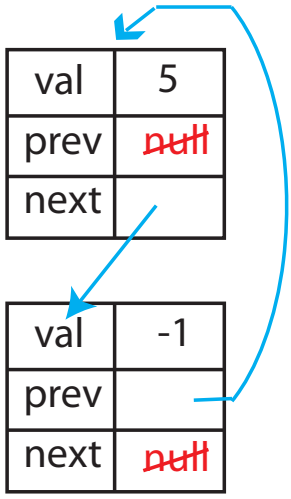
Random Access Machine (RAM)



- Random Access Memory (RAM) modeled by a big array
- $\Theta(1)$ registers (each 1 word)
- In $\Theta(1)$ time, can
  - load word @ r_i into register r_j
  - compute $(+, -, *, /, \&, |, \hat{})$ on registers
  - store register r_j into memory @ r_i
- What's a word? $w \geq \lg(\text{memory size})$ bits
  - assume basic objects (e.g., int) fit in word
  - unit 4 in the course deals with big numbers
- realistic and powerful $\rightarrow$ implement abstractions

Pointer Machine

- dynamically allocated objects (namedtuple)
- object has $O(1)$ fields
- field = word (e.g., int) or pointer to object/null (a.k.a. reference)
- weaker than (can be implemented on) RAM



Python Model

Python lets you use either mode of thinking

1. "list" is actually an array → RAM

   $L[i] = L[j] + 5 \rightarrow \Theta(1)$ time

2. object with $O(1)$ attributes (including references) → pointer machine

   $x = x.next \rightarrow \Theta(1)$ time

Python has many other operations. To determine their cost, imagine implementation in terms of (1) or (2):

1. list

(a) L.append(x) $\rightarrow \theta(1)$ time

obvious if you think of infinite array

but how would you have > 1 on RAM?
via *table doubling* [Lecture 9]

(b) $\underbrace{L = L1 + L2}_{(\theta(1+\lvert L1\rvert+\lvert L2\rvert)\text{ time})} \equiv L = [\,] \rightarrow \theta(1)$

for x in $L1$: } $\theta(\lvert L_1\rvert)$
  L.append(x) $\rightarrow \theta(1)$ }

for x in $L2$: } $\theta(\lvert L_2\rvert)$
  L.append(x) $\rightarrow \theta(1)$ }

(c) $L1$.extend($L2$) $\equiv$ for x in $L2$:
$\equiv L1 += L2$ $L1$.append(x) $\rightarrow \theta(1)$ } $\theta(1 + |L_2|)$ time

(d) $L2 = L1[i:j] \equiv L2 = []$
for k in range(i, j):
$L2$.append$(L1[i]) \rightarrow \theta(1)$ } $\theta(j - i + 1) = O(|L|)$

(e) $b = x$ in L $\equiv$ for y in L:
& L.index(x) if $x == y$: } $\theta(1)$ } $\theta(\text{index of } x) = \theta(|L|)$
& L.find(x) $b = True$;
break
else
$b = False$

(f) len(L) $\rightarrow \theta(1)$ time - list stores its length in a field

(g) L.sort() $\rightarrow \theta(|L| \log |L|)$ - via *comparison sort* [Lecture 3, 4 & 7]

2. tuple, str: similar, (think of as immutable lists)
3. dict: via *hashing* [Unit 3 = Lectures 8-10]
D[key] = val } $\theta(1)$ time w.h.p.
key in D
4. set: similar (think of as dict without vals)
5. heapq: heappush & heappop - via *heaps* [Lecture 4] $\rightarrow \theta(\log(n))$ time
6. long: via *Karatsuba algorithm* [Lecture 11]
$x + y \rightarrow O(|x| + |y|)$ time where $|y|$ reflects # words
$x * y \rightarrow O((|x| + |y|)^{\log(3)})$ $\approx O((|x| + |y|)^{1.58})$ time

Document Distance Problem — compute $d(D_1, D_2)$

The document distance problem has applications in finding similar documents, detecting duplicates (Wikipedia mirrors and Google) and plagiarism, and also in web search (D_2 = query).
Some Definitions:

- Word = sequence of alphanumeric characters
- Document = sequence of words (ignore space, punctuation, etc.)

The idea is to define distance in terms of shared words. Think of document D as a vector: $D[w]$ = # occurrences of word W. For example:

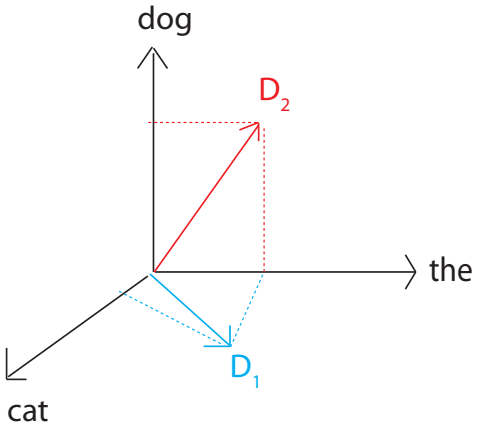


Figure 2: D_1 = "the cat", D_2 = "the dog"

As a first attempt, define document distance as

$$d'(D_1, D_2) = D_1 \cdot D_2 = \sum_W D_1[W] \cdot D_2[W]$$

The problem is that this is not scale invariant. This means that long documents with 99% same words seem farther than short documents with 10% same words.
This can be fixed by normalizing by the number of words:

$$d''(D_1, D_2) = \frac{D_1 \cdot D_2}{|D_1| \cdot |D_2|}$$

where $|D_i|$ is the number of words in document i. The geometric (rescaling) interpretation of this would be that:

$$d(D_1, D_2) = \arccos(d''(D_1, D_2))$$

or the document distance is the angle between the vectors. An angle of $0°$ means the two documents are identical whereas an angle of $90°$ means there are no common words. This approach was introduced by [Salton, Wong, Yang 1975].

Document Distance Algorithm

1. split each document into words
2. count word frequencies (document vectors)
3. compute dot product (& divide)

(1) re.findall (r" w+", doc) → what cost?
in general re can be exponential time
→ for char in doc: } $\Theta(|doc|)$
if not alphanumeric
add previous word } $\Theta(1)$
(if any) to list
start new word

(2) sort word list ← $O(k \log k \cdot |word|)$ where k is #words
for word in list: } $O(\sum |word|) = O(|doc|)$
if same as last word: ← $O(|word|)$
increment counter } $\Theta(1)$
else:
add last word and count to list
reset counter to 0

(3) for word, count1 in doc1: ← $\Theta(k_1)$ } $O(k_1 \cdot k_2)$
if word, count2 in doc2: ← $\Theta(k_2)$
total += count1 * count2 $\Theta(1)$

(3)' start at first word of each list } $O(\sum |word|) = O(|doc|)$
if words equal: ← $O(|word|)$
total += count1 * count2
if word1 ≤ word2: ← $O(|word|)$
advance list1
else:
advance list2
repeat either until list done

Dictionary Approach

(2)' count = {}

for word in doc: } $O(|doc|)$ w.h.p.
if word in count: ← $\Theta(|word|) + \Theta(1)$ w.h.p
count[word] += 1 } $\Theta(1)$
else
count[word] = 1

(3)' as above → $O(|doc_1|)$ w.h.p.

Code (lecture2_code.zip & _data.zip on website)

t2.bobsey.txt 268,778 chars/49,785 words/3354 uniq
t3.lewis.txt 1,031,470 chars/182,355 words/8534 uniq
seconds on Pentium 4, 2.8 GHz, C-Python 2.62, Linux 2.6.26

- docdist1: 228.1 — (1), (2), (3) (with extra sorting)
  words = words + words_on_line
- docdist2: 164.7 — words += words_on_line
- docdist3: 123.1 — (3)' . . . with insertion sort
- docdist4: 71.7 — (2)' but still sort to use (3)'
- docdist5: 18.3 — split words via string.translate
- docdist6: 11.5 — merge sort (vs. insertion)
- docdist7: 1.8 — (3) (full dictionary)
- docdist8: 0.2 — whole doc, not line by line

Lecture 5: Scheduling and Binary Search Trees

Lecture Overview

- Runway reservation system
  - Definition
  - How to solve with lists
- Binary Search Trees
  - Operations

Readings

CLRS Chapter 10, 12.1-3

Runway Reservation System

- Airport with single (very busy) runway (Boston $6 \rightarrow 1$)
- "Reservations" for future landings
- When plane lands, it is removed from set of pending events
- Reserve req specify "requested landing time" t
- Add t to the set if no other landings are scheduled within k minutes either way. Assume that k can vary.
  - else error, don't schedule

Example

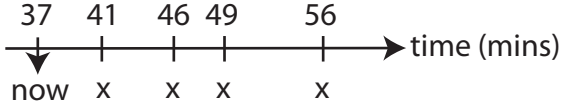


Figure 1: Runway Reservation System Example

Let R denote the reserved landing times: $R = (41, 46, 49, 56)$ and $k = 3$

Request for time: 44 not allowed ($46 \in R$)
53 OK
20 not allowed (already past)
$| R | = n$

Goal: Run this system efficiently in $O(\lg n)$ time

Algorithm

Keep R as a sorted list.

```
init: R = [ ]
req(t): if t < now: return "error"
for i in range (len(R)):
    if abs(t-R[i]) < k: return "error"
R.append(t)
R = sorted(R)
land: t = R[0]
if (t != now) return error
R = R[1: ]            (drop R[0] from R)
```

Can we do better?

- **Sorted list**: Appending and sorting takes $\Theta(n \lg n)$ time. However, it is possible to insert new time/plane rather than append and sort but insertion takes $\Theta(n)$ time. A k minute check can be done in $O(1)$ once the insertion point is found.
- **Sorted array**: It is possible to do binary search to find place to insert in $O(\lg n)$ time. Using binary search, we find the smallest i such that $R[i] \geq t$, i.e., the next larger element. We then compare $R[i]$ and $R[i-1]$ against t. Actual insertion however requires shifting elements which requires $\Theta(n)$ time.
- **Unsorted list/array**: k minute check takes $O(n)$ time.
- **Min-Heap**: It is possible to insert in $O(\lg n)$ time. However, the k minute check will require $O(n)$ time.
- **Dictionary or Python Set**: Insertion is $O(1)$ time. k minute check takes $\Omega(n)$ time

What if times are in whole minutes?

Large array indexed by time does the trick. This will not work for arbitrary precision time or verifying width slots for landing.

Key Lesson: Need fast insertion into sorted list.

Binary Search Trees (BST)

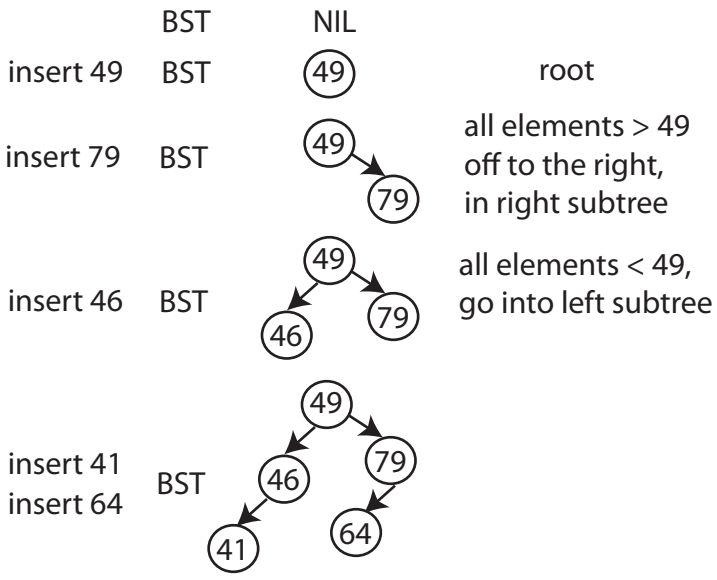


Figure 2: Binary Search Tree

Properties

Each node x in the binary tree has a key $key(x)$. Nodes other than the root have a parent $p(x)$. Nodes may have a left child $left(x)$ and/or a right child $right(x)$. These are pointers unlike in a heap.

The invariant is: for any node x, for all nodes y in the left subtree of x, $key(y) \leq key(x)$. For all nodes y in the right subtree of x $key(y) \geq key(x)$.

Insertion: insert(val)

Follow left and right pointers till you find the position (or see the value), as illustrated in Figure 2. We can do the "within $k = 3$" check for runway reservation during insertion. If you see on the path from the root an element that is within $k = 3$ of what you are inserting, then you interrupt the procedure, and do not insert.

Finding a value in the BST if it exists: find(val)

Follow left and right pointers until you find it or hit NIL.

Finding the minimum element in a BST: findmin()

Key is to just go left till you cannot go left anymore.

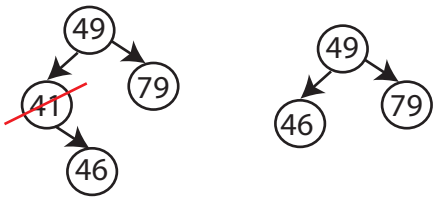


Figure 3: Delete-Min: finds minimum and eliminates it

Complexity

All operations are $O(h)$ where h is height of the BST.

Finding the next larger element: next-larger(x)

Note that x is a node in the BST, not a value.
next-larger(x)

```
if right child not NIL, return minimum(right)
else y = parent(x)
while y not NIL and x = right(y)
    x = y; y = parent(y)
return(y);
```

See Fig. 4 for an example. What would next-larger(find(46)) return?

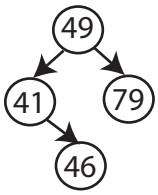


Figure 4: next-larger(x)

New Requirement

Rank(t): How many planes are scheduled to land at times $\leq t$? The new requirement necessitates a design amendment.

Cannot solve it efficiently with what we have but can augment the BST structure.

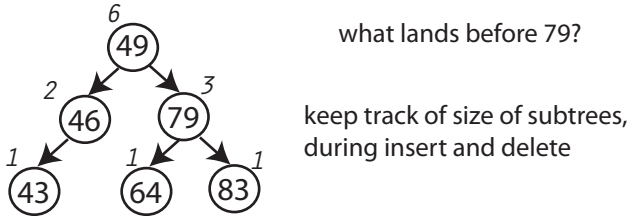


Figure 5: Augmenting the BST Structure

Summarizing from Fig. 5, the algorithm for augmentation is as follows:

1. Walk down tree to find desired time
2. Add in nodes that are smaller
3. Add in subtree sizes to the left

In total, this takes $O(h)$ time.

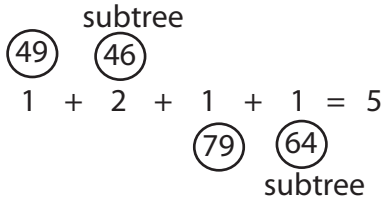


Figure 6: Augmentation Algorithm Example

All the Python code for the Binary Search Trees discussed here are available at this link.

Have we accomplished anything?

Height h of the tree should be $O(\lg n)$.

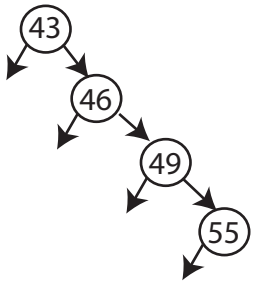


Figure 7: Insert into BST in sorted order

The tree in Fig. 7 looks like a linked list. We have achieved $O(n)$ not $O(\lg n)$!!

Balanced BSTs to the rescue in the next lecture!

Lecture 6: Balanced Binary Search Trees

Lecture Overview

- The importance of being balanced
- AVL trees
  - Definition and balance
  - Rotations
  - Insert
- Other balanced trees
- Data structures in general
- Lower bounds

Recall: Binary Search Trees (BSTs)

- rooted binary tree
- each node has
  - key
  - left pointer
  - right pointer
  - parent pointer

See Fig. 1

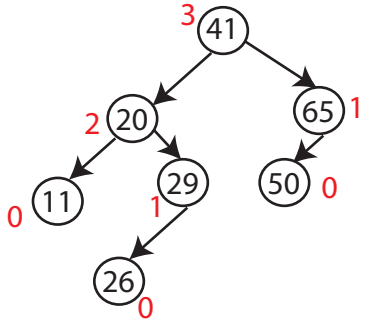


Figure 1: Heights of nodes in a BST

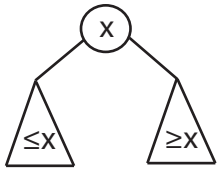


Figure 2: BST property

- BST property (see Fig. 2).
- height of node = length (# edges) of longest downward path to a leaf (see CLRS B.5 for details).

The Importance of Being Balanced:

- BSTs support insert, delete, min, max, next-larger, next-smaller, etc. in $O(h)$ time, where $h =$ height of tree (= height of root).
- h is between $\lg n$ and n: Fig. 3.

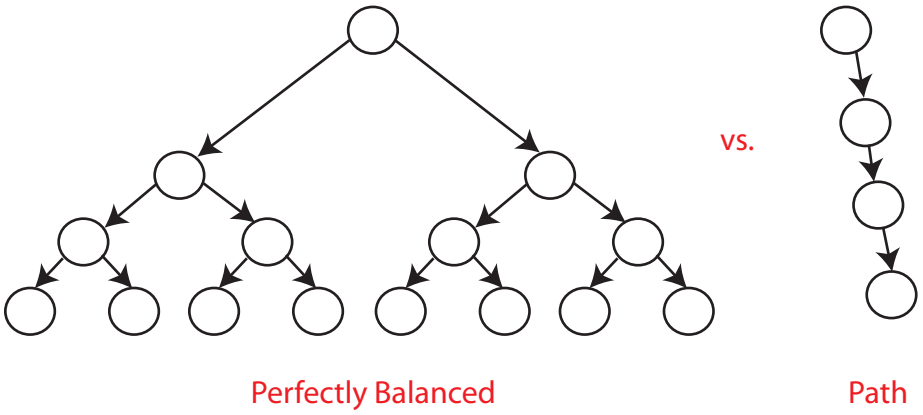


Figure 3: Balancing BSTs

- balanced BST maintains $h = O(\lg n) \Rightarrow$ all operations run in $O(\lg n)$ time.

AVL Trees: Adel'son-Vel'skii & Landis 1962

For every node, require heights of left & right children to differ by at most ± 1.

- treat nil tree as height -1
- each node stores its height (DATA STRUCTURE AUGMENTATION) (like subtree size) (alternatively, can just store difference in heights)

This is illustrated in Fig. 4

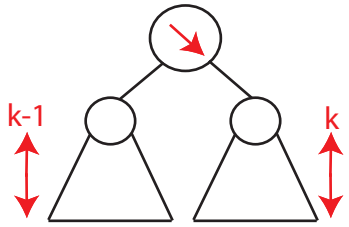


Figure 4: AVL Tree Concept

Balance:

Worst when every node differs by 1 — let N_h = (min.) # nodes in height-h AVL tree

$$\begin{aligned} \implies N_h &= N_{h-1} + N_{h-2} + 1 \\ &> 2N_{h-2} \\ \implies N_h &> 2^{h/2} \\ \implies h &< 2 \lg N_h \end{aligned}$$

Alternatively:
$N_h > F_h$ (nth Fibonacci number)

- In fact $N_h = F_{n+1} - 1$ (simple induction)
- $F_h = \frac{\varphi_h}{\sqrt{5}}$ rounded to nearest integer where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ (golden ratio)
- $\implies$ max. $h \approx \log_\varphi n \approx 1.440 \lg n$

AVL Insert:

1. insert as in simple BST
2. work your way up tree, restoring AVL property (and updating heights as you go).

Each Step:

- suppose x is lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- if x's right child is right-heavy or balanced: follow steps in Fig. 5

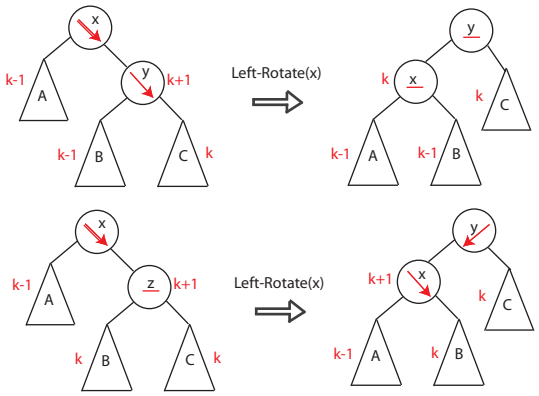


Figure 5: AVL Insert Balancing

- else: follow steps in Fig. 6

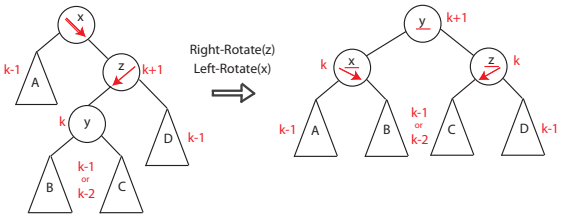


Figure 6: AVL Insert Balancing

- then continue up to x's grandparent, greatgrandparent ...

Example: An example implementation of the AVL Insert process is illustrated in Fig. 7

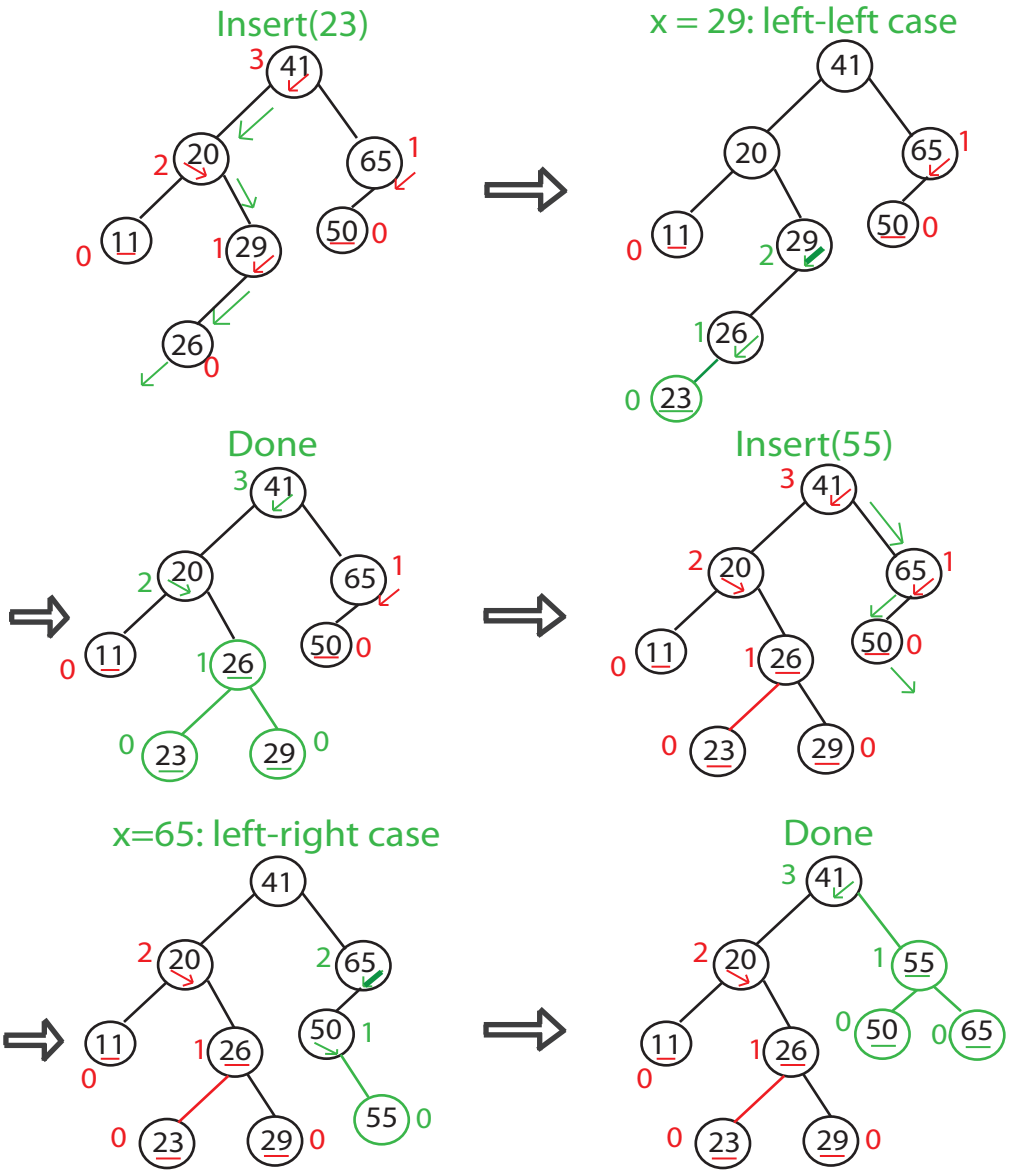


Figure 7: Illustration of AVL Tree Insert Process

Comment 1. In general, process may need several rotations before done with an Insert.

Comment 2. Delete(-min) is similar — harder but possible.

AVL sort:

- insert each item into AVL tree $\Theta(n \lg n)$
- in-order traversal $\Theta(n)$

$\overline{\Theta(n \lg n)}$

Balanced Search Trees:

There are many balanced search trees.

AVL Trees	Adel'son-Velsii and Landis 1962
B-Trees/2-3-4 Trees	Bayer and McCreight 1972 (see CLRS 18)
BB[α] Trees	Nievergelt and Reingold 1973
Red-black Trees	CLRS Chapter 13
(A) — Splay-Trees	Sleator and Tarjan 1985
(R) — Skip Lists	Pugh 1989
(A) — Scapegoat Trees	Galperin and Rivest 1993
(R) — Treaps	Seidel and Aragon 1996

(R) = use random numbers to make decisions fast with high probability
(A) = "amortized": adding up costs for several operations $\implies$ fast on average
For example, Splay Trees are a current research topic — see 6.854 (Advanced Algorithms) and 6.851 (Advanced Data Structures)

Big Picture:

Abstract Data Type(ADT): interface spec.

vs.

Data Structure (DS): algorithm for each op.

There are many possible DSs for one ADT. One example that we will discuss much later in the course is the "heap" priority queue.

Priority Queue ADT	**heap**	**AVL tree**
Q = new-empty-queue()	$\Theta(1)$	$\Theta(1)$
Q.insert(x)	$\Theta(\lg n)$	$\Theta(\lg n)$
x = Q.deletemin()	$\Theta(\lg n)$	$\Theta(\lg n)$
x = Q.findmin()	$\Theta(1)$	$\Theta(\lg n) \to \Theta(1)$

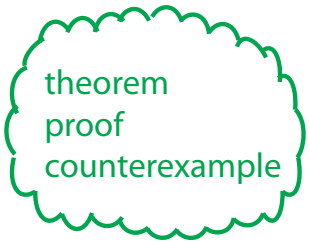
Predecessor/Successor ADT	**heap**	**AVL tree**
S = new-empty()	$\Theta(1)$	$\Theta(1)$
S.insert(x)	$\Theta(\lg n)$	$\Theta(\lg n)$
S.delete(x)	$\Theta(\lg n)$	$\Theta(\lg n)$
y = S.predecessor(x) → next-smaller	$\Theta(n)$	$\Theta(\lg n)$
y = S.successor(x) → next-larger	$\Theta(n)$	$\Theta(\lg n)$

Lecture 7: Linear-Time Sorting

Lecture Overview

- Comparison model
- Lower bounds
  - searching: $\Omega(\lg n)$
  - sorting: $\Omega(n \lg n)$
- $O(n)$ sorting algorithms for small integers
  - counting sort
  - radix sort

Lower Bounds



Claim

- searching among n preprocessed items requires $\Omega(\lg n)$ time
  $\Longrightarrow$ binary search, AVL tree search optimal
- sorting n items requires $\Omega(n \lg n)$
  $\Longrightarrow$ mergesort, heap sort, AVL sort optimal

. . . in the comparison model

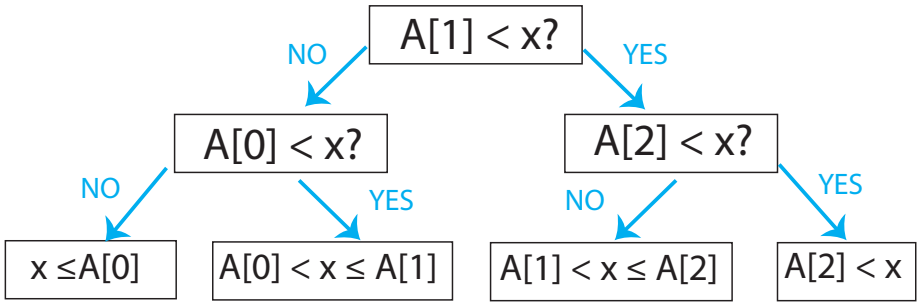
Comparison Model of Computation

- input items are black boxes (ADTs)
- only support comparisons ($<, >, \leq$, etc.)
- time cost = # comparisons

Decision Tree

Any comparison algorithm can be viewed/specified as a tree of all possible comparison outcomes & resulting output, for a particular n:

- example, binary search for $n = 3$:



- internal node = binary decision
- leaf = output (algorithm is done)
- root-to-leaf path = algorithm execution
- path length (depth) = running time
- height of tree = worst-case running time

In fact, binary decision tree model is more powerful than comparison model, and lower bounds extend to it

Search Lower Bound

- # leaves $\geq$ # possible answers $\geq n$ (at least 1 per $A[i]$)
- decision tree is binary
- $\implies$ height $\geq \lg \Theta(n) = \lg n \underbrace{\pm \Theta(1)}_{\lg \Theta(1)}$

Sorting Lower Bound

- leaf specifies answer as permutation: $A[3] \leq A[1] \leq A[9] \leq \ldots$
- all $n!$ are possible answers

- # leaves $\geq n!$

$$\begin{aligned}
\implies \text{height} &\geq \lg n! \\
&= \lg(1 \cdot 2 \cdots (n-1) \cdot n) \\
&= \lg 1 + \lg 2 + \cdots + \lg(n-1) + \lg n \\
&= \sum_{i=1}^{n} \lg i \\
&\geq \sum_{i=n/2}^{n} \lg i \\
&\geq \sum_{i=n/2}^{n} \underbrace{\lg \frac{n}{2}}_{=\lg n - 1} \\
&= \frac{n}{2} \lg n - \frac{n}{2} = \Omega(n \lg n)
\end{aligned}$$

- in fact $\lg n! = n \lg n - O(n)$ via Sterling's Formula:

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \implies \lg n! \sim n \lg n - \underbrace{(\lg e)n + \tfrac{1}{2} \lg n + \tfrac{1}{2} \lg(2\pi)}_{O(n)}$$

Linear-time Sorting

If n keys are integers (fitting in a word) $\in 0, 1, \ldots, k-1$, can do more than compare them

- $\implies$ lower bounds don't apply
- if $k = n^{O(1)}$, can sort in $O(n)$ time
  OPEN: $O(n)$ time possible for all k?

Counting Sort

```
L = array of k empty lists                          } O(k)
    — linked or Python lists
for j in range n:                                   } O(n)
    L[key(A[j])].append(A[j])      → O(1)
    random access using integer key
output = [ ]                                        } O(∑_i (1 + |L[i]|)) = O(k + n)
for i in range k:
    output.extend(L[i])
```

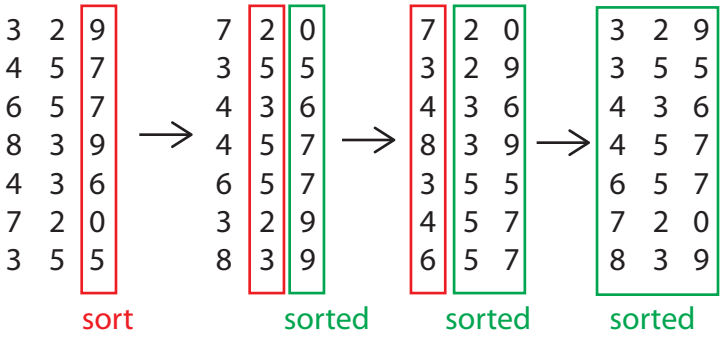
Time: $\Theta(n+k)$ — also $\Theta(n+k)$ space

Intuition: Count key occurrences using RAM output <count> copies of each key in order . . . but item is more than just a key

CLRS has cooler implementation of counting sort with counters, no lists — but time bound is the same

Radix Sort

- imagine each integer in base b
  $\implies$ $d = \log_b k$ digits $\in \{0, 1, \ldots, b-1\}$
- sort (all n items) by least significant digit $\rightarrow$ can extract in $O(1)$ time
- $\cdots$
- sort by most significant digit $\rightarrow$ can extract in $O(1)$ time
  sort must be stable: preserve relative order of items with the same key
  $\implies$ don't mess up previous sorting
  For example:



- use counting sort for digit sort
  - $\implies$ $\Theta(n+b)$ per digit
  - $\implies$ $\Theta((n+b)d) = \Theta((n+b)\log_b k)$ total time
  - minimized when $b = n$
  - $\implies$ $\Theta(n \log_n k)$
  - $= O(nc)$ if $k \leq n^c$

Lecture 8: Hashing I

Lecture Overview

- Dictionaries and Python
- Motivation
- Prehashing
- Hashing
- Chaining
- Simple uniform hashing
- "Good" hash functions

Dictionary Problem

Abstract Data Type (ADT) — maintain a set of items, each with a key, subject to

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists

We assume items have distinct keys (or that inserting new one clobbers old).
Balanced BSTs solve in $O(\lg n)$ time per op. (in addition to inexact searches like next-largest).
Goal: $O(1)$ time per operation.

Python Dictionaries:

Items are (key, value) pairs e.g. $d =$ {'algorithms': 5, 'cool': 42}

d.items()	→	[('algorithms', 5),('cool',5)]
d['cool']	→	42
d[42]	→	KeyError
'cool' in d	→	True
42 in d	→	False

Python set is really dict where items are keys (no values)

Motivation

Dictionaries are perhaps the most popular data structure in CS

- built into most modern programming languages (Python, Perl, Ruby, JavaScript, Java, C++, C#, ...)
- e.g. best docdist code: word counts & inner product
- implement databases: (DB_HASH in Berkeley DB)
  - English word → definition (literal dict.)
  - English words: for spelling correction
  - word → all webpages containing that word
  - username → account object
- compilers & interpreters: names → variables
- network routers: IP address → wire
- network server: port number → socket/app.
- virtual memory: virtual address → physical

Less obvious, using hashing techniques:

- substring search (grep, Google) [L9]
- string commonalities (DNA) [PS4]
- file or directory synchronization (rsync)
- cryptography: file transfer & identification [L10]

How do we solve the dictionary problem?

Simple Approach: Direct Access Table

This means items would need to be stored in an array, indexed by key (random access)

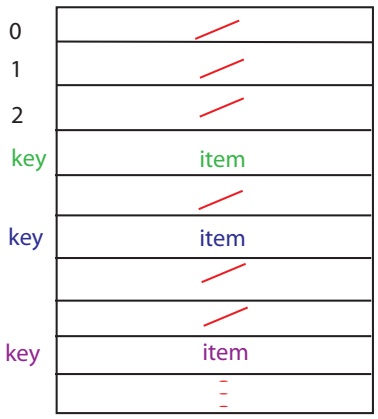


Figure 1: Direct-access table

Problems:

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range $\implies$ large space — e.g. one key of 2^{256} is bad news.

2 Solutions:

Solution to 1: "prehash" keys to integers.

- In theory, possible because keys are finite $\implies$ set of keys is countable
- In Python: hash(object) (actually hash is misnomer should be "prehash") where object is a number, string, tuple, etc. or object implementing __hash__ (default = id = memory address)
- In theory, $x = y \Leftrightarrow \text{hash}(x) = \text{hash}(y)$
- Python applies some heuristics for practicality: for example, hash('\0B') = 64 = hash('\0\0C')
- Object's key should not change while in table (else cannot find it anymore)
- No mutable objects like lists

Solution to 2: hashing (verb from French 'hache' = hatchet, & Old High German 'happja' = scythe)

- Reduce universe $\mathcal{U}$ of all keys (say, integers) down to reasonable size m for table
- idea: $m \approx n =$ # keys stored in dictionary
- hash function h: $\mathcal{U} \to \{0, 1, \ldots, m-1\}$

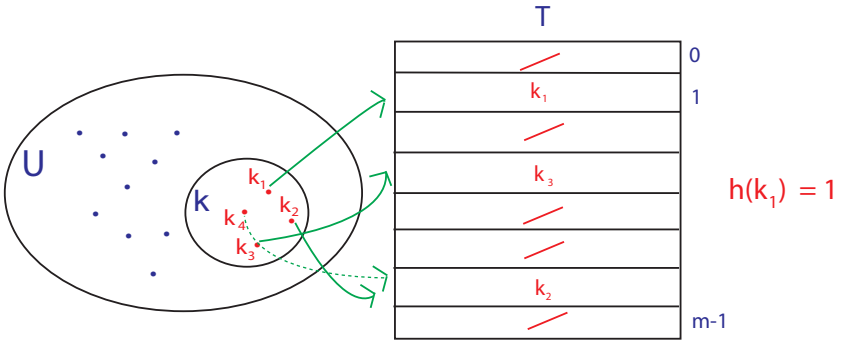


Figure 2: Mapping keys to a table

- two keys $k_i, k_j \in K$ collide if $h(k_i) = h(k_j)$

How do we deal with collisions?

We will see two ways

1. Chaining: TODAY
2. Open addressing: L10

Chaining

Linked list of colliding elements in each slot of table

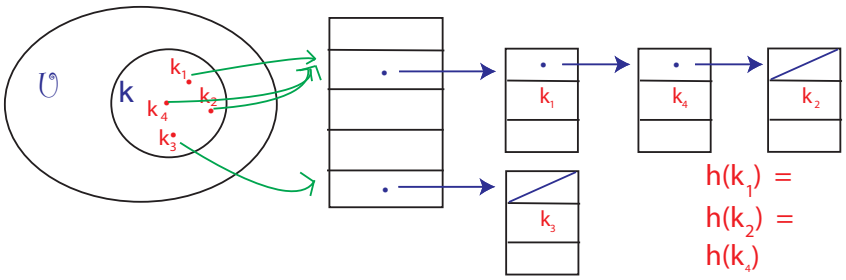


Figure 3: Chaining in a Hash Table

- Search must go through *whole* list T[h(key)]
- Worst case: all n keys hash to same slot $\implies \Theta(n)$ per operation

Simple Uniform Hashing:

An assumption (cheating): Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.

$$\begin{aligned} \text{let } n &= \#\text{ keys stored in table} \\ m &= \#\text{ slots in table} \\ \underline{\text{load factor}}\ \alpha &= n/m = \text{ expected } \#\text{ keys per slot} = \text{expected length of a chain} \end{aligned}$$

Performance

This implies that expected running time for search is $\Theta(1+\alpha)$ — the 1 comes from applying the hash function and random access to the slot whereas the α comes from searching the list. This is equal to $O(1)$ if $\alpha = O(1)$, i.e., $m = \Omega(n)$.

Hash Functions

We cover three methods to achieve the above performance:

Division Method:

$$h(k) = k \bmod m$$

This is practical when m is prime but not too close to power of 2 or 10 (then just depending on low bits/digits).

But it is inconvenient to find a prime number, and division is slow.

Multiplication Method:

$$h(k) = [(a \cdot k) \bmod 2^w] \gg (w - r)$$

where a is random, k is w bits, and $m = 2^r$.
This is practical when a is odd & $2^{w-1} < a < 2^w$ & a not too close to 2^{w-1} or 2^w.

Multiplication and bit extraction are faster than division.

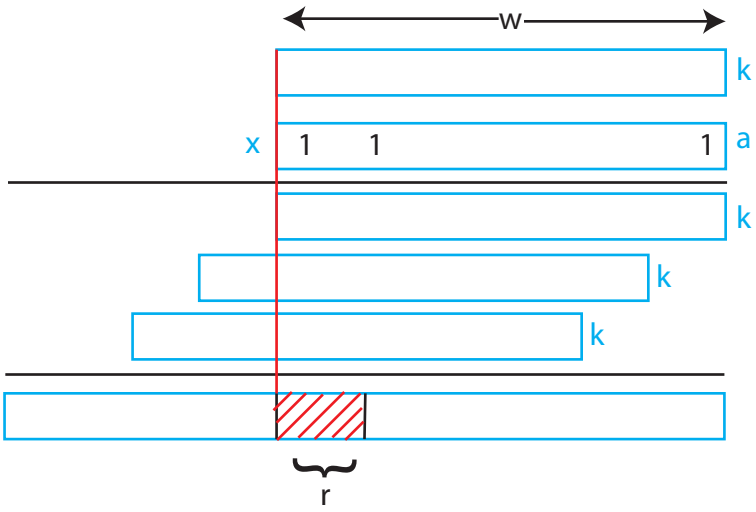


Figure 4: Multiplication Method

Universal Hashing

[6.046; CLRS 11.3.3]

For example: $h(k) = [(ak+b) \mod p] \mod m$ where a and b are random $\in \{0, 1, \ldots p-1\}$, and p is a large prime ($> |\mathcal{U}|$).
This implies that for *worst case* keys $k_1 \neq k_2$, (and for a, b choice of h):

$$Pr_{a,b}\{\text{event } X_{k_1k_2}\} = Pr_{a,b}\{h(k_1) = h(k_2)\} = \frac{1}{m}$$

This lemma not proved here
This implies that:

$$\begin{aligned} E_{a,b}[\#\text{ collisions with } k_1] &= E[\sum_{k_2} X_{k_1k_2}] \\ &= \sum_{k_2} E[X_{k_1k_2}] \\ &= \sum_{k_2} \underbrace{Pr\{X_{k_1k_2} = 1\}}_{\frac{1}{m}} \\ &= \frac{n}{m} = \alpha \end{aligned}$$

This is just as good as above!

Lecture 9: Hashing II

Lecture Overview

- Table Resizing
- Amortization
- String Matching and Karp-Rabin
- Rolling Hash

Recall:

Hashing with Chaining:

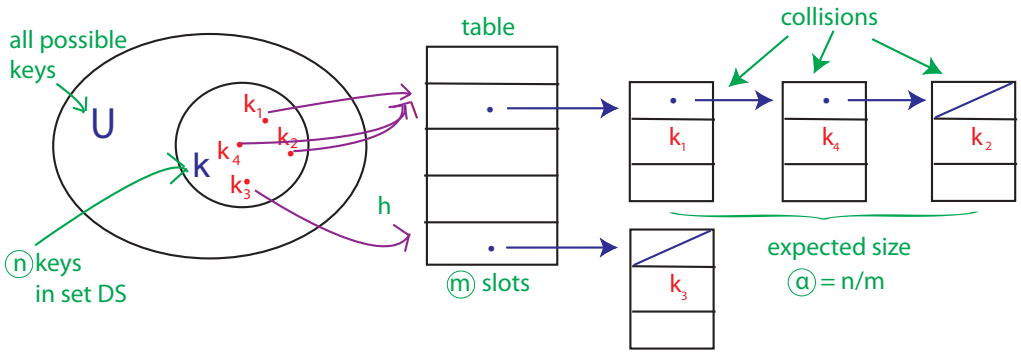


Figure 1: Hashing with Chaining

Expected cost (insert/delete/search): $\Theta(1+\alpha)$, assuming simple uniform hashing *OR* universal hashing & hash function h takes $O(1)$ time.

Division Method:

$$h(k) = k \mod m$$

where m is ideally prime

Multiplication Method:

$$h(k) = [(a \cdot k) \bmod 2^w] \gg (w - r)$$

where a is a random odd integer between 2^{w-1} and 2^w, k is given by w bits, and m = table size $= 2^r$.

How Large should Table be?

- want $m = \Theta(n)$ at all times
- don't know how large n will get at creation
- m too small $\implies$ slow; m too big $\implies$ wasteful

Idea:

Start small (constant) and grow (or shrink) as necessary.

Rehashing:

To grow or shrink table hash function must change (m, r)

$\implies$ must rebuild hash table from scratch
for item in old table: $\rightarrow$ for each slot, for item in slot
insert into new table
$\implies$ $\Theta(n + m)$ time $= \Theta(n)$ if $m = \Theta(n)$

How fast to grow?

When n reaches m, say

- $m+ =1$?
  $\implies$ rebuild every step
  $\implies$ n inserts cost $\Theta(1 + 2 + \cdots + n) = \Theta(n^2)$
- $m* =2$? $m = \Theta(n)$ still $(r+ =1)$
  $\implies$ rebuild at insertion 2^i
  $\implies$ n inserts cost $\Theta(1 + 2 + 4 + 8 + \cdots + n)$ where n is really the next power of 2
  $= \Theta(n)$
- a few inserts cost linear time, but $\Theta(1)$ "on average".

Amortized Analysis

This is a common technique in data structures — like paying rent: \$1500/month $\approx$ \$50/day

- operation has <u>amortized cost</u> $T(n)$ if k operations cost $\leq k \cdot T(n)$
- "$T(n)$ amortized" roughly means $T(n)$ "on average", but averaged over all ops.
- e.g. inserting into a hash table takes $O(1)$ amortized time.

Back to Hashing:

Maintain $m = \Theta(n) \implies \alpha = \Theta(1) \implies$ support search in $O(1)$ expected time (assuming simple uniform or universal hashing)

Delete:

Also $O(1)$ expected as is.

- space can get big with respect to n e.g. $n\times$ insert, $n\times$ delete
- solution: when n decreases to $m/4$, shrink to half the size $\implies O(1)$ amortized cost for both insert and delete — analysis is harder; see CLRS 17.4.

Resizable Arrays:

- same trick solves Python "list" (array)
- $\implies$ list.append and list.pop in $O(1)$ amortized

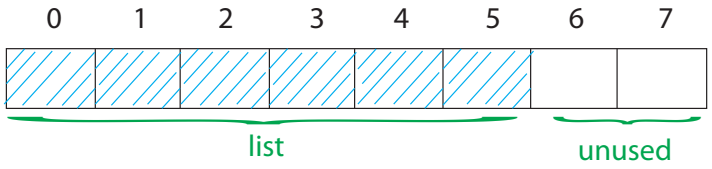


Figure 2: Resizeable Arrays

String Matching

Given two strings s and t, does s occur as a substring of t? (and if so, where and how many times?)
E.g. $s =$ '6.006' and $t =$ your entire INBOX ('grep' on UNIX)

Simple Algorithm:

any($s == t[i : i + \text{len(s)}]$ for i in range(len(t) − len(s)))
— $O(|s|)$ time for each substring comparison
$\implies O(|s| \cdot (|t| - |s|))$ time
$= O(|s| \cdot |t|)$ potentially quadratic

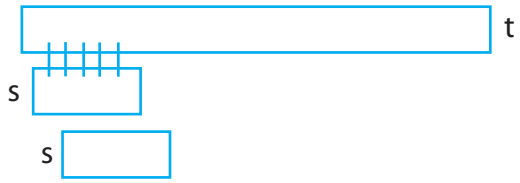


Figure 3: Illustration of Simple Algorithm for the String Matching Problem

Karp-Rabin Algorithm:

- Compare $h(s) == h(t[i : i + \text{len(s)}])$
- If hash values match, likely so do strings
  - can check $s == t[i : i + \text{len(s)}]$ to be sure $\sim$ cost $O(|s|)$
  - if yes, found match — done
  - if no, happened with probability $< \frac{1}{|s|}$
    $\implies$ expected cost is $O(1)$ per i.
- need suitable hash function.
- expected time $= O(|s| + |t| \cdot \text{cost(h)})$.
  - naively $h(x)$ costs $|x|$
  - we'll achieve $O(1)$!
  - idea: $t[i : i + len(s)] \approx t[i + 1 : i + 1 + len(s)]$.

Rolling Hash ADT

Maintain string x subject to

- $r()$: reasonable hash function $h(x)$ on string x
- r.append(c): add letter c to end of string x
- r.skip(c): remove front letter from string x, assuming it is c

Karp-Rabin Application:

```
for c in s: rs.append(c)
for c in t[:len(s)]: rt.append(c)
if rs() == rt(): ...
```

This first block of code is $O(|s|)$

```
for i in range(len(s), len(t)):
   rt.skip(t[i-len(s)])
   rt.append(t[i])
   if rs() == rt(): ...
```

The second block of code is $O(|t|) + O(\#\,\text{matches} - |s|)$ to verify.

Data Structure:

Treat string x as a multidigit number u in base a where a denotes the alphabet size, e.g., 256

- $r() = u \bmod p$ for (ideally random) prime $p \approx |s|$ or $|t|$ (division method)
- r stores $u \bmod p$ and $|x|$ (really $a^{|x|}$), not u

  $\implies$ smaller and faster to work with ($u \bmod p$ fits in one machine word)
- r.append(c): $(u \cdot a + \text{ord}(c)) \bmod p = [(u \bmod p) \cdot a + \text{ord}(c)] \bmod p$
- r.skip(c): $[u - \text{ord}(c) \cdot (a^{|u|-1} \bmod p)] \bmod p$

  $= [(u \bmod p) - \text{ord}(c) \cdot (a^{|x-1|} \bmod p)] \bmod p$

Lecture 10: Hashing III: Open Addressing

Lecture Overview

- Open Addressing, Probing Strategies
- Uniform Hashing, Analysis
- Cryptographic Hashing

Readings

CLRS Chapter 11.4 (and 11.3.3 and 11.5 if interested)

Open Addressing

Another approach to collisions:

- no chaining; instead all items stored in table (see Fig. 1)

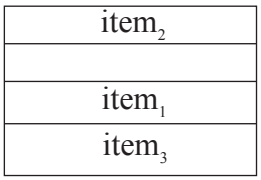


Figure 1: Open Addressing Table

- one item per slot $\implies m \geq n$
- hash function specifies *order* of slots to probe (try) for a key (for insert/search/delete), not just one slot; **in math. notation**:

  We want to design a function h, with the property that for all $k \in \mathcal{U}$:

$$h : \mathcal{U} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

universe of keys trial count slot in table

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

is a permutation of $0, 1, \ldots, m-1$. i.e. if I keep trying $h(k,i)$ for increasing i, I will eventually hit all slots of the table.

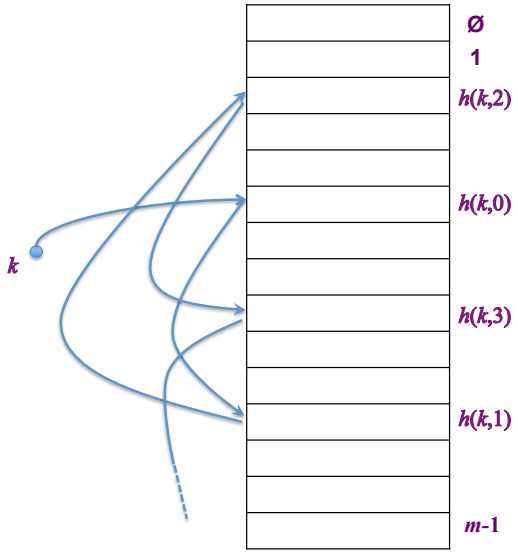


Figure 2: Order of Probes

Insert(k,v) : Keep probing until an empty slot is found. Insert item into that slot.

```
for i in xrange(m):
    if T[h(k,i)] is None:          # empty slot
        T[h(k,i)] = (k,v)          # store item
        return
raise 'full'
```

Example: Insert $k = 496$

Search(k): As long as the slots you encounter by probing are occupied by keys $\neq k$, keep probing until you either encounter k or find an empty slot—return *success* or *failure* respectively.

```
for i in xrange(m):
    if   T[h(k,i)] is None:        # empty slot?
         return None               # end of "chain"
    elif T[h(k,i)][Ø] == k:        # matching key
         return T[h(k,i)]          # return item
return None                        # exhausted table
```

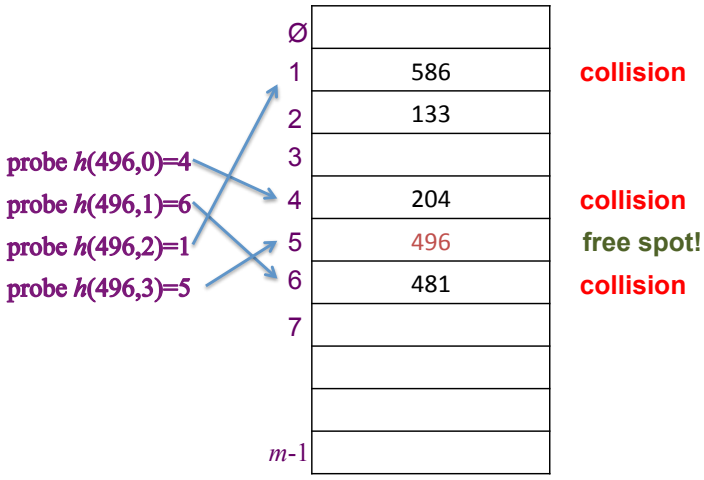


Figure 3: Insert Example

Deleting Items?

- can't just find item and remove it from its slot (i.e. set $T[h(k,i)] = \text{None}$)
- *example*: delete(586) $\implies$ search(496) fails
- replace item with special flag: "DeleteMe", which Insert treats as None but Search doesn't

Probing Strategies

Linear Probing

$h(k,i) = (h'(k) + i) \bmod m$ where $h'(k)$ is ordinary hash function

- like street parking
- **problem?** *clustering*—cluster: consecutive group of occupied slots
  as clusters become longer, it gets *more* likely to grow further (see Fig. 4)
- can be shown that for $0.01 < \alpha < 0.99$ say, clusters of size $\Theta(\log n)$.

Double Hashing

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ where $h_1(k)$ and $h_2(k)$ are two ordinary hash functions.

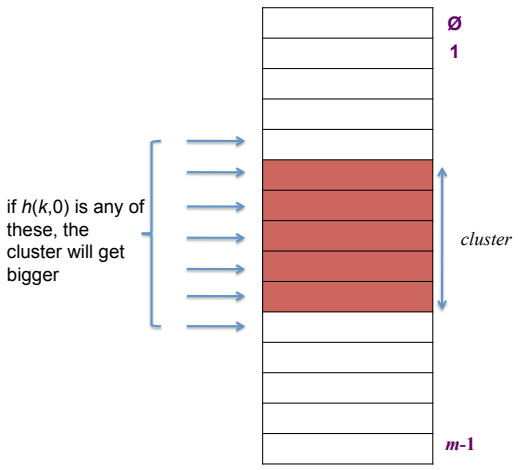


Figure 4: Primary Clustering

- actually hit all slots (permutation) if $h_2(k)$ is relatively prime to m for all k **why?**

$$h_1(k) + i \cdot h_2(k) \bmod m = h_1(k) + j \cdot h_2(k) \bmod m \Rightarrow m \textit{ divides } (i - j)$$

- e.g. $m = 2^r$, make $h_2(k)$ always odd

Uniform Hashing Assumption (cf. Simple Uniform Hashing Assumption)

Each key is equally likely to have any one of the $m!$ permutations as its probe sequence

- not really true
- but double hashing can come close

Analysis

Suppose we have used open addressing to insert n items into table of size m. Under the uniform hashing assumption the next operation has expected cost of $\leq \frac{1}{1-\alpha}$, where $\alpha = n/m (< 1)$.
Example: $\alpha = 90\% \implies$ 10 expected probes

Proof:

Suppose we want to insert an item with key k. Suppose that the item is not in the table.

- probability first probe successful: $\frac{m-n}{m} =: p$
  (n bad slots, m total slots, and first probe is uniformly random)
- if first probe fails, probability second probe successful: $\frac{m-n}{m-1} \geq \frac{m-n}{m} = p$
  (one bad slot already found, $m-n$ good slots remain and the second probe is uniformly random over the $m-1$ total slots left)
- if 1st & 2nd probe fail, probability 3rd probe successful: $\frac{m-n}{m-2} \geq \frac{m-n}{m} = p$
  (since two bad slots already found, $m-n$ good slots remain and the third probe is uniformly random over the $m-2$ total slots left)
- ...

$\Rightarrow$ Every trial, success with probability at least p.
Expected Number of trials for success?

$$\frac{1}{p} = \frac{1}{1-\alpha}.$$

With a little thought it follows that search, delete take time $O(1/(1-\alpha))$. Ditto if we attempt to insert an item that is already there.■

Open Addressing vs. Chaining

Open Addressing: better cache performance (better memory usage, no pointers needed)

Chaining: less sensitive to hash functions (OA requires extra care to avoid clustering) and the load factor α (OA degrades past 70% or so and in any event cannot support values larger than 1)

Cryptographic Hashing

A cryptographic hash function is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, the (cryptographic) hash value, such that an accidental or intentional change to the data will change the hash value. The data to be encoded is often called the *message*, and the hash value is sometimes called the *message digest* or simply digest.

The ideal cryptographic hash function has the properties listed below. d is the number of bits in the output of the hash function. You can think of m as being 2^d. d is typically 160 or more. These hash functions can be used to index hash tables, but they are typically used in computer security applications.

Desirable Properties

1. **One-Way (OW)**: Infeasible, given $y \in_R \{0,1\}^d$ to find any x s.t. $h(x) = y$. This means that if you choose a random d-bit vector, it is hard to find an input to the hash that produces that vector. This involves "inverting" the hash function.

2. **Collision-resistance (CR)**: Infeasible to find x, x', s.t. $x \neq x'$ and $h(x) = h(x')$. This is a collision, two input values have the same hash.

3. **Target collision-resistance (TCR)**: Infeasible given x to find $x' = x$ s.t. $h(x) = h(x')$.

TCR is weaker than CR. If a hash function satisfies CR, it automatically satisfies TCR. There is no implication relationship between OW and CR/TCR.

Applications

1. **Password storage**: Store $h(PW)$, not PW on computer. When user inputs PW', compute $h(PW')$ and compare against $h(PW)$. The property required of the hash function is OW. The adversary does not know PW or PW' so TCR or CR is not really required. Of course, if many, many passwords have the same hash, it is a problem, but a small number of collisions doesn't really affect security.

2. **File modification detector**: For each file F, store $h(F)$ securely. Check if F is modified by recomputing $h(F)$. The property that is required is TCR, since the adversary wins if he/she is able to modify F without changing $h(F)$.

3. **Digital signatures**: In public-key cryptography, Alice has a public key PK_A and a private key SK_A. Alice can sign a message M using her private key to produce $\sigma = sign(SK_A, M)$. Anyone who knows Alice's public key PK_A and verify Alice's signature by checking that $verify(M, \sigma, PK_A)$ is true. The adversary wants to forge a signature that verifies. For large M it is easier to sign $h(M)$ rather than M, i.e., $\sigma = sign(SK_A, h(M))$. The property that we

require is CR. We don't want an adversary to ask Alice to sign x and then claim that she signed x', where $h(x) = h(x')$.

Implementations

There have been many proposals for hash functions which are OW, CR and TCR. Some of these have been broken. MD-5, for example, has been shown to not be CR. There is a competition underway to determine SHA-3, which would be a Secure Hash Algorithm certified by NIST. Cryptographic hash functions are significantly more complex than those used in hash tables. You can think of a cryptographic hash as running a regular hash function many, many times with pseudo-random permutations interspersed.

Lecture 11: Numerics I

Lecture Overview

- Irrationals
- Newton's Method ($\sqrt{(a)}, 1/b$)
- High precision multiply ←

Irrationals:

Pythagoras discovered that a square's diagonal and its side are incommensurable, i.e., could not be expressed as a ratio - he called the ratio "speechless"!

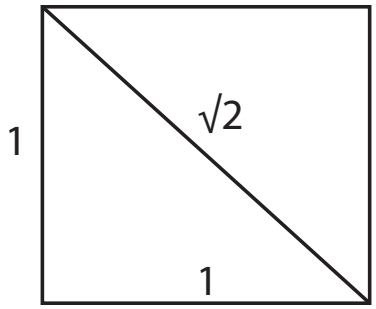


Figure 1: Ratio of a Square's Diagonal to its Sides.

Pythagoras worshipped numbers
"All is number"
Irrationals were a threat!

Motivating Question: Are there hidden patterns in irrationals?

$$\sqrt{2} = 1.\ 414\ 213\ 562\ 373\ 095$$
$$048\ 801\ 688\ 724\ 209$$
$$698\ 078\ 569\ 671\ 875$$

Can you see a pattern?

Digression

Catalan numbers:
Set P of balanced parentheses strings are recursively defined as

- $\lambda \in P$ (λ is empty string)
- If $\alpha, \beta \in P$, then $(\alpha)\beta \in P$

Every nonempty balanced paren string can be obtained via Rule 2 from a unique α, β pair.
For example, (()) ()() obtained by ($\underbrace{()}_{\alpha}$) $\underbrace{()()}_{\beta}$

Enumeration

C_n: number of balanced parentheses strings with exactly n pairs of parentheses
$C_0 = 1$ empty string

C_{n+1}? Every string with $n+1$ pairs of parentheses can be obtained in a unique way via rule 2.

One paren pair comes explicitly from the rule.
k pairs from α, $n-k$ pairs from β

$$C_{n+1} = \sum_{k=0}^{n} C_k \cdot C_{n-k} \quad n \geq 0$$

$$C_0 = 1 \quad C_1 = {C_0}^2 = 1 \quad C_2 = C_0C_1 + C_1C_0 = 2 \quad C_3 = \cdots = 5$$

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, 18367353072152, 69533550916004, 263747951750360, 1002242216651368

Newton's Method

Find root of $f(x) = 0$ through successive approximation e.g., $f(x) = x^2 - a$

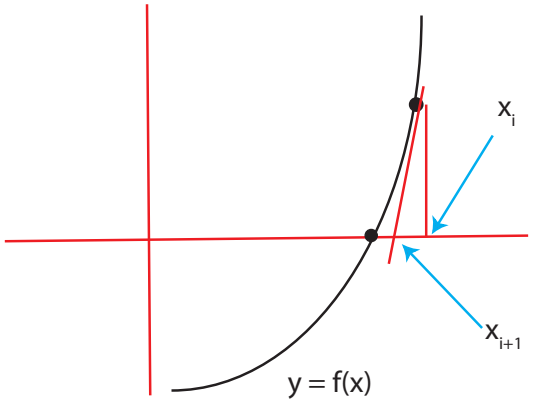


Figure 2: Newton's Method.

Tangent at $(x_i, f(x_i))$ is line $y = f(x_i) + f'(x_i) \cdot (x - x_i)$ where $f'(x_i)$ is the derivative. x_{i+1} = intercept on x-axis

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}}$$

Square Roots

$$f(x) = x^2 - a$$

$$\chi_{i+1} = \chi_i - \frac{({\chi_i}^2 - a)}{2\chi_i} = \frac{\chi_i + \frac{a}{\chi_i}}{2}$$

Example

$$\begin{aligned} \chi_0 &= 1.000000000 \qquad a = 2 \\ \chi_1 &= 1.500000000 \\ \chi_2 &= 1.416666666 \\ \chi_3 &= 1.414215686 \\ \chi_4 &= 1.414213562 \end{aligned}$$

Quadratic convergence, ♯ digits doubles. Of course, in order to use Newton's method, we need high-precision division. We'll start with multiplication and cover division in Lecture 12.

High Precision Computation

$\sqrt{2}$ to d-digit precision: $\underbrace{1.414213562373}_{\text{d digits}}\cdots$

Want integer $\lfloor 10^d\sqrt{2}\rfloor = \lfloor\sqrt{2\cdot 10^{2d}}\rfloor$ - integral part of square root
Can still use Newton's Method.

High Precision Multiplication

Multiplying two n-digit numbers (radix $r = 2, 10$)
$0 \le x, y < r^n$

$$\begin{aligned} x &= x_1\cdot r^{n/2} + x_0 \qquad x_1 = \text{ high half} \\ y &= y_1\cdot r^{n/2} + y_0 \qquad x_0 = \text{ low half} \\ 0 &\le x_0, x_1 < r^{n/2} \\ 0 &\le y_0, y_1 < r^{n/2} \end{aligned}$$

$$z = x\cdot y = x_1y_1\cdot r^n + (x_0\cdot y_1 + x_1\cdot y_0)r^{n/2} + x_0\cdot y_0$$

4 multiplications of half-sized ♯'s $\implies$ quadratic algorithm $\theta(n^2)$ time

Karatsuba's Method

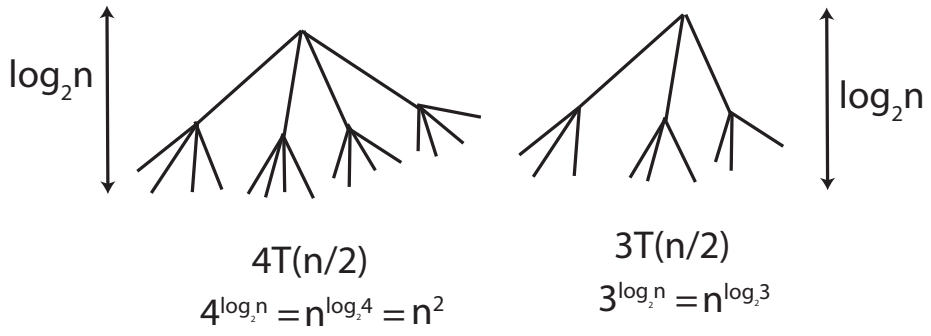


Figure 3: Branching Factors.

Let

$$\begin{aligned} z_0 &= x_0 \cdot y_0 \\ z_2 &= x_1 \cdot y_1 \\ z_1 &= (x_0 + x_1) \cdot (y_0 + y_1) - z_0 - z_2 \\ &= x_0 y_1 + x_1 y_0 \\ z &= z_2 \cdot r^n + z_1 \cdot r^{n/2} + z_0 \end{aligned}$$

There are three multiplies in the above calculations.

$$\begin{aligned} T(n) &= \text{time to multiply two } n\text{-digit} \sharp's \\ &= 3T(n/2) + \theta(n) \\ &= \theta\left(n^{\log_2 3}\right) = \theta\left(n^{1.5849625\cdots}\right) \end{aligned}$$

This is better than $\theta(n^2)$. Python does this, and more (see Lecture 12).

Fun Geometry Problem

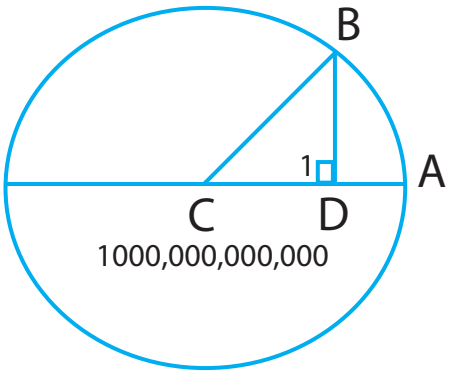


Figure 4: Geometry Problem.

$BD = 1$
What is AD?

$$AD = AC - CD = 500,000,000,000 - \sqrt{\underbrace{500,000,000,000^2 - 1}_{a}}$$

Let's calculate AD to a million places. (This assumes we have high-precision division, which we will cover in Lecture 12.) Remarkably, if we evaluate the length

to several hundred digits of precision using Newton's method, the Catalan numbers come marching out! Try it at:
`http://people.csail.mit.edu/devadas/numerics_demo/chord.html`.

An Explanation

This was *not* covered in lecture and will *not* be on a test. Let's start by looking at the power series of a real-valued function Q.

$$Q(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \ldots \quad (1)$$

Then, by ordinary algebra, we have:

$$1 + xQ(x)^2 = 1 + c_0^2 x + (c_0 c_1 + c_1 c_0)x^2 + (c_0 c_2 + c_1 c_1 + c_2 c_0)x^3 + \ldots \quad (2)$$

Now consider the equation:

$$Q(x) = 1 + xQ(x)^2 \quad (3)$$

For this equation to hold, the power series of $Q(x)$ must equal the power series of $1 + xQ(x)^2$. This happens only if all the coefficients of the two power series are equal; that is, if:

$$\begin{aligned} c_0 &= 1 && (4)\\ c_1 &= c_0^2 && (5)\\ c_2 &= c_0 c_1 + c_1 c_0 && (6)\\ c_3 &= c_0 c_2 + c_1 c_1 + c_2 c_0 && (7)\\ &\text{etc.} && (8) \end{aligned}$$

In other words, the coefficients of the function Q must be the Catalan numbers!

We can solve for Q using the quadratic equation:

$$Q(x) = \frac{1 \pm \sqrt{1 - 4x}}{2x} \quad (9)$$

Let's use the negative square root. From this formula for Q, we find:

$$10^{-12} \cdot Q(10^{-24}) = 10^{-12} \cdot \frac{1 \pm \sqrt{1 - 4 \cdot 10^{-24}}}{2 \cdot 10^{-24}} \quad (10)$$

$$= 500000000000 - \sqrt{500000000000^2 - 1} \quad (11)$$

From the original power-series expression for Q, we find:

$$10^{-12} \cdot Q(10^{-24}) = c_0 10^{-12} + c_1 10^{-36} + c_2 10^{-60} + c_3 10^{-84} + \ldots \quad (12)$$

Therefore, $500000000000 - \sqrt{500000000000^2 - 1}$ should contain a Catalan number in every twenty-fourth position, which is what we observed.

Lecture 12: Numerics II

Lecture Overview

- Review:
  - high precision arithmetic
  - multiplication
- Division
  - Algorithm
  - Error Analysis
- Termination

Review:

Want millionth digit of $\sqrt{2}$:

$$\lfloor\sqrt{2 \cdot 10^{2d}}\rfloor \qquad d = 10^6$$

Compute $\lfloor\sqrt{a}\rfloor$ via Newton's Method

$$\begin{aligned} \chi_0 &= 1 \quad \text{(initial guess)} \\ \chi_{i+1} &= \frac{\chi_i + a/\chi_i}{2} \quad \leftarrow \text{division!} \end{aligned}$$

Error Analysis of Newton's Method

Suppose $X_n = \sqrt{a} \cdot (1 + \epsilon_n)$ ϵ_n may be + or -
Then,

$$\begin{aligned} X_{n+1} &= \frac{X_n + a/X_n}{2} \\ &= \frac{\sqrt{a}(1+\epsilon_n) + \frac{a}{\sqrt{a}(1+\epsilon_n)}}{2} \\ &= \sqrt{(a)}\frac{\left((1+\epsilon_n) + \frac{1}{(1+\epsilon_n)}\right)}{2} \\ &= \sqrt{(a)}\left(\frac{2 + 2\epsilon_n + {\epsilon_n}^2}{2(1+\epsilon_n)}\right) \\ &= \sqrt{(a)}\left(1 + \frac{{\epsilon_n}^2}{2(1+\epsilon_n)}\right) \end{aligned}$$

Therefore,

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

Quadratic convergence, as ♯ correct digits doubles each step.

Newton's method requires high-precision division. We covered multiplication in Lecture 12.

Multiplication Algorithms:

1. Naive Divide & Conquer method: $\Theta(d^2)$ time
2. Karatsuba: $\Theta(d^{\log_2 3}) = \Theta(d^{1.584\ldots})$
3. Toom-Cook generalizes Karatsuba (break into $k \geq 2$ parts)

$$T(d) = 5T(d/3) + \Theta(d) = \Theta\left(d^{\log_3 5}\right) = \Theta\left(d^{1.465\ldots}\right)$$

4. Schönhage-Strassen - almost linear! $\Theta(d \lg d \lg\lg d)$ using FFT. All of these are in gmpy package
5. Furer (2007): $\Theta\left(n \log n \; 2^{O(\log^* n)}\right)$ where $\log^* n$ is iterated logarithm. ♯ times log needs to be applied to get a number that is less than or equal to 1.

High Precision Division

We want high precision rep of $\frac{a}{b}$

- Compute high-precision rep of $\frac{1}{b}$ first
- High-precision rep of $\frac{1}{b}$ means $\lfloor \frac{R}{b} \rfloor$ where R is large value s.t. it is easy to divide by R
  Ex: $R = 2^k$ for binary representations

Division

Newton's Method for computing $\frac{R}{b}$

$$\begin{aligned} f(x) &= \frac{1}{x} - \frac{b}{R} \quad \left(\text{zero at } x = \frac{R}{b}\right) \\ f'(x) &= \frac{-1}{x^2} \\ \chi_{i+1} &= \chi_i - \frac{f(\chi_i)}{f'(\chi_i)} = \chi_i - \frac{\left(\frac{1}{\chi_i} - \frac{b}{R}\right)}{-1/{\chi_i}^2} \\ \chi_{i+1} &= \chi_i + {\chi_i}^2\left(\frac{1}{\chi_i} - \frac{b}{R}\right) = 2\chi_i - \frac{b{\chi_i}^2 \to \text{ multiply}}{R \to \text{ easy div}} \end{aligned}$$

Example

Want $\frac{R}{b} = \frac{2^{16}}{5} = \frac{65536}{5} = 13107.2$

Try initial guess $\frac{2^{16}}{4} = 2^{14}$

$$\begin{aligned} \chi_0 &= 2^{14} = 16384 \\ \chi_1 &= 2 \cdot (16384) - 5(16384)^2/65536 = \underline{12}288 \\ \chi_2 &= 2 \cdot (12288) - 5(12288)^2/65536 = \underline{13}056 \\ \chi_3 &= 2 \cdot (13056) - 5(13056)^2/65536 = \underline{13107} \end{aligned}$$

Error Analysis

$$\begin{aligned} \chi_{i+1} &= 2\chi_i - \frac{b{\chi_i}^2}{R} \qquad \text{Assume } \chi_i = \frac{R}{b}(1+\epsilon_i) \\ &= 2\frac{R}{b}(1+\epsilon_i) - \frac{b}{R}\left(\frac{R}{b}\right)^2(1+\epsilon_i)^2 \\ &= \frac{R}{b}\left((2+2\epsilon_i) - (1+2\epsilon_i+{\epsilon_i}^2)\right) \\ &= \frac{R}{b}\left(1-{\epsilon_i}^2\right) = \frac{R}{b}(1+\epsilon_{i+1}) \text{ where } \epsilon_{i+1} = -{\epsilon_i}^2 \end{aligned}$$

Quadratic convergence; ♯ digits doubles at each step

One might think that the complexity of division is $\lg d$ times the complexity of multiplication given that we will have $\lg d$ multiplications in the $\lg d$ iterations required

to reach precision d. However, the complexity of division equals the complexity of multiplication.

To understand this, assume that the complexity of multiplication is $\Theta(n^\alpha)$ for n-digit numbers, with $\alpha \geq 1$. Division requires multiplication of *different-sized* numbers at each iteration. Initially the numbers are small, and then they grow to d digits. The number of operations in division are:

$$c \cdot 1^\alpha \ + \ c \cdot 2^\alpha \ + \ c \cdot 4^\alpha \ + \ \cdots \ + \ c \cdot \left(\frac{d}{4}\right)^\alpha \ + \ c \cdot \left(\frac{d}{2}\right)^\alpha \ + \ c \cdot d^\alpha \ < \ 2c \cdot d^\alpha$$

Complexity of Computing Square Roots

We apply a first level of Newton's method to solve $f(x) = x^2 - a$. Each iteration of this first level[1] requires a division. If we set the precision to d digits right from the beginning, then convergence at the first level will require $\lg d$ iterations. This means the complexity of computing a square root will be $\Theta(d^\alpha \lg d)$ if the complexity of multiplication is $\Theta(d^\alpha)$, given that we have shown that the complexity of division is the same as the complexity of multiplication.

However, we can do better, if we recognize that the number of digits of precision we need at beginning of the first level of Newton's method starts out small and then grows. If the complexity of a d-digit division is $\Theta(d^\alpha)$, then a similar summation to the one above tells us that the complexity of computing square roots is $\Theta(d^\alpha)$.

Termination

Iteration: $\chi_{i+1} = \lfloor \frac{\chi_i + \lfloor a/\chi_i \rfloor}{2} \rfloor$
Do floors hurt? Does program terminate? (α and β are the fractional parts below.)
Iteration is

$$\begin{aligned} \chi_{i+1} &= \frac{\chi_i + \frac{a}{\chi_i} - \alpha}{2} - \beta \\ &= \frac{\chi_i + \frac{a}{\chi_i}}{2} - \gamma \quad \text{where } \gamma = \frac{\alpha}{2} + \beta \text{ and } 0 \leq \gamma < 1 \end{aligned}$$

Since $\frac{a+b}{2} \geq \sqrt{ab}$, $\frac{\chi_i + \frac{a}{\chi_i}}{2} \geq \sqrt{a}$, so subtracting γ always leaves us $\geq \lfloor \sqrt{a} \rfloor$. This won't stay stuck above if $\epsilon_i < 1$ (good initial guess).

[1] We are calling this the first level, since Newton's method is used within division, which would be the second level of applying it when we are computing square roots.

Lecture 13: Graphs I: Breadth First Search

Lecture Overview

- Applications of Graph Search
- Graph Representations
- Breadth-First Search

Recall:

Graph $G = (V, E)$

- V = set of vertices (arbitrary labels)
- E = set of edges i.e. vertex pairs (v, w)
  - ordered pair $\implies$ directed edge of graph
  - unordered pair $\implies$ undirected

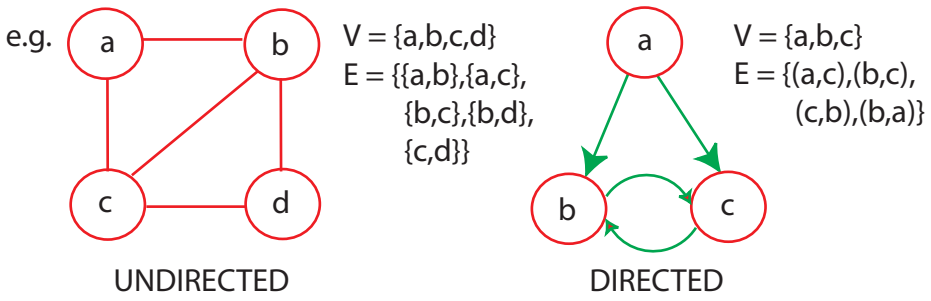


Figure 1: Example to illustrate graph terminology

Graph Search

"Explore a graph", e.g.:

- find a path from start vertex s to a desired vertex
- visit all vertices or edges of graph, or only those reachable from s

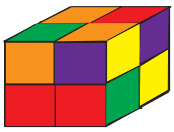
Applications:

There are many.

- web crawling (how Google finds pages)
- social networking (Facebook friend finder)
- network broadcast routing
- garbage collection
- model checking (finite state machine)
- checking mathematical conjectures
- solving puzzles and games

Pocket Cube:

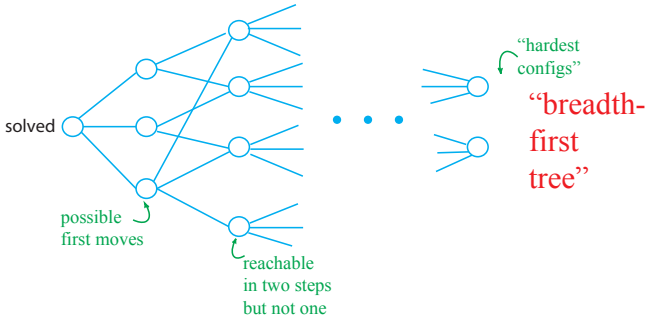
Consider a $2 \times 2 \times 2$ Rubik's cube



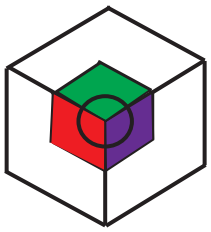
Configuration Graph:

- vertex for each possible state
- edge for each basic move (e.g., 90 degree turn) from one state to another
- undirected: moves are reversible

Diameter ("God's Number")
11 for $2 \times 2 \times 2$, 20 for $3 \times 3 \times 3$, $\Theta(n^2/\lg n)$ for $n \times n \times n$ [Demaine, Demaine, Eisenstat Lubiw Winslow 2011]



vertices $= 8! \cdot 3^8 = 264,539,520$ where 8! comes from having 8 cubelets in arbitrary positions and 3^8 comes as each cubelet has 3 possible twists.



This can be divided by 24 if we remove cube symmetries and further divided by 3 to account for actually reachable configurations (there are 3 connected components).

Graph Representations: (data structures)

Adjacency lists:

Array Adj of $|V|$ linked lists

- for each vertex $u \in V$, $Adj[u]$ stores u's neighbors, i.e., $\{v \in V \mid (u, v) \in E\}$. (u, v) are just outgoing edges if directed. (See Fig. 2 for an example.)

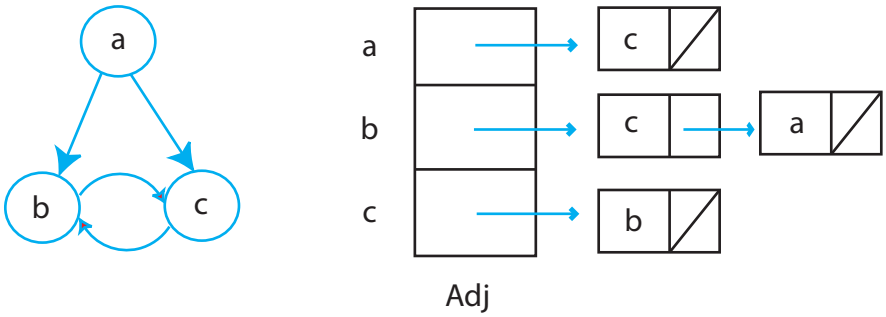


Figure 2: Adjacency List Representation: Space $\Theta(V + E)$

- in Python: Adj = dictionary of list/set values; vertex = any hashable object (e.g., int, tuple)
- advantage: multiple graphs on same vertices

Implicit Graphs:

Adj(u) is a function — compute local structure on the fly (e.g., Rubik's Cube). This requires "Zero" Space.

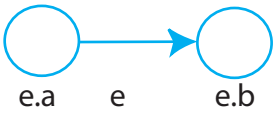
Object-oriented Variations:

- object for each vertex u
- u.neighbors = list of neighbors i.e. $Adj[u]$

In other words, this is method for implicit graphs

Incidence Lists:

- can also make edges objects



- u.edges = list of (outgoing) edges from u.
- advantage: store edge data without hashing

Breadth-First Search

Explore graph level by level from s

- level 0 = $\{s\}$
- level i = vertices reachable by path of i edges but not fewer

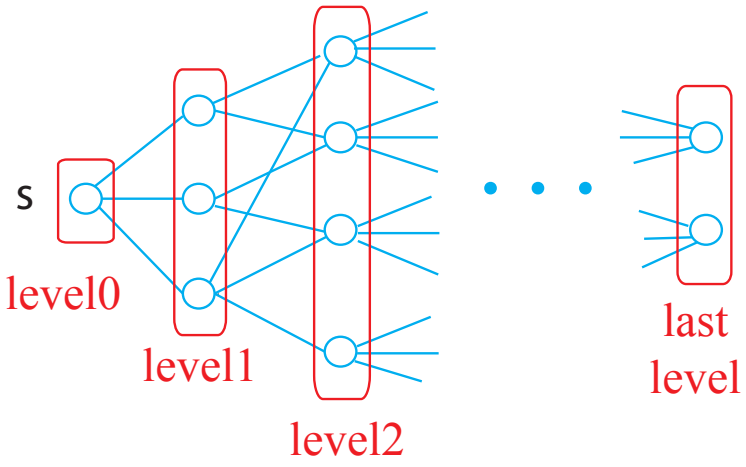


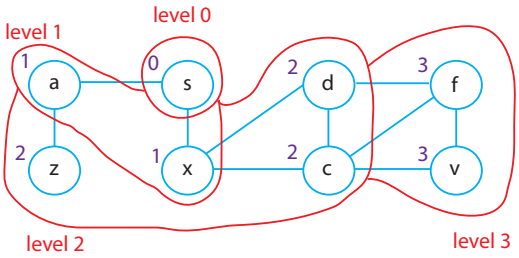
Figure 3: Illustrating Breadth-First Search

- build level $i > 0$ from level $i - 1$ by trying all outgoing edges, but ignoring vertices from previous levels

Breadth-First-Search Algorithm

```
BFS (V,Adj,s):                                  See CLRS for queue-based implementation
    level = { s: 0 }
    parent = {s : None }
    i = 1
    frontier = [s]                              # previous level, i − 1
    while frontier:
        next = [ ]                              # next level, i
        for u in frontier:
            for v in Adj[u]:
                if v not in level:              # not yet seen
                    level[v] = i                ♯ = level[u] + 1
                    parent[v] = u
                    next.append(v)
        frontier = next
        i + =1
```

Example



$\text{frontier}_0 = \{s\}$
$\text{frontier}_1 = \{a, x\}$
$\text{frontier}_2 = \{z, d, c\}$
$\text{frontier}_3 = \{f, v\}$
(not x, c, d)

Figure 4: Breadth-First Search Frontier

Analysis:

- vertex V enters next (& then frontier) only once (because level[v] then set)

  base case: $v = s$

- $\implies$ Adj[v] looped through only once

$$\text{time} = \sum_{v \in V} |Adj[V]| = \begin{cases} |E| \text{ for directed graphs} \\ 2|E| \text{ for undirected graphs} \end{cases}$$

- $\implies$ $O(E)$ time

- $O(V + E)$ ("LINEAR TIME") to also list vertices unreachable from v (those still not assigned level)

Shortest Paths:

cf. L15-18

- for every vertex v, fewest edges to get from s to v is

$$\begin{cases} \text{level}[v] \text{ if } v \text{ assigned level} \\ \infty \quad \text{else (no path)} \end{cases}$$

- parent pointers form shortest-path tree = union of such a shortest path for each v
  $\implies$ to find shortest path, take v, parent[v], parent[parent[v]], etc., until s (or None)

Lecture 14: Graphs II: Depth-First Search

Lecture Overview

- Depth-First Search
- Edge Classification
- Cycle Testing
- Topological Sort

Recall:

- graph search: explore a graph
  e.g., find a path from start vertex s to a desired vertex
- adjacency lists: array Adj of $|V|$ linked lists
  - for each vertex $u \in V$, Adj[u] stores u's neighbors, i.e., $\{v \in V \mid (u, v) \in E\}$
    (just outgoing edges if directed)

For example:

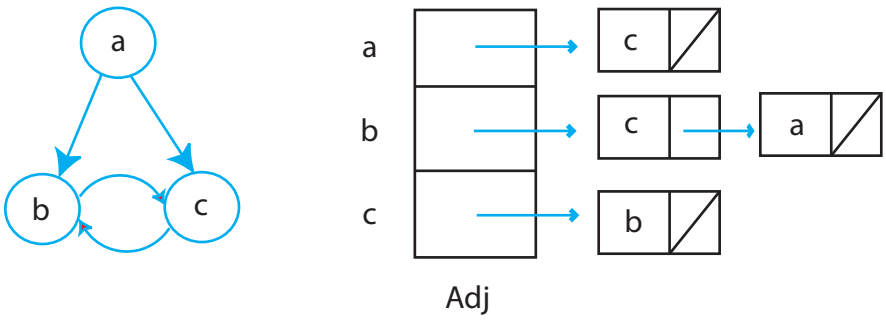


Figure 1: Adjacency Lists

Breadth-first Search (BFS):

Explore level-by-level from s — find shortest paths

Depth-First Search (DFS)

This is like exploring a maze.

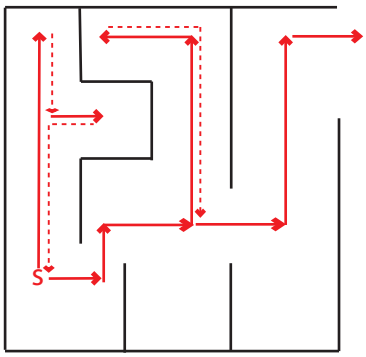


Figure 2: Depth-First Search Frontier

Depth First Search Algorithm

- follow path until you get stuck
- backtrack along breadcrumbs until reach unexplored neighbor
- recursively explore
- careful not to repeat a vertex

```
parent = {s: None}

DFS-visit (V, Adj, s):
start → for v in Adj [s]:
  v             if v not in parent:
                    parent [v] = s
finish →            DFS-visit (V, Adj, v)
  v

DFS (V, Adj)
    parent = { }
    for s in V:
        if s not in parent:
            parent [s] = None
            DFS-visit (V, Adj, s)
```

search from start vertex s (only see stuff reachable from s)

explore entire graph

(could do same to extend BFS)

Figure 3: Depth-First Search Algorithm

Example

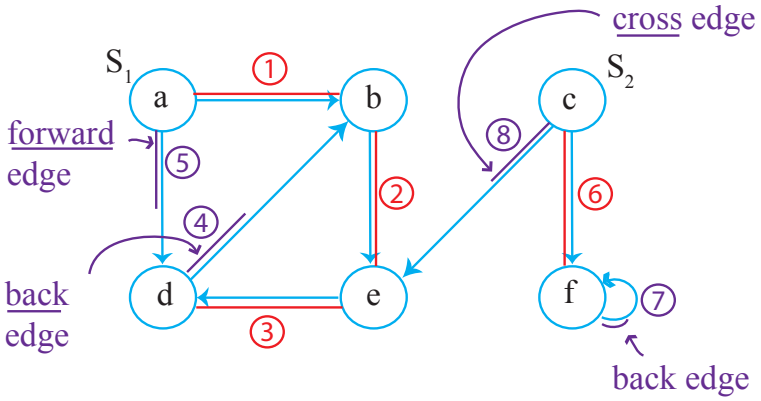


Figure 4: Depth-First Traversal

Edge Classification

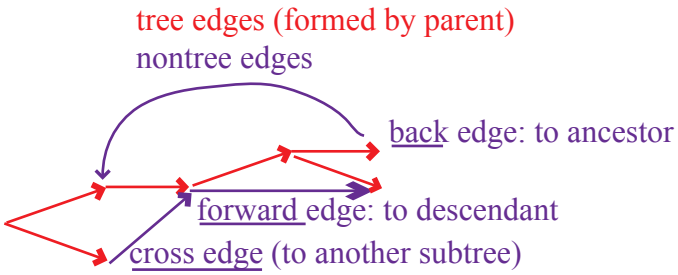


Figure 5: Edge Classification

- to compute this classification (back or not), mark nodes for duration they are "on the stack"
- only tree and back edges in undirected graph

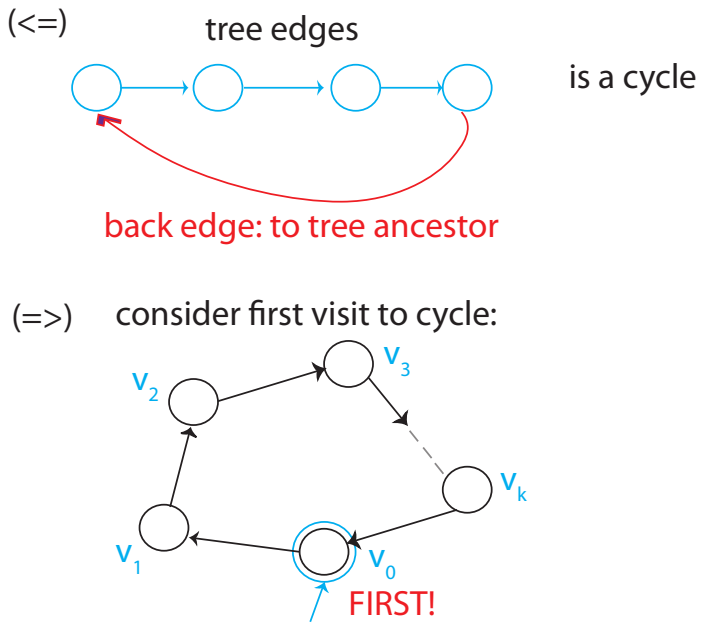
Analysis

- DFS-visit gets called with a vertex s only once (because then parent[s] set)
  $\implies$ time in DFS-visit $= \sum_{s \in V} |\text{Adj}[s]| = O(E)$
- DFS outer loop adds just $O(V)$
  $\implies$ $O(V + E)$ time (linear time)

Cycle Detection

Graph G has a cycle $\Leftrightarrow$ DFS has a back edge

Proof



- before visit to v_i finishes,
  will visit v_{i+1} (& finish):
  will consider edge (v_i, v_{i+1})
  $\Longrightarrow$ visit v_{i+1} now or already did
- $\Longrightarrow$ before visit to v_0 finishes,
  will visit v_k (& didn't before)
- $\Longrightarrow$ before visit to v_k (or v_0) finishes,
  will see (v_k, v_0) as back edge

Job scheduling

Given Directed Acylic Graph (DAG), where vertices represent tasks & edges represent dependencies, order tasks without violating dependencies

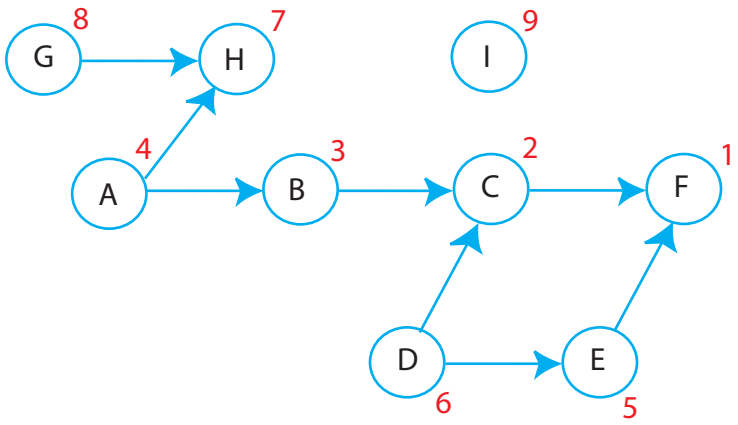


Figure 6: Dependence Graph: DFS Finishing Times

Source:

Source = vertex with no incoming edges
= schedulable at beginning (A,G,I)

Attempt:

BFS from each source:

- from A finds A, BH, C, F
- from D finds D, BE, CF ← slow . . . and wrong!
- from G finds G, H
- from I finds I

Topological Sort

Reverse of DFS finishing times (time at which DFS-Visit(v) finishes)

```
DFS-Visit(v)
    ...
    order.append(v)

order.reverse()
```

Correctness

For any edge (u, v) — u ordered before v, i.e., v finished before u

- if u visited before v:
  - before visit to u finishes, will visit v (via (u, v) or otherwise)
  - $\implies$ v finishes before u
- if v visited before u:
  - graph is acyclic
  - $\implies$ u cannot be reached from v
  - $\implies$ visit to v finishes before visiting u

Lecture 15: Shortest Paths I: Intro

Lecture Overview

- Weighted Graphs
- General Approach
- Negative Edges
- Optimal Substructure

Readings

CLRS, Sections 24 (Intro)

Motivation:

Shortest way to drive from A to B Google maps "get directions"

Formulation: Problem on a weighted graph $G(V, E) \quad W : E \to \Re$

Two algorithms: Dijkstra $O(V \lg V + E)$ assumes non-negative edge weights
Bellman Ford $O(VE)$ is a general algorithm

Application

- Find shortest path from CalTech to MIT
  - See "CalTech Cannon Hack" photos web.mit.edu
  - See Google Maps from CalTech to MIT
- Model as a weighted graph $G(V, E), W : E \to \Re$
  - V = vertices (street intersections)
  - E = edges (street, roads); directed edges (one way roads)
  - $W(U, V)$ = weight of edge from u to v (distance, toll)

$$\begin{aligned} \text{path } p &= < v_0, v_1, \ldots v_k > \\ & (v_i, v_{i+1}) \in E \quad \text{for} \quad 0 \leq i < k \\ w(p) &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \end{aligned}$$

Weighted Graphs:

Notation:

$v_0 \overset{p}{\longrightarrow} v_k$ means p is a path from v_0 to v_k. (v_0) is a path from v_0 to v_0 of weight 0.

Definition:

Shortest path weight from u to v as

$$\delta(u,v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\longrightarrow} v\right\} & \text{if } \exists \text{ any such path} \\ \infty & \text{otherwise} \quad (v \text{ unreachable from } u) \end{cases}$$

Single Source Shortest Paths:

Given $G = (V, E), w$ and a source vertex S, find $\delta(S, V)$ [and the best path] from S to each $v \in V$.

Data structures:

$$\begin{aligned} d[v] &= \text{value inside circle} \\ &= \left\{ \begin{array}{ll} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{array} \right\} \Longleftarrow \quad \text{initially} \\ &= \delta(s,v) \Longleftarrow \quad \text{at end} \\ d[v] &\geq \delta(s,v) \quad \text{at all times} \end{aligned}$$

$d[v]$ decreases as we find better paths to v, see Figure 1.
$\Pi[v]$ = predecessor on best path to v, $\Pi[s]$ = NIL

Example:

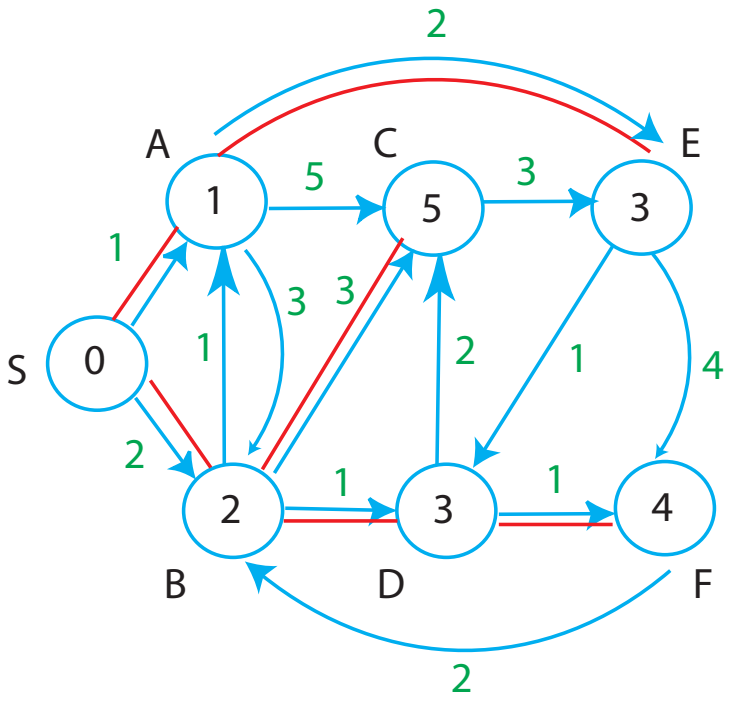


Figure 1: Shortest Path Example: Bold edges give predecessor Π relationships

Negative-Weight Edges:

- Natural in some applications (e.g., logarithms used for weights)
- Some algorithms disallow negative weight edges (e.g., Dijkstra)
- If you have negative weight edges, you might also have negative weight cycles $\Longrightarrow$ may make certain shortest paths undefined!

Example:

See Figure 2

$B \to D \to C \to B$ (origin) has weight $-6 + 2 + 3 = -1 < 0$!
Shortest path $S \longrightarrow C$ (or B, D, E) is undefined. Can go around $B \to D \to C$ as

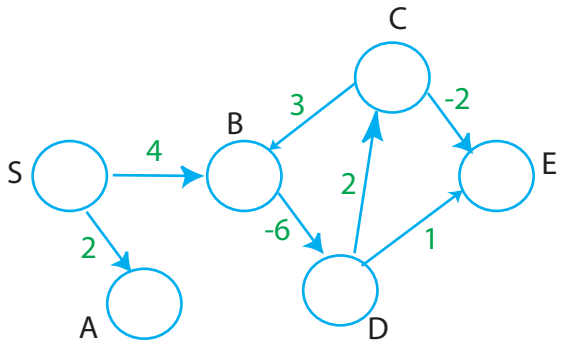


Figure 2: Negative-weight Edges.

many times as you like
Shortest path $S \longrightarrow A$ is defined and has weight 2
If negative weight edges are present, s.p. algorithm should find negative weight cycles (e.g., Bellman Ford)

General structure of S.P. Algorithms (no negative cycles)

Initialize:	for $v \in V$: $d[v] \leftarrow \infty$, $\Pi[v] \leftarrow$ NIL
	$d[S] \leftarrow 0$
Main:	repeat
	select edge (u, v) [somehow]
"Relax" edge (u, v)	if $d[v] > d[u] + w(u, v)$: $d[v] \leftarrow d[u] + w(u, v)$, $\pi[v] \leftarrow u$
	until all edges have $d[v] \leq d[u] + w(u, v)$

Complexity:

Termination? (needs to be shown even without negative cycles)
Could be exponential time with poor choice of edges.

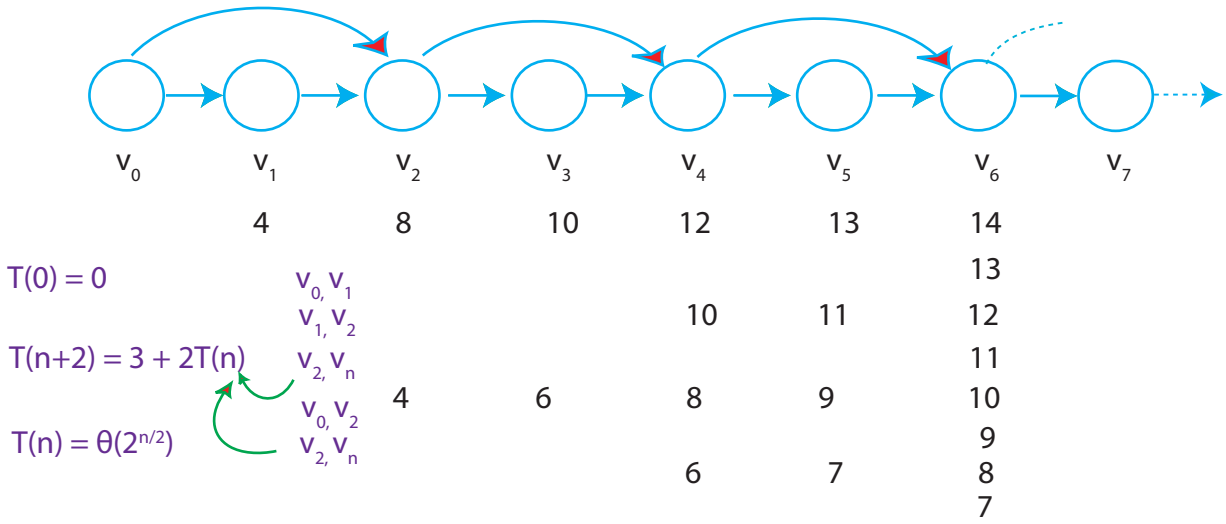


Figure 3: Running Generic Algorithm. The outgoing edges from v_0 and v_1 have weight 4, the outgoing edges from v_2 and v_3 have weight 2, the outgoing edges from v_4 and v_5 have weight 1.

In a generalized example based on Figure 3, we have n nodes, and the weights of edges in the first 3-tuple of nodes are $2^{\frac{n}{2}}$. The weights on the second set are $2^{\frac{n}{2}-1}$, and so on. A pathological selection of edges will result in the initial value of $d(v_{n-1})$ to be $2 \times (2^{\frac{n}{2}} + 2^{\frac{n}{2}-1} + \cdots + 4 + 2 + 1)$. In this ordering, we may then relax the edge of weight 1 that connects v_{n-3} to v_{n-1}. This will reduce $d(v_{n-1})$ by 1. After we relax the edge between v_{n-5} and v_{n-3} of weight 2, $d(v_{n-2})$ reduces by 2. We then might relax the edges (v_{n-3}, v_{n-2}) and (v_{n-2}, v_{n-1}) to reduce $d(v_{n-1})$ by 1. Then, we relax the edge from v_{n-3} to v_{n-1} *again*. In this manner, we might reduce $d(v_{n-1})$ by 1 at each relaxation all the way down to $2^{\frac{n}{2}} + 2^{\frac{n}{2}-1} + \cdots + 4 + 2 + 1$. This will take $O(2^{\frac{n}{2}})$ time.

Optimal Substructure:

Theorem: Subpaths of shortest paths are shortest paths

Let $p = < v_0, v_1, \ldots v_k >$ be a shortest path

Let $p_{ij} = < v_i, v_{i+1}, \ldots v_j > \quad 0 \leq i \leq j \leq k$

Then p_{ij} is a shortest path.

Proof: $p = \overset{p_{0,i}}{v_0 \rightarrow v_i} \overset{p_{ij}}{\underset{p'_{ij}}{\rightarrow \rightarrow}} \overset{p_{jk}}{v_j \rightarrow v_k}$

If p'_{ij} is shorter than p_{ij}, cut out p_{ij} and replace with p'_{ij}; result is shorter than p. Contradiction.

Triangle Inequality:

Theorem: For all $u, v, x \in X$, we have

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v)$$

Proof:

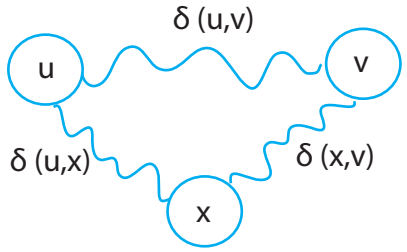


Figure 4: Triangle inequality

Lecture 16: Shortest Paths II - Dijkstra

Lecture Overview

- Review
- Shortest paths in DAGs
- Shortest paths in graphs without negative edges
- Dijkstra's Algorithm

Readings

CLRS, Sections 24.2-24.3

Review

$d[v]$ is the length of the current shortest path from starting vertex s. Through a process of relaxation, $d[v]$ should eventually become $\delta(s, v)$, which is the length of the shortest pathfrom s to v. $\Pi[v]$ is the predecessor of v in the shortest path from s to v.

Basic operation in shortest path computation is the *relaxation operation*

```
RELAX(u, v, w)
    if d[v] > d[u] + w(u, v)
      then d[v] ← d[u] + w(u, v)
           Π[v] ← u
```

Relaxation is Safe

Lemma: The relaxation algorithm maintains the invariant that $d[v] \geq \delta(s, v)$ for all $v \in V$.

Proof: By induction on the number of steps.

Consider $RELAX(u, v, w)$. By induction $d[u] \geq \delta(s, u)$. By the triangle inequality, $\delta(s, v) \leq \delta(s, u) + \delta(u, v)$. This means that $\delta(s, v) \leq d[u] + w(u, v)$, since $d[u] \geq \delta(s, u)$ and $w(u, v) \geq \delta(u, v)$. So setting $d[v] = d[u] + w(u, v)$ is safe. □

DAGs:

Can't have negative cycles because there are no cycles!

1. Topologically sort the DAG. Path from u to v implies that u is before v in the linear ordering.
2. One pass over vertices in topologically sorted order relaxing each edge that leaves each vertex.
   $\Theta(V + E)$ time

Example:

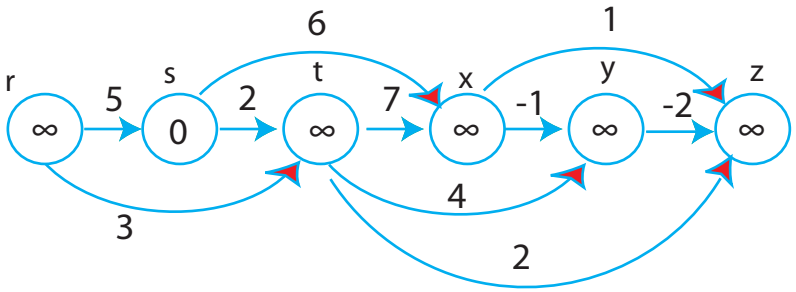


Figure 1: Shortest Path using Topological Sort.

Vertices sorted left to right in topological order

Process r: stays ∞. All vertices to the left of s will be ∞ by definition

Process s: $t : \infty \to 2$ $x : \infty \to 6$ (see top of Figure 2)

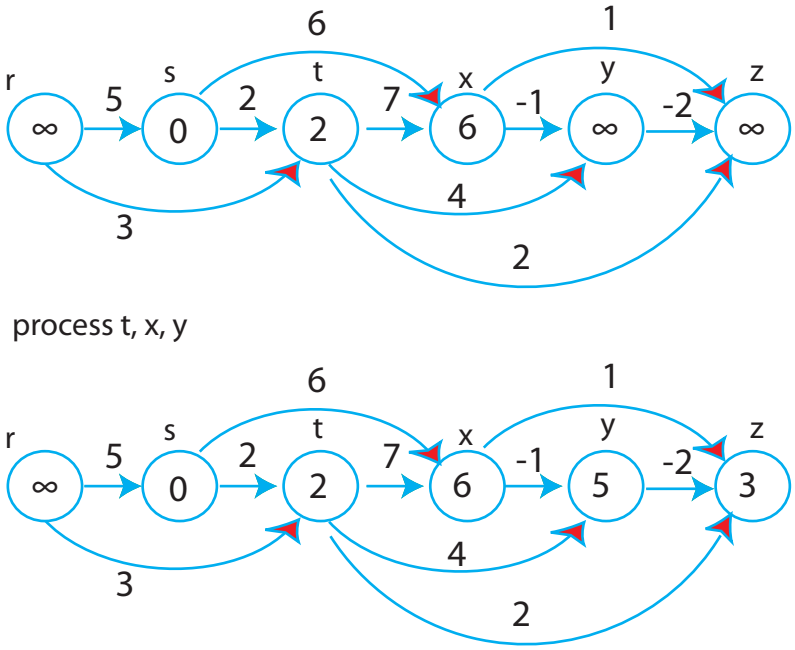


Figure 2: Preview of Dynamic Programming

DIJKSTRA Demo

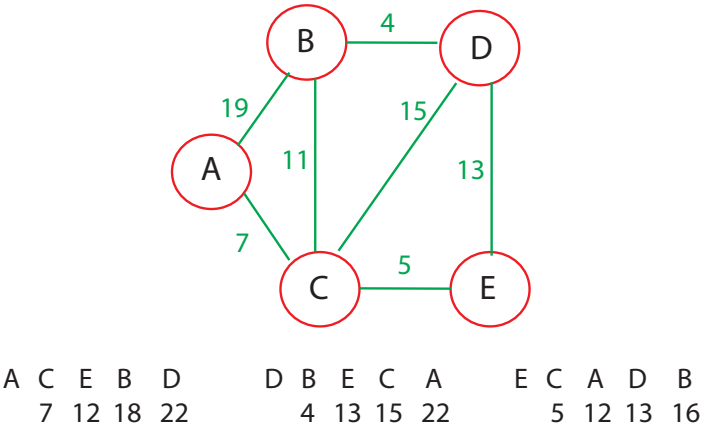


Figure 3: Dijkstra Demonstration with Balls and String.

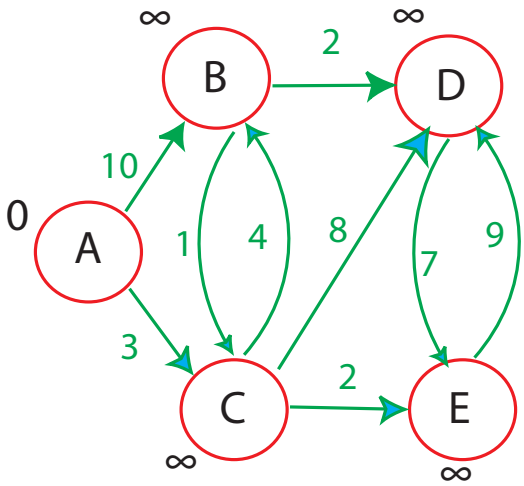
Dijkstra's Algorithm

For each edge (u, v) ϵ E, assume $w(u, v) \geq 0$, maintain a set S of vertices whose final shortest path weights have been determined. Repeatedly select $u \epsilon V - S$ with minimum shortest path estimate, add u to S, relax all edges out of u.

Pseudo-code

```
Dijkstra (G, W, s)        //uses priority queue Q
        Initialize (G, s)
        S ← φ
        Q ← V[G]          //Insert into Q
        while Q ≠ φ
              do u ← EXTRACT-MIN(Q)        //deletes u from Q
              S = S ∪ {u}
              for each vertex v ϵ Adj[u]
                    do RELAX (u, v, w)     ← this is an implicit DECREASE_KEY operation
```

Example



S = { }	{ A B C D E } =	Q
S = { A }	0 ∞ ∞ ∞ ∞	
S = { A, C }	0 10 3 ∞ ∞	← after relaxing edges from A
S = { A, C }	0 7 3 11 5	← after relaxing edges from C
S = { A, C, E }	0 7 3 11 5	
S = { A, C , E, B}	0 7 3 9 5	← after relaxing edges from B

Figure 4: Dijkstra Execution

Strategy: Dijkstra is a greedy algorithm: choose closest vertex in $V - S$ to add to set S.

Correctness: We know relaxation is safe. The key observation is that each time a vertex u is added to set S, we have $d[u] = \delta(s, u)$.

Dijkstra Complexity

$\Theta(v)$ inserts into priority queue
$\Theta(v)$ EXTRACT_MIN operations
$\Theta(E)$ DECREASE_KEY operations

Array impl:

$\Theta(v)$ time for extra min
$\Theta(1)$ for decrease key
Total: $\Theta(V.V + E.1) = \Theta(V^2 + E) = \Theta(V^2)$

Binary min-heap:

$\Theta(\lg V)$ for extract min
$\Theta(\lg V)$ for decrease key
Total: $\Theta(V \lg V + E \lg V)$

Fibonacci heap (not covered in 6.006):

$\Theta(\lg V)$ for extract min
$\Theta(1)$ for decrease key
amortized cost
Total: $\Theta(V \lg V + E)$

Lecture 17: Shortest Paths III: Bellman-Ford

Lecture Overview

- Review: Notation
- Generic S.P. Algorithm
- Bellman-Ford Algorithm
  - Analysis
  - Correctness

Recall:

$$\begin{aligned} \text{path } p &= < v_0, v_1, \ \ldots, \ v_k > \\ & \quad (v_1, v_{i+1}) \ \in E \quad 0 \leq i < k \\ w(p) &= \sum_{i-0}^{k-1} w(v_i, v_{i+1}) \end{aligned}$$

Shortest path weight from u to v is $\delta(u, v)$. $\delta(u, v)$ is ∞ if v is unreachable from u, undefined if there is a negative cycle on some path from u to v.

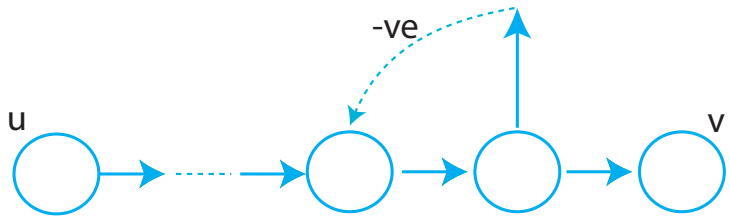


Figure 1: Negative Cycle.

Generic S.P. Algorithm

Initialize: for $v \in V$: $d[v] \leftarrow \infty$, $\Pi[v] \leftarrow$ NIL

$d[S] \leftarrow 0$

Main: repeat

select edge (u, v) [somehow]

"Relax" edge (u, v):

if $d[v] > d[u] + w(u, v)$:

$d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$

until you can't relax any more edges or you're tired or ...

Complexity:

Termination: Algorithm will continually relax edges when there are negative cycles present.

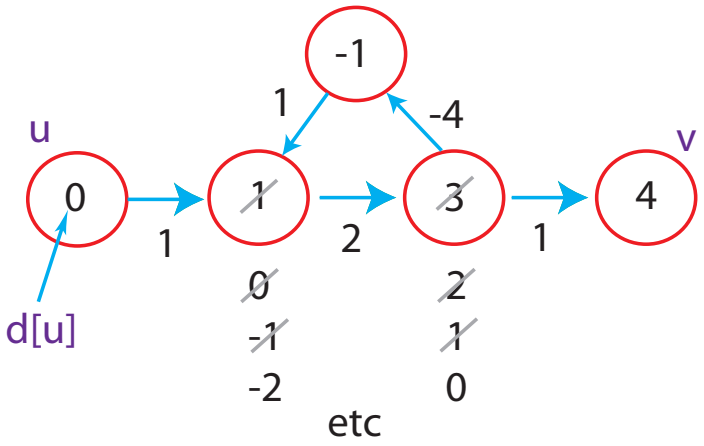


Figure 2: Algorithm may not terminate due to negative cycles.

Complexity could be exponential time with poor choice of edges.

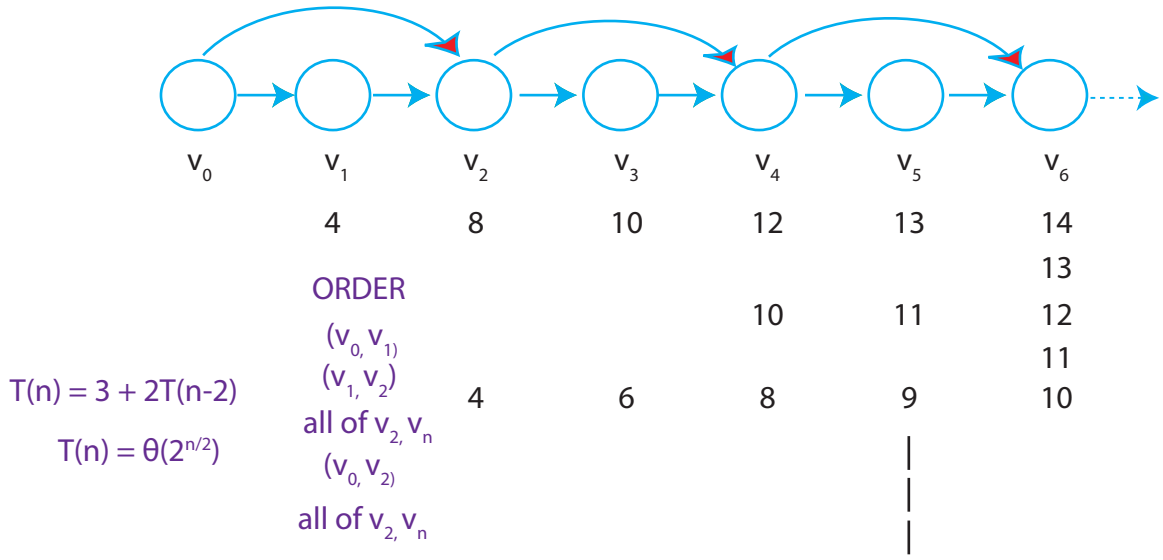


Figure 3: Algorithm could take exponential time. The outgoing edges from v_0 and v_1 have weight 4, the outgoing edges from v_2 and v_3 have weight 2, the outgoing edges from v_4 and v_5 have weight 1.

5-Minute 6.006

Figure 4 is what I want you to remember from 6.006 five years after you graduate!

Bellman-Ford(G,W,s)

```
Initialize ()
for i = 1 to |V| − 1
    for each edge (u, v) ∈ E:
        Relax(u, v)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
        then report a negative-weight cycle exists
```

At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.

Theorem:
If $G = (V, E)$ contains no negative weight cycles, then after Bellman-Ford executes $d[v] = \delta(s, v)$ for all $v \in V$.

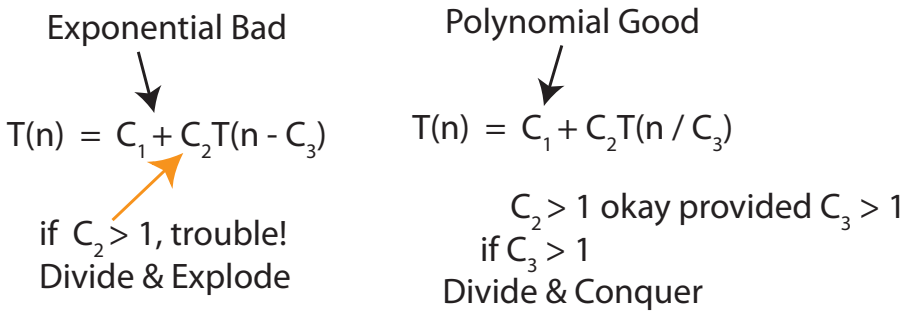


Figure 4: Exponential vs. Polynomial.

Proof:
Let $v \in V$ be any vertex. Consider path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from $v_0 = s$ to $v_k = v$ that is a shortest path with minimum number of edges. No negative weight cycles $\implies$ p is simple $\implies$ $k \leq |V| - 1$.

Consider Figure 6. Initially $d[v_0] = 0 = \delta(s, v_0)$ and is unchanged since no negative cycles.
After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$, because we will relax the edge (v_0, v_1) in the pass, and we can't find a shorter path than this shortest path. (Note that we are invoking optimal substructure and the safeness lemma from Lecture 16 here.)
After 2 passes through E, we have $d[v_2] = \delta(s, v_2)$, because in the second pass we will relax the edge (v_1, v_2).
After i passes through E, we have $d[v_i] = \delta(s, v_i)$.
After $k \leq |V| - 1$ passes through E, we have $d[v_k] = d[v] = \delta(s, v)$. □

Corollary

If a value $d[v]$ fails to converge after $|V| - 1$ passes, there exists a negative-weight cycle reachable from s.
Proof:
After $|V| - 1$ passes, if we find an edge that can be relaxed, it means that the current shortest path from s to some vertex is not simple and vertices are repeated. Since this cyclic path has less weight than any simple path the cycle has to be a negative-weight cycle. □

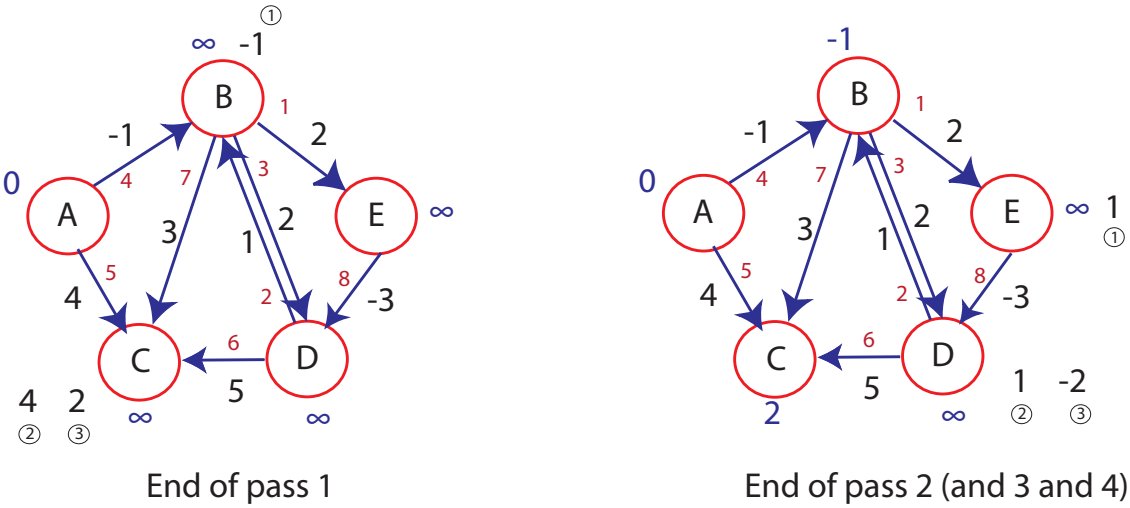


Figure 5: The numbers in circles indicate the order in which the δ values are computed.

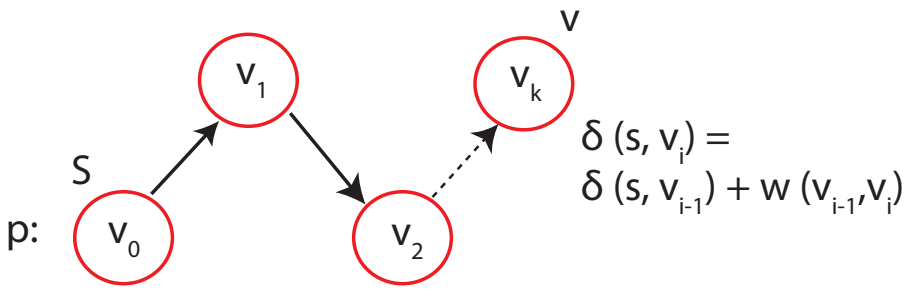


Figure 6: Illustration for proof.

Longest Simple Path and Shortest Simple Path

Finding the longest simple path in a graph with non-negative edge weights is an NP-hard problem, for which no known polynomial-time algorithm exists. Suppose one simply negates each of the edge weights and runs Bellman-Ford to compute shortest paths. Bellman-Ford will not necessarily compute the longest paths in the original graph, since there might be a negative-weight cycle reachable from the source, and the algorithm will abort.

Similarly, if we have a graph with negative cycles, and we wish to find the longest *simple* path from the source s to a vertex v, we cannot use Bellman-Ford. The shortest simple path problem is also NP-hard.

Lecture 18: Shortest Paths IV - Speeding up Dijkstra

Lecture Overview

- Single-source single-target Dijkstra
- Bidirectional search
- Goal directed search - potentials and landmarks

Readings

Wagner, Dorothea, and Thomas Willhalm. "Speed-up Techniques for Shortest-Path Computations." Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS 2007): 23-36. Read up to Section 3.2.

DIJKSTRA single-source, single-target

```
Initialize()
Q ← V[G]
while Q ≠ φ
      do u ← EXTRACT_MIN(Q) (stop if u = t!)
      for each vertex v ϵ Adj[u]
          do RELAX(u, v, w)
```

Observation: If only shortest path from s to t is required, stop when t is removed from Q, i.e., when $u = t$

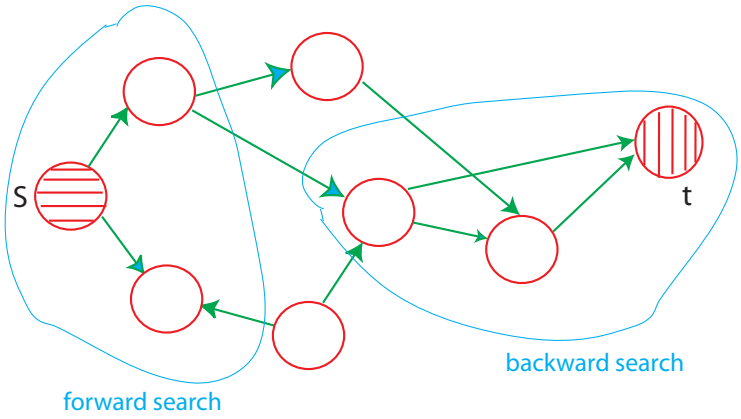


Figure 1: Bi-directional Search Idea.

Bi-Directional Search

Note: Speedup techniques covered here do not change worst-case behavior, but reduce the number of visited vertices in practice.

Bi-D Search

Alternate forward search from s
backward search from t
(follow edges backward)
$d_f(u)$ distances for forward search
$d_b(u)$ distances for backward search

Algorithm terminates when some vertex w has been processed, i.e., deleted from the queue of both searches, Q_f and Q_b
Subtlety: After search terminates, find node x with minimum value of $d_f(x) + d_b(x)$. x may not be the vertex w that caused termination as in example to the left!
Find shortest path from s to x using Π_f and shortest path backwards from t to x using Π_b. *Note:* x will have been deleted from either Q_f or Q_b or both.

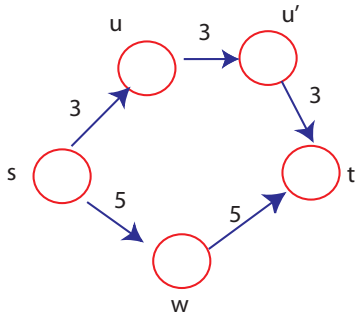


Figure 2: Bi-D Search Example.

Minimum value for $d_f(x) + d_b(x)$ over all vertices that have been processed in at least one search (see Figure 3):

$$d_f(u) + d_b(u) = 3 + 6 = 9$$

$$d_f(u') + d_b(u') = 6 + 3 = 9$$

$$d_f(w) + d_b(w) = 5 + 5 = 10$$

Goal-Directed Search or A^*

Modify edge weights with potential function over vertices.

$$\overline{w}\ (u,v) = w\ (u,v) - \lambda(u) + \lambda(v)$$

Search toward target as shown in Figure 4:

Correctness

$$\overline{w}(p) = w(p) - \lambda_t(s) + \lambda_t(t)$$

So shortest paths are maintained in modified graph with $\overline{w}$ weights (see Figure 5).

To apply Dijkstra, we need $\overline{w}(u,v) \geq 0$ for all (u,v).

Choose potential function appropriately, to be feasible.

Landmarks

Small set of landmarks LCV. For all $u \in V, l \in L$, pre-compute $\delta(u,l)$.

Potential $\lambda_t^{(l)}(u) = \delta(u,l) - \delta(t,l)$ for each l.

CLAIM: $\lambda_t^{(l)}$ is feasible.

Feasibility

$$\begin{aligned}
\overline{w}(u,v) &= w(u,v) - \lambda_t^{(l)}(u) + \lambda_t^{(l)}(v) \\
&= w(u,v) - \delta(u,l) + \delta(t,l) + \delta(v,l) - \delta(t,l) \\
&= w(u,v) - \delta(u,l) + \delta(v,l) \geq 0 \quad \text{by the } \Delta\text{ -inequality} \\
\lambda_t(u) &= \max_{l\ \epsilon\ L} \lambda_t^{(l)}(u) \ \text{ is also feasible}
\end{aligned}$$

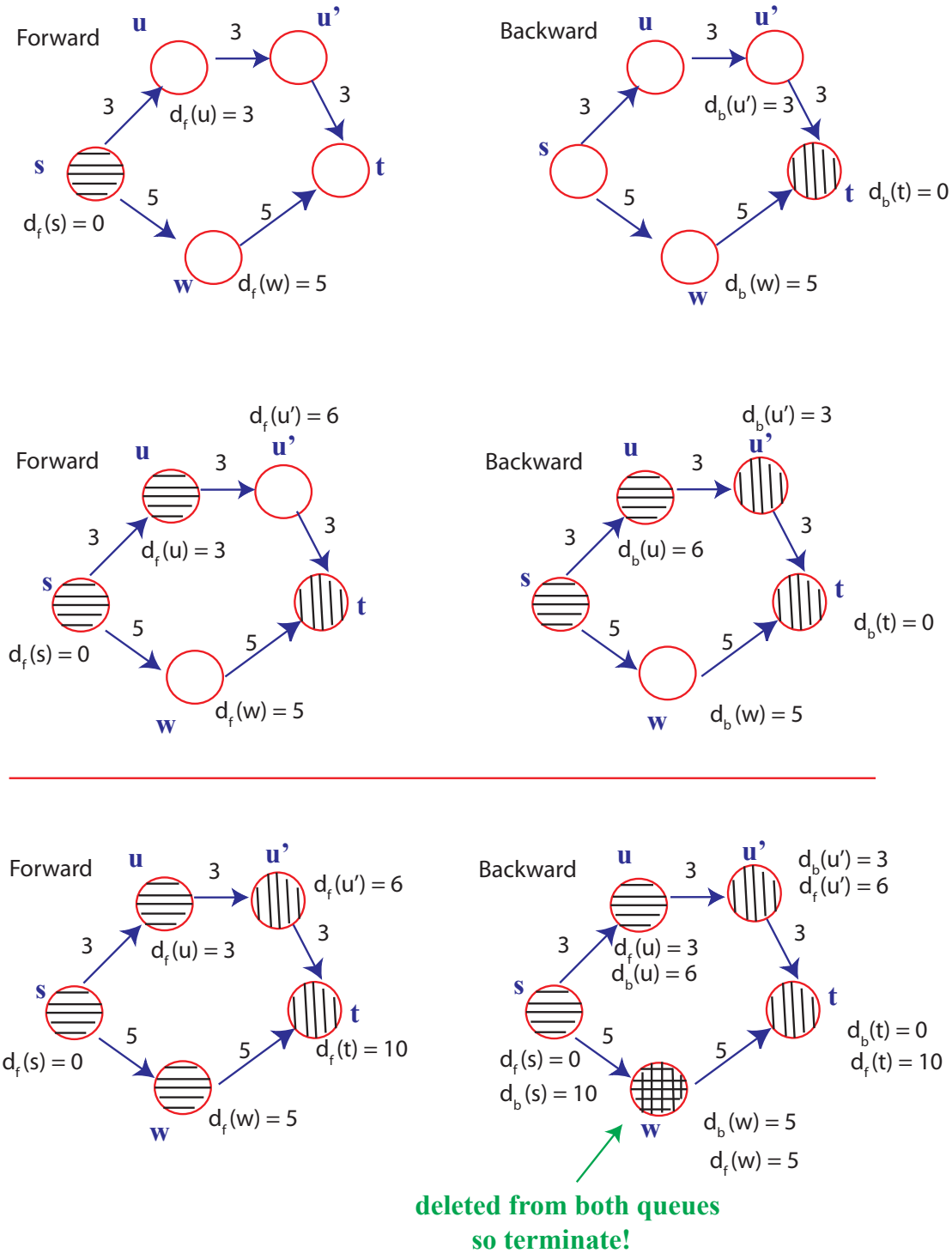


Figure 3: Forward and Backward Search and Termination.

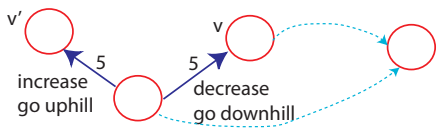


Figure 4: Targeted Search

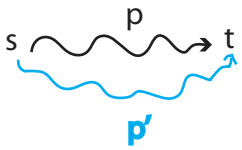


Figure 5: Modifying Edge Weights.

Lecture 19: Dynamic Programming I: Memoization, Fibonacci, Shortest Paths, Guessing

Lecture Overview

- Memoization and subproblems
- Examples
  - Fibonacci
  - Shortest Paths
- Guessing & DAG View

Dynamic Programming (DP)

Big idea, hard, yet simple

- Powerful algorithmic design technique
- Large class of seemingly exponential problems have a polynomial solution ("only") via DP
- Particularly for optimization problems (min / max) (e.g., shortest paths)

* DP ≈ "controlled brute force"
* DP ≈ recursion + re-use

History

Richard E. Bellman (1920-1984)
Richard Bellman received the IEEE Medal of Honor, 1979. "Bellman ... explained that he invented the name 'dynamid programming' to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who 'had a pathological fear and hatred of the term, research'. He settled on the term 'dynamic programming' because it would be difficult to give a 'pejorative meaning' and because 'it was something not even a Congressman could object to' " [John Rust 2006]

Fibonacci Numbers

$$F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2}$$

Goal: compute F_n

Naive Algorithm

follow recursive definition

fib(n):
    if $n \leq 2$: return $f = 1$
    else: return $f = \text{fib}(n-1) + \text{fib}(n-2)$
$\implies T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n$
    $\geq 2T(n-2) + O(1) \geq 2^{n/2}$
    EXPONENTIAL — BAD!

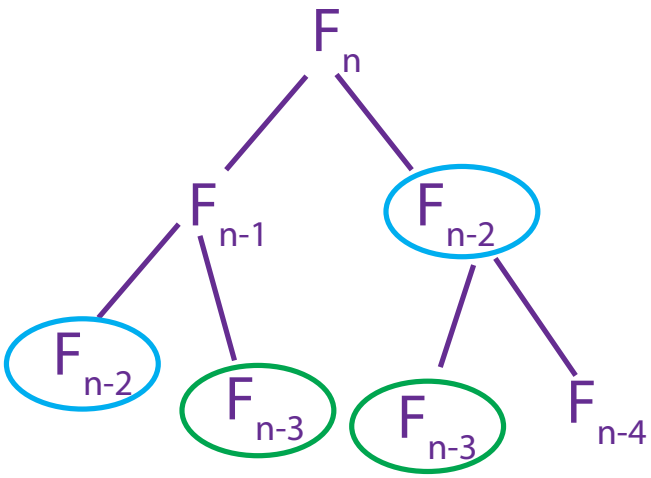


Figure 1: Naive Fibonacci Algorithm.

Memoized DP Algorithm

Remember, remember

memo = { }
fib(n):
    if n in memo: return memo[n]
    else: if $n \leq 2 : f = 1$
        else: $f = \text{fib}(n-1) + \text{fib}(n-2)$
        memo[n] = f
        return f

- $\implies$ fib(k) only recurses first time called, $\forall k$
- $\implies$ only n nonmemoized calls: $k = n, n-1, \ldots, 1$
- memoized calls free ($\Theta(1)$ time)
- $\implies$ $\Theta(1)$ time per call (ignoring recursion)

POLYNOMIAL — GOOD!

* DP $\approx$ recursion + memoization

- memoize (remember) & re-use solutions to subproblems that help solve problem
  - in Fibonacci, subproblems are $F_1, F_2, \ldots, F_n$

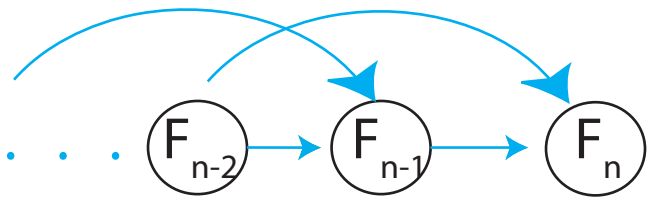
* $\implies$ time = # of subproblems · time/subproblem

- Fibonacci: # of subproblems is n, and time/subproblem is $\Theta(1) = \Theta(n)$ (ignore recursion!).

Bottom-up DP Algorithm

```
fib = {}                                              } Θ(n)
for k in [1, 2, ..., n]:                              }
    if k ≤ 2: f = 1                                   }
    else: f = fib[k − 1] + fib[k − 2]   } Θ(1)         }
    fib[k] = f                                        }
return fib[n]                                         }
```

- exactly the same computation as memoized DP (recursion "unrolled")
- in general: topological sort of subproblem dependency DAG



- practically faster: no recursion
- analysis more obvious
- can save space: just remember last 2 fibs $\implies$ $\Theta(1)$

[Sidenote: There is also an $O(\lg n)$-time algorithm for Fibonacci, via different techniques]

Shortest Paths

- Recursive formulation:
  $\delta(s,v) = \min\{w(u,v) + \delta(s,u) \,\big|\, (u,v) \in E\}$
- Memoized DP algorithm: takes infinite time if cycles!
  in some sense necessary to handle negative cycles

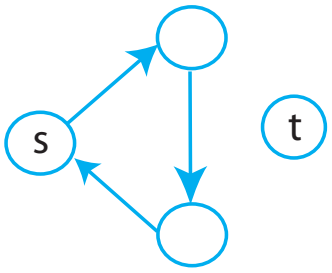


Figure 2: Shortest Paths

- works for directed acyclic graphs in $O(V+E)$
  effectively DFS/topological sort + Bellman-Ford round rolled into a single recursion

* Subproblem dependency should be acyclic

- more subproblems remove cyclic dependence:
  $\delta_k(s,v)$ = shortest $s \to v$ path using $\leq k$ edges
- recurrence:

$$\begin{aligned}
\delta_k(s,v) &= \min\{\delta_{k-1}(s,u) + w(u,v) \,\big|\, (u,v) \in E\} \\
\delta_0(s,v) &= \infty \text{ for } s \neq v \text{ (base case)} \\
\delta_k(s,s) &= 0 \text{ for any } k \text{ (base case, if no negative cycles)}
\end{aligned}$$

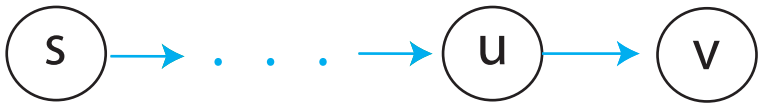
- Goal: $\delta(s,v) = \delta_{|V|-1}(s,v)$ (if no negative cycles)
- memoize
- time: $\underbrace{\#\text{ subproblems}}_{|V|\cdot|V|} \cdot \underbrace{\text{time/subproblem}}_{O(v)} = 0(V^3)$
- actually $\Theta(\text{indegree}(v))$ for $\delta_k(s,v)$
- $\implies$ time $= \Theta(V \sum_{v \in V} \text{indegree}(V)) = \Theta(VE)$

BELLMAN-FORD!

Guessing

How to design recurrence

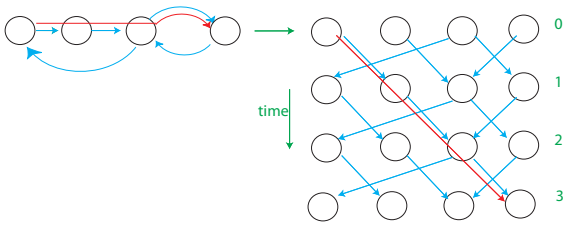
- want shortest $s \to v$ path



- what is the last edge in path? dunno
- guess it is (u, v)
- path is shortest $s \to u$ path + edge (u, v)
  by optimal substructure
- cost is $\delta_{k-1}(s, u) + w(u, v)$
  another subproblem
- to find best guess, try all ($|V|$ choices) and use best
- * key: small (polynomial) # possible guesses per subproblem — typically this dominates time/subproblem

* DP ≈ recursion + memoization + guessing

DAG view



- like replicating graph to represent time
- converting shortest paths in graph → shortest paths in DAG

* DP ≈ shortest paths in some DAG

Lecture 20: Dynamic Programming II

Lecture Overview

- 5 easy steps
- Text justification
- Perfect-information Blackjack
- Parent pointers

Summary

* DP ≈ "careful brute force"
* DP ≈ guessing + recursion + memoization
* DP ≈ dividing into reasonable # subproblems whose solutions relate — acyclicly — usually via guessing parts of solution.
* time = # subproblems × $\underbrace{\text{time/subproblem}}$
  treating recursive calls as $O(1)$
  (usually mainly guessing)
  - essentially an amortization
  - count each subproblem only once; after first time, costs $O(1)$ via memoization
* DP ≈ shortest paths in some DAG

5 Easy Steps to Dynamic Programming

1. define subproblems — count # subproblems
2. guess (part of solution) — count # choices
3. relate subproblem solutions — compute time/subproblem
4. recurse + memoize — time = time/subproblem · # subproblems
   OR build DP table bottom-up
   check subproblems acyclic/topological order
5. solve original problem: = a subproblem
   OR by combining subproblem solutions — $\Longrightarrow$ extra time

Examples:	Fibonacci	Shortest Paths
subprobs:	F_k for $1 \le k \le n$	$\delta_k(s,v)$ for $v \in V$, $0 \le k < \lvert V\rvert$ $= \min s \to v$ path using $\le k$ edges
# subprobs:	n	V^2
guess:	nothing	edge into v (if any)
# choices:	1	indegree$(v)+1$
recurrence:	$F_k = F_{k-1} + F_{k-2}$	$\delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\}$
time/subpr:	$\Theta(1)$	$\Theta(1 +$ indegree$(v))$
topo. order:	for $k = 1, \ldots, n$	for $k = 0, 1, \ldots \lvert V\rvert - 1$ for $v \in V$
total time:	$\Theta(n)$	$\Theta(VE)$ $+ \Theta(V^2)$ unless efficient about indeg. 0
orig. prob.:	F_n	$\delta_{\lvert V\rvert-1}(s,v)$ for $v \in V$
extra time:	$\Theta(1)$	$\Theta(V)$

Text Justification

Split text into "good" lines

- obvious (MS Word/Open Office) algorithm: put as many words that fit on first line, repeat
- but this can make very bad lines

Figure 1: Good vs. Bad Text Justification.

- Define badness(i,j) for line of words$[i:j]$.
  For example, ∞ if total length $>$ page width, else (page width $-$ total length)^3.
- goal: split words into lines to min $\sum$ badness

1. subproblem = min. badness for suffix words$[i:]$
   $\implies$ # subproblems $= \Theta(n)$ where $n =$ # words
2. guessing = where to end first line, say $i:j$
   $\implies$ # choices $= n - i = O(n)$

3. recurrence:
   - DP[i] = min(badness $(i, j) + DP[j]$ for j in range $(i + 1, n + 1)$)
   - $DP[n] = 0$
     $\implies$ time per subproblem $= \Theta(n)$
4. order: for $i = n, n - 1, \ldots, 1, 0$
   total time $= \Theta(n^2)$

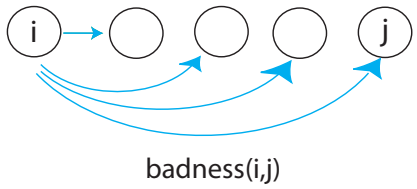


Figure 2: DAG.

5. solution $= DP[0]$

Perfect-Information Blackjack

- Given entire deck order: $c_0, c_1, \cdots c_{n-1}$
- 1-player game against stand-on-17 dealer
- when should you hit or stand? GUESS
- goal: maximize winnings for fixed bet \$1
- may benefit from losing one hand to improve future hands!

1. subproblems: BJ(i) = best play of $\underbrace{c_i, \ldots c_{n-1}}_{\text{remaining cards}}$ where i is # cards "already played"
   $\implies$ # subproblems $= n$
2. guess: how many times player "hits" (hit means draw another card)
   $\implies$ # choices $\leq n$
3. recurrence: BJ(i) = max(
   outcome $\in \{+1, 0, -1\}$ + BJ$(i$ + # cards used$)$ $O(n)$
   for # hits in $0, 1, \ldots$ if valid play $\sim$ don't hit after bust $O(n)$

)
$\implies$ time/subproblem $= \Theta(n^2)$

4. order: for i in reversed(range(n))
   total time $= \Theta(n^3)$
   time is really $\sum_{i=0}^{n-1} \sum_{\#h=0}^{n-i-O(1)} \Theta(n-i-\#h) = \Theta(n^3)$ still

5. solution: BJ(0)
   detailed recurrence: before memoization (ignoring splits/betting)

$\Theta(n^2)$ {
  $\Theta(n)$ {
  $\Theta(n)$ with care {

```
BJ(i):
if n − i < 4: return 0 (not enough cards)
for p in range(2, n − i − 1): (# cards taken)
        player = sum(c_i, c_{i+2}, c_{i+4:i+p+2})
        if player > 21: (bust)
                options.append(−1(bust) + BJ(i + p + 2))
                break
        for d in range(2, n − i − p)
                dealer = sum(c_{i+1}, c_{i+3}, c_{i+p+2:i+p+d})
                if dealer ≥ 17: break
        if dealer > 21: dealer = 0 (bust)
        options.append(cmp(player, dealer) + BJ(i + p + d))
return      max(options)
```

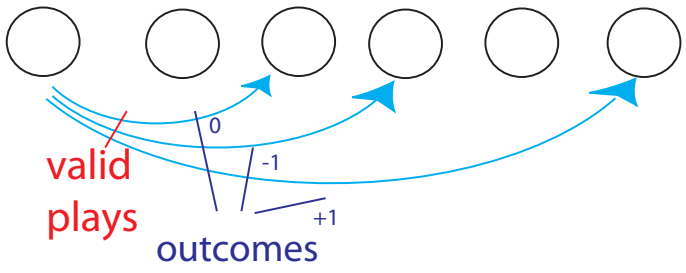


Figure 3: DAG View

Parent Pointers

To recover actual solution in addition to cost, store parent pointers (which guess used at each subproblem) & walk back

- typically: remember argmin/argmax in addition to min/max
- example: text justification

```
(3)' DP[i] = min(badness(i,j) + DP[i][0],j)
                 for j in range(i+1,n+1)
     DP[n] = (0, None)
(5)' i = 0
     while i is not None:
        start line before word i
        i = DP[i][1]
```

- just like memoization & bottom-up, this transformation is automatic
  no thinking required

Lecture 21: Dynamic Programming III

Lecture Overview

- Subproblems for strings
- Parenthesization
- Edit distance (& longest common subseq.)
- Knapsack
- Pseudopolynomial Time

Review:

* 5 easy steps to dynamic programming

  (a) define subproblems — count # subproblems

  (b) guess (part of solution) — count # choices

  (c) relate subproblem solutions — compute time/subproblem

  (d) recurse + memoize — time = time/subproblem · # subproblems
  OR build DP table bottom-up
  check subproblems acyclic/topological order

  (e) solve original problem: = a subproblem
  OR by combining subproblem solutions $\implies$ extra time

* problems from L20 (text justification, Blackjack) are on sequences (words, cards)

* useful problems for strings/sequences x:
  suffixes $x[i:]$ } $\Theta(|x|)$ ← cheaper $\implies$ use if possible
  prefixes $x[:i]$
  substrings $x[i:j]$ } $\Theta(x^2)$

Parenthesization:

Optimal evaluation of associative expression $A[0] \cdot A[1] \cdots A[n-1]$ — e.g., multiplying rectangular matrices

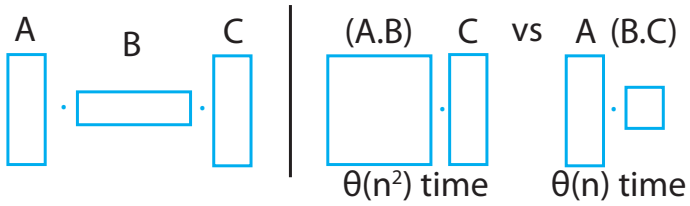


Figure 1:

2. guessing = outermost multiplication $(\underbrace{\cdots}_{\uparrow k-1})(\underbrace{\cdots}_{\uparrow k})$

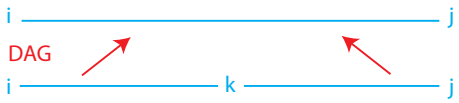
$\implies$ # choices $= O(n)$

1. subproblems = ~~prefixes & suffixes?~~ NO
   = cost of substring $A[i:j]$

$\implies$ # subproblems $= \Theta(n^2)$

3. recurrence:

   - DP$[i,j]$ = min(DP$[i,k]$ + DP$[k,j]$+ cost of multiplying $(A[i] \cdots A[k-1])$ by $(A[k] \cdots A[j-1])$ for k in range$(i+1, j)$)



   - DP$[i, i+1] = 0$
     $\implies$ cost per subproblem $= O(j-i) = O(n)$

4. topological order: increasing substring size. Total time $= O(n^3)$

5. original problem $= DP[0, n]$

(& use parent pointers to recover parens.)

NOTE: Above DP is not shortest paths in the subproblem DAG! Two dependencies $\implies$ not path!

Edit Distance

Used for DNA comparison, diff, CVS/SVN/..., spellchecking (typos), plagiarism detection, etc.
Given two strings x & y, what is the cheapest possible sequence of character edits (insert c, delete c, replace c → c') to transform x into y?

- cost of edit depends only on characters c, c'
- for example in DNA, C → G common mutation $\implies$ low cost
- cost of sequence = sum of costs of edits
- If insert & delete cost 1, replace costs 0, minimum edit distance equivalent to finding longest common subsequence. Note that a subsequence is sequential but not necessarily contiguous.
- for example H I E R O G L Y P H O L O G Y vs. M I C H A E L A N G E L O $\implies$ HELLO

Subproblems for multiple strings/sequences

- combine suffix/prefix/substring subproblems
- multiply state spaces
- still polynomial for $O(1)$ strings

Edit Distance DP

(1) subproblems: $c(i, j) =$ edit-distance$(x[i:], y[j:])$ for $0 \le i < |x|, 0 \le j < |y|$
$\implies \Theta(|x| \cdot |y|)$ subproblems

(2) guess whether, to turn x into y, (3 choices):

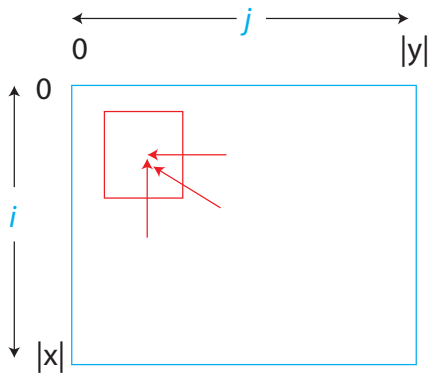
- $x[i]$ deleted
- $y[j]$ inserted
- $x[i]$ replaced by $y[j]$

(3) recurrence: $c(i, j) =$ maximum of:

- cost(delete $x[i]$) $+ c(i + 1, j)$ if $i < |x|$,
- cost(insert $y[j]$) $+ c(i, j + 1)$ if $j < |y|$,
- cost(replace $x[i] \to y[j]$) $+ c(i + 1, j + 1)$ if $i < |x| \& j < |y|$

base case: $c(|x|, |y|) = 0$
$\implies \Theta(1)$ time per subproblem

(4) topological order: DAG in 2D table:



- bottom-up OR right to left
- only need to keep last 2 rows/columns
  $\implies$ linear space
- total time $= \Theta(|x| \cdot |y|)$

(5) original problem: $c(0, 0)$

Knapsack:

Knapsack of size S you want to pack

- item i has integer size s_i & real value v_i
- goal: choose subset of items of maximum total value subject to total size $\leq S$

First Attempt:

1. ~~subproblem = value for suffix i:~~ WRONG
2. guessing = whether to include item i $\implies$ # choices = 2
3. recurrence:
   - $DP[i] = \max(DP[i+1], v_i + DP[i+1]$ if ~~$s_i \leq S$~~?!)
   - not enough information to know whether item i fits — how much space is left? GUESS!

Correct:

1. subproblem = value for suffix i:
   given knapsack of size X

$\implies$ # subproblems $= O(nS)$!

3. recurrence:
   - $DP[i, X] = \max(DP[i+1, X], v_i + DP[i+1, X - s_i] \text{ if } s_i \leq X)$
   - $DP[n, X] = 0$
     $\implies$ time per subproblem $= O(1)$
4. topological order: for i in $n, \ldots, 0$: for X in $0, \ldots S$

   total time $= O(nS)$
5. original problem $= DP[0, S]$
   (& use parent pointers to recover subset)

AMAZING: effectively trying all possible subsets! ... but is this actually fast?

Polynomial time

Polynomial time = polynomial in input size

- here $\Theta(n)$ if number S fits in a word
- $O(n \lg S)$ in general
- S is exponential in $\lg S$ (not polynomial)

Pseudopolynomial Time

Pseudopolynomial time = polynomial in the problem size AND the numbers (here: S, s_i's, v_i's) in input. $\Theta(nS)$ is pseudopolynomial.

Remember:
polynomial — GOOD
exponential — BAD
pseudopoly — SO SO

Lecture 22: Dynamic Programming IV

Lecture Overview

- 2 kinds of guessing
- Piano/Guitar Fingering
- Tetris Training
- Super Mario Bros.

Review:

* 5 easy steps to dynamic programming

(a) define subproblems — count # subproblems

(b) guess (part of solution) — count # choices

(c) relate subproblem solutions — compute time/subproblem

(d) recurse + memoize — time = time/subproblem · # subproblems
OR build DP table bottom-up
check subproblems acyclic/topological order

(e) solve original problem: = a subproblem
OR by combining subproblem solutions — $\Longrightarrow$ extra time

* 2 kinds of guessing:

(A) In (3), guess which other subproblems to use (used by every DP except Fibonacci)

(B) In (1), create more subproblems to guess/remember more structure of solution used by knapsack DP

- effectively report many solutions to subproblem.
- lets parent subproblem know features of solution.

Piano/Guitar Fingering:

Piano

[Parncutt, Sloboda, Clarke, Raekallio, Desain, 1997]
[Hart, Bosch, Tsai 2000]
[Al Kasimi, Nichols, Raphael 2007] etc.

- given musical piece to play, say sequence of n (single) notes with right hand

- fingers $1, 2, \ldots, F = 5$ for humans
- metric $d(f, p, g, q)$ of difficulty going from note p with finger f to note q with finger g

  e.g., $1 < f < g$ & $p > q \implies$ uncomfortable
  stretch rule: $p \ll q \implies$ uncomfortable
  legato (smooth) $\implies \infty$ if $f = g$
  weak-finger rule: prefer to avoid $g \in \{4, 5\}$
  $3 \to 4$ & $4 \to 3$ annoying $\sim$ etc.

First Attempt:

1. ~~subproblem = min. difficulty for suffix notes$[i:]$~~
2. ~~guessing = finger f for first note$[i]$~~
3. recurrence:
   ~~DP$[i] = \min(DP[i+1] + d($note$[i], f,$ note$[i+1], ?)$ for $f \cdots)$~~
   $\to$ not enough information!

Correct DP:

1. subproblem = min difficulty for suffix notes$[i:]$ given finger f on first note$[i]$
   $\implies n \cdot F$ subproblems
2. guessing = finger g for next note$[i+1]$
   $\implies F$ choices
3. recurrence:
   $DP[i, f] = \min(DP[i+1, g] + d(\text{note}[i], f, \text{note}[i+1], g)$ for g in range$(F))$
   DP$[n, f] = 0$
   $\implies \Theta(F)$ time/subproblem
4. topo. order: for i in reversed(range(n)):
   for f in $1, 2, \ldots, F$:
   total time $O(nF^2)$
5. orig. prob. = min(DP$[0, f]$ for f in $1, \ldots, F)$
   (guessing very first finger)

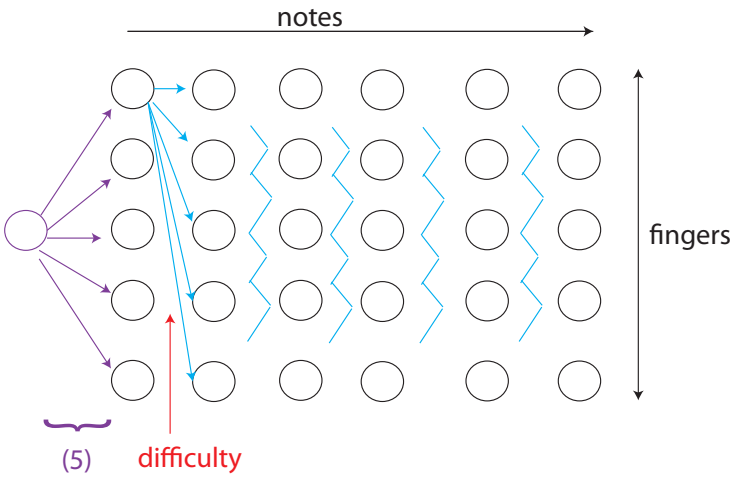


Figure 1: DAG.

Guitar

Up to S ways to play same note! (where S is # strings)

- redefine "finger" = finger playing note + string playing note
- $\implies F \to F \cdot S$

Generalization:

Multiple notes at once e.g. chords

- input: notes[i] = list of $\leq F$ notes
  (can't play > 1 note with a finger)
- state we need to know about "past" now assignment of F fingers to $\leq F+1$ notes/null
  $\implies (F+1)^F$ such mappings

(1) $n \cdot (F+1)^F$ subproblems where $(F+1)^F$ is how notes$[i]$ is played

(2) $(F+1)^F$ choices (how notes$[i+1]$ played)

(3) $n \cdot (F+1)^{2F}$ total time

- works for 2 hands $F = 10$
- just need to define appropriate d

Figure 2: Tetris.

Tetris Training:

- given sequence of n Tetris pieces & an empty board of small width w
- must choose orientation & x coordinate for each
- then must drop piece till it hits something
- full rows do not clear
  without the above two artificialities WE DON'T KNOW!
  (but: if nonempty board & w large then NP-complete)
- goal: survive i.e., stay within height h

First Attempt:

1. ~~subproblem = survive in suffix i:?~~ WRONG
2. guessing = how to drop piece i $\implies$ # choices $= O(w)$
3. ~~recurrence: $DP[i] = DP[i+1]$~~ ?! not enough information!
   What do we need to know about prefix : i?

Correct:

- 1. subproblem = survive? in suffix i:
  given initial column occupancies $h_0, h_1, \cdots, h_{w-1}$, call it $\boldsymbol{h}$
  $\implies$ # subproblems $= O(n \cdot h^w)$
- 3. recurrence: $DP[i, \boldsymbol{h}] = \max(DP[i, \boldsymbol{m}]$ for valid moves $\boldsymbol{m}$ of piece i in $\boldsymbol{h})$
  $\implies$ time per subproblem $= O(w)$
- 4. topo. order: for i in reversed(range(n)): for $\boldsymbol{h} \cdots$
  total time $= O(nwh^w)$ (DAG as above)
- 5. solution $= DP[0, \mathbf{0}]$
  (& use parent pointers to recover moves)

Super Mario Bros

Platform Video Game

- given entire level (objects, enemies, ...) ($\leftarrow n$)
- small $w \times h$ screen
- configuration
  - screen shift ($\leftarrow n$)
  - player position & velocity ($O(1)$) ($\leftarrow w$)
  - object states, monster positions, etc. ($\leftarrow c^{w \cdot h}$)
  - anything outside screen gets reset ($\leftarrow c^{w \cdot h}$)
  - score ($\leftarrow S$)
  - time ($\leftarrow T$)
- transition function δ: (config, action) $\rightarrow$ config'
  nothing, $\uparrow, \downarrow, \leftarrow, \rightarrow$, B, A press/release

(1) subproblem: best score (or time) from config. C
$\implies n \cdot c^{w \cdot h} \cdot S \cdot T$ subproblems

(2) guess: next action to take from C
$\implies O(1)$ choices

(3) recurrence:

$$DP(C) = \begin{cases} C.\text{score} & \text{if on flag} \\ \infty & \text{if } C.\text{dead or } C.\text{time} = 0 \\ \max(DP(\delta(C, A))) & \text{for } A \text{ in actions} \end{cases}$$

$\implies O(1)$ time/subproblem

(4) topo. order: increasing time

(5) orig. prob.: DP(start config.)

- pseudopolynomial in S & T
- polynomial in n
- exponential in $w \cdot h$

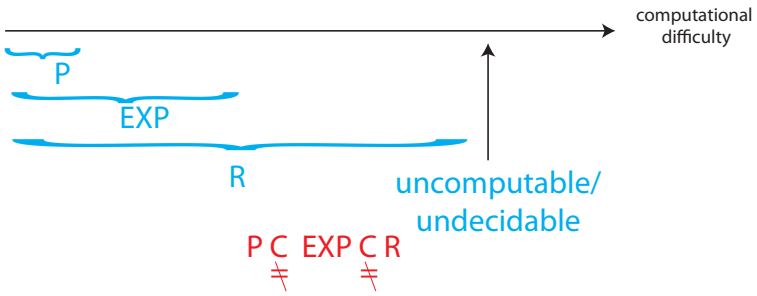
Lecture 23: Computational Complexity

Lecture Overview

- P, EXP, R
- Most problems are uncomputable
- NP
- Hardness & completeness
- Reductions

Definitions:

- P = {problems solvable in polynomial (n^c) time}
  (what this class is all about)
- EXP = {problems solvable in exponential (2^{n^c}) time}
- R = {problems solvable in finite time} "recursive" [Turing 1936; Church 1941]



Examples

- negative-weight cycle detection ∈ P
- $n \times n$ Chess ∈ EXP but ∉ P
  Who wins from given board configuration?
- Tetris ∈ EXP but don't know whether ∈ P
  Survive given pieces from given board.

Halting Problem:

Given a computer program, does it ever halt (stop)?

- uncomputable ($\notin$ R): no algorithm solves it (correctly in finite time on all inputs)
- decision problem: answer is YES or NO

Most Decision Problems are Uncomputable

- program $\approx$ binary string $\approx$ nonneg. integer $\in N$
- decision problem = a function from binary strings ($\approx$ nonneg. integers) to {YES (1), NO (0)}
- $\approx$ infinite sequence of bits $\approx$ real number $\in \mathbb{R}$
  $|\mathbb{N}| \ll |\mathbb{R}|$: no assignment of unique nonneg. integers to real numbers ($\mathbb{R}$ uncountable)
- $\implies$ not nearly enough programs for all problems
- each program solves only one problem
- $\implies$ almost all problems cannot be solved

NP

NP = {Decision problems solvable in polynomial time via a "lucky" algorithm}. The "lucky" algorithm can make lucky guesses, always "right" without trying all options.

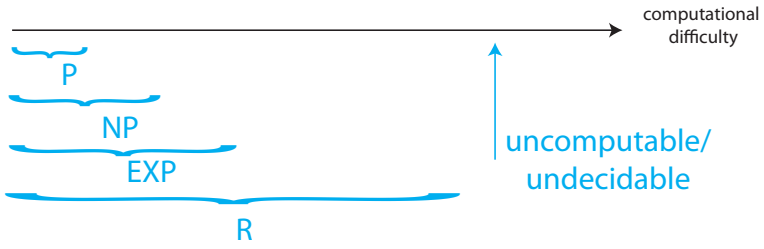
- nondeterministic model: algorithm makes guesses & then says YES or NO
- guesses guaranteed to lead to YES outcome if possible (no otherwise)

In other words, NP = {decision problems with solutions that can be "checked" in polynomial time}. This means that when answer = YES, can "prove" it & polynomial-time algorithm can check proof

Example

Tetris $\in$ NP

- nondeterministic algorithm: guess each move, did I survive?
- proof of YES: list what moves to make (rules of Tetris are easy)



P ≠ NP

Big conjecture (worth $1,000,000)

- ≈ cannot engineer luck
- ≈ generating (proofs of) solutions can be harder than checking them

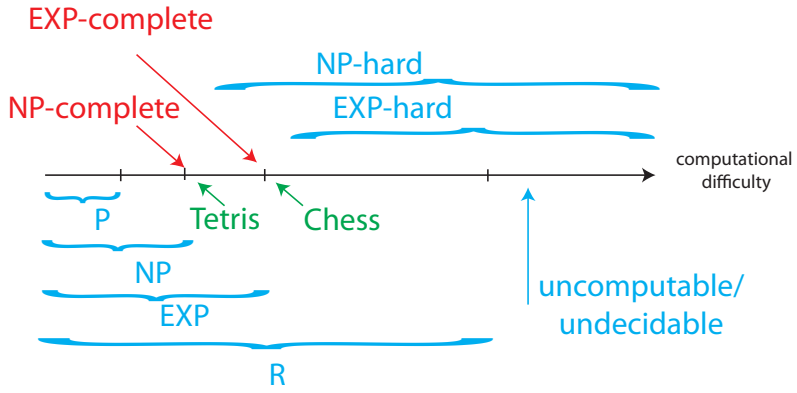
Hardness and Completeness

Claim:

If P ≠ NP, then Tetris ∈ NP - P
[Breukelaar, Demaine, Hohenberger, Hoogeboom, Kosters, Liben-Nowell 2004]

Why:

Tetris is NP-hard = "as hard as" every problem ∈ NP. In fact NP-complete = NP ∩ NP-hard.



Similarly

Chess is EXP-complete = EXP ∩ EXP-hard. EXP-hard is as hard as every problem in EXP. If NP ≠ EXP, then Chess ∉ EXP \ NP. Whether NP ≠ EXP is also an open problem but less famous/"important".

Reductions

Convert your problem into a problem you already know how to solve (instead of solving from scratch)

- most common algorithm design technique
- unweighted shortest path → weighted (set weights = 1)
- min-product path → shortest path (take logs) [PS6-1]
- longest path → shortest path (negate weights) [Quiz 2, P1k]
- shortest ordered tour → shortest path (k copies of the graph) [Quiz 2, P5]
- cheapest leaky-tank path → shortest path (graph reduction) [Quiz 2, P6]

All the above are One-call reductions: A problem → B problem → B solution → A solution
Multicall reductions: solve A using free calls to B — in this sense, every algorithm reduces problem → model of computation
NP-complete problems are all interreducible using polynomial-time reductions (same difficulty). This implies that we can use reductions to prove NP-hardness — such as in 3-Partition → Tetris

Examples of NP-Complete Problems

- Knapsack (pseudopoly, not poly)
- 3-Partition: given n integers, can you divide them into triples of equal sum?
- Traveling Salesman Problem: shortest path that visits all vertices of a given graph — decision version: is minimum weight $\leq x$?
- longest common subsequence of k strings
- Minesweeper, Sudoku, and most puzzles
- SAT: given a Boolean formula (and, or, not), is it ever true? x and not x → NO
- shortest paths amidst obstacles in 3D

- 3-coloring a given graph
- find largest clique in a given graph

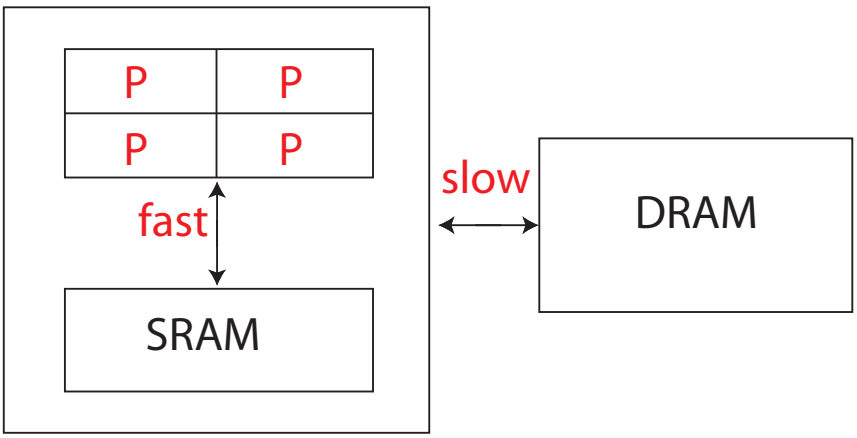
Lecture 24: Parallel Processor Architecture & Algorithms

Processor Architecture

Computer architecture has evolved:

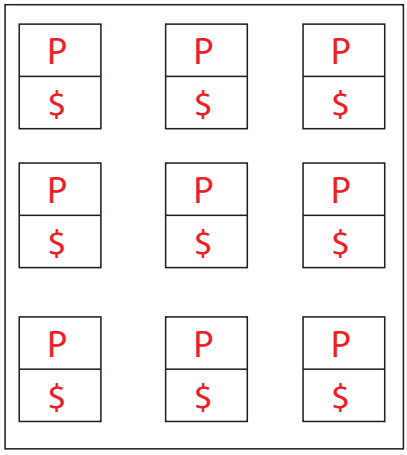
- Intel 8086 (1981): 5 MHz (used in first IBM PC)
- Intel 80486 (1989): 25 MHz (became i486 because of a court ruling that prohibits the trademarking of numbers)
- Pentium (1993): 66 MHz
- Pentium 4 (2000): 1.5 GHz (deep $\approx$ 30-stage pipeline)
- Pentium D (2005): 3.2 GHz (and then the clock speed stopped increasing)
- Quadcore Xeon (2008): 3 GHz (increasing number of cores on chip is key to performance scaling)

Processors need data to compute on:



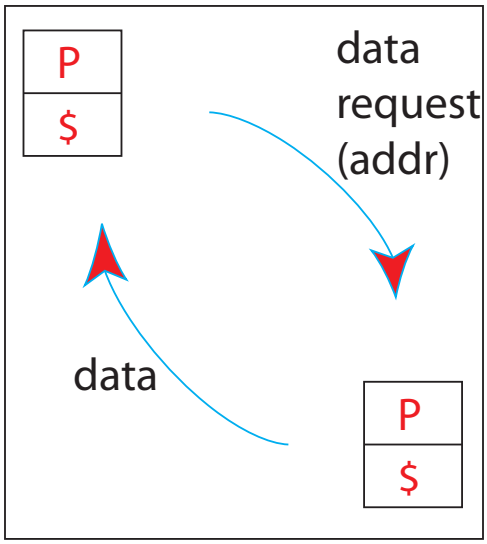
Problem: SRAM cannot support more than ≈ 4 memory requests in parallel.

$: cache P: processor



Most of the time program running on the processor accesses local or "cache" memory

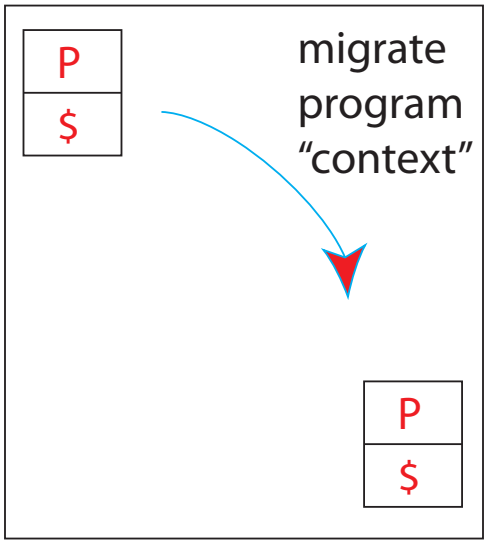
Every once in a while, it accesses remote memory:



Round-trip required to obtain data

Research Idea: Execution Migration

When program running on a processor needs to access cache memory of another processor, it migrates its "context" to the remote processor and executes there:



One-way trip for data access

Context = $\underbrace{\text{ProgramCounter} + \text{RegisterFile} + \ldots}_{\text{fewKbits}}$ (can be larger than data to be accessed)

Assume we know or can predict the access pattern of a program

$m_1, m_2, \ldots, m_N$ (memory addresses)

$p(m_1), p(m_2), \ldots p(m_N)$ (processor caches for each m_i)

Example

$p_1\ p_2\ p_2\ p_1\ p_1\ p_3\ p_2$

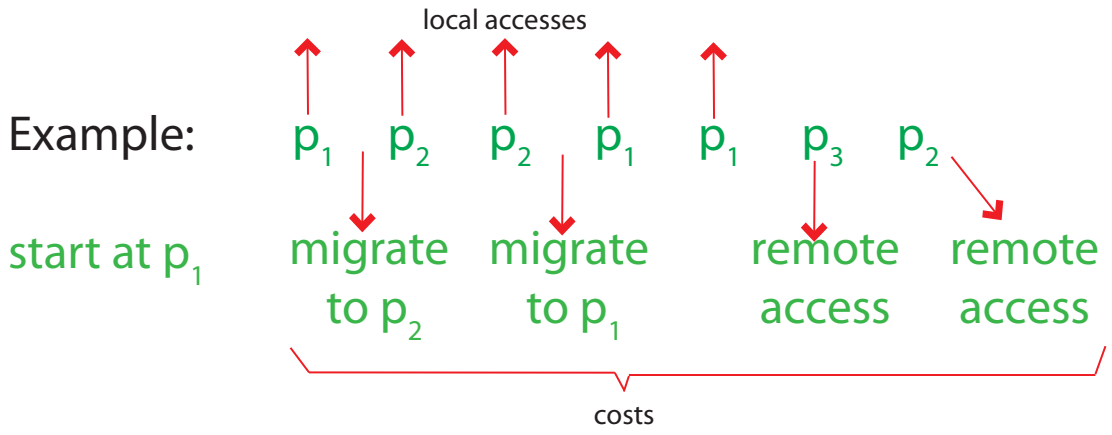
$\text{cost}_{\text{mig}}(s, d) = \text{distance(s, d)} + \text{L}$ ← load latency L is a function of context size

$\text{cost}_{\text{access}}(s, d) = 2 * \text{distance(s, d)}$

if $s == d$, costs are defined to be 0

Problem

Decide when to migrate to minimize total memory cost of trace For example:



What can we use to solve this problem?
Dynamic Programming!

Dynamic Programming Solution

Program at p, initially, number of processors $= Q$

Subproblems?

Define $DP(k, p_i)$ as cost of optimal solution for the prefix $m_1, \ldots, m_k$ of memory accesses when program starts at p_1 and ends up at p_i.

$$DP(k+1, p_j) = \begin{cases} DP(k, p_j) + \text{cost}_{\text{access}}(\text{p}_\text{j}, \text{p}(\text{m}_{\text{k+1}})) & \text{if } p_j \neq p(m_{k+1}) \\ MIN_{i=1}^{Q}(DP(k, p_i) + \text{cost}_{\text{mig}}(\text{p}_\text{i}, \text{p}_\text{j})) & \text{if } p_j = p(m_{k+1}) \end{cases}$$

Complexity?

$$O(\underbrace{N \cdot Q}_{\text{no.of subproblems}} \cdot \underbrace{Q}_{\text{cost per subproblem}}) = O(NQ^2)$$

My research group is building a 128-processor Execution Migration Machine that uses a migration predictor based on this analysis.

Lecture 24: Research Areas and Beyond 6.006

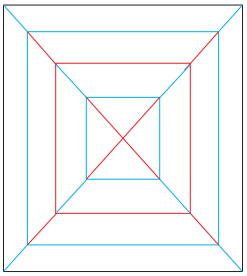
Erik's Main Research Areas

- computational geometry [6.850]
  - geometric folding algorithms [6.849]
  - self-assembly
- data structures [6.851]
- graph algorithms [6.899]
- recreational algorithms [SP.268]
- algorithmic sculpture

Geometric Folding Algorithms: [6.849], Videos Online

Two aspects: design and foldability

- design: algorithms to fold any polyhedral surface from a square of paper [Demaine, Demaine, Mitchell (2000); Demaine & Tachi (2011)]
  - bicolor paper $\implies$ can 2-color faces
  - OPEN: how to best optimize "scale-factor"
  - e.g. best $n \times n$ checkerboard folding — recently improved from $\approx n/2 \rightarrow \approx n/4$
- foldability: given a crease pattern, can you fold it flat
  - NP-complete in general Bern & Hayes (1996)
  - OPEN: $m \times n$ map with creases specified as mountain/valley [Edmonds (1997)]
  - just solved: $2 \times n$ Demaine, Liu, Morgan (2011)
  - hyperbolic paraboloid [Bauhaus (1929)] doesn't exist [Demaine, Demaine, Hart, Price, Tachi (2009)]



- understanding circular creases
- any straight-line graph can be made by folding flat & one straight cut [Demaine, Demaine, Lubiw (1998); Bern, Demaine, Eppstein, Hayes (1999)]

Self-Assembly

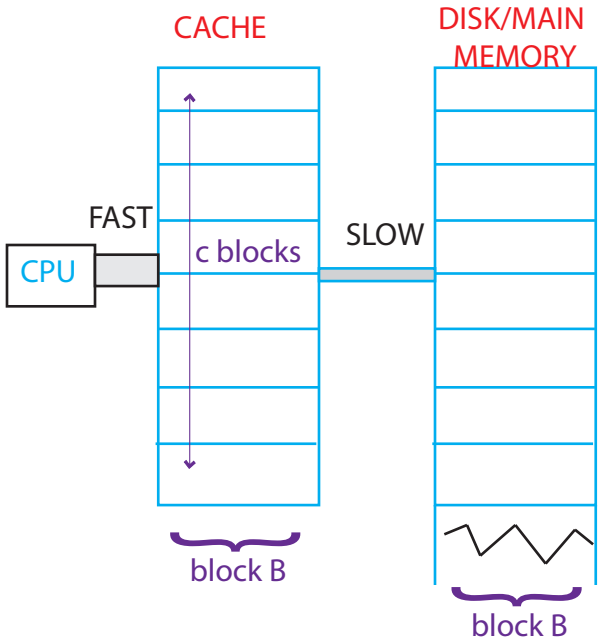
Geometric model of computation

- glue e.g. DNA strands, each pair has strength
- square tiles with glue on each side
- Brownian motion: tiles/constructions — stick together if $\sum$glue strengths $\geq$ temperature
- can build $n \times n$ square using $O\left(\frac{\lg n}{\lg \lg n}\right)$ tiles [Rothemund & Winfree 2000] or using $O(1)$ tiles & $O(\lg n)$ "stages" algorithmic steps by the bioengineer [Demaine, Demaine, Fekete, Ishaque, Rafalin, Schweller, Souvaine (2007)]
- can replicate ∞ copies of given unknown shape using $O(1)$ tiles and $O(1)$ stages [Abel, Benbernou, Damian, Demaine, Flatland, Kominers, Schweller (2010)]

Data Structures: [6.851], Videos Next Semester

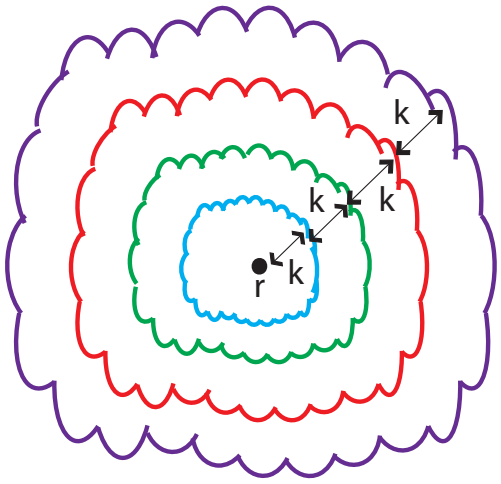
There are 2 main categories of data structures

- Integer data structures: store n integers in $\{0, 1, \cdots u-1\}$ subject to insert, delete, predecessor, successor (on word RAM)
  - hashing does exact search in $O(1)$
  - AVL trees do all in $O(\lg n)$
  - $O(\lg \lg u)$/op van Emde Boas
  - $O\left(\frac{\lg n}{\lg \lg u}\right)$/op fusion trees: Fredman & Willard
  - $O\left(\sqrt{\frac{\lg n}{\lg \lg n}}\right)$/op min of above
- Cache-efficient data structures
  - memory transfers happen in blocks (from cache to disk/main memory)
  - searching takes $\Theta(\log_B N)$ transfers (vs. $\lg n$)
  - sorting takes $\Theta\left(\frac{N}{B} \log_C \frac{N}{B}\right)$ transfers
  - possible even if you don't know B & C !



(Almost) Planar Graphs: [6.889], Videos Online

- Dijkstra in $O(n)$ time [Henzinger, Klein, Rao, Subramanian (1997)]
- Bellman-Ford in $O\left(\frac{n \lg^2 n}{\lg \lg n}\right)$ time [Mozes & Wolff-Nilson (2010)]
- Many problems NP-hard, even on planar graphs. But can find a solution within $1 + \varepsilon$



factor of optimal, for any ϵ [Baker 1994 & Others]:

  - run BFS from any root vertex r
  - delete every k layers
  - for many problems, solution messed up by only $1 + \frac{1}{k}$ factor ($\implies k = \frac{1}{\varepsilon}$)
  - connected components of remaining graph have $< k$ layers. Can solve via DP typically in $\approx 2^k \cdot n$ time

Recreational Algorithms

- many algorithms and complexities of games [some in SP.268 and our book *Games, Puzzles & Computation* (2009)]
- $n \times n \times n$ Rubik's Cube diameter is $\Theta\left(\frac{n^2}{\lg n}\right)$ [Demaine, Demaine, Eisenstat, Lubiw, Winslow (2011)]
- Tetris is NP-complete [Breukelaar, Demaine, Hohenberger, Hoogeboom, Kosters, Liben-Nowell (2004)]
- balloon twisting any polyhedron [Demaine, Demaine, Hart (2008)]
- algorithmic magic tricks

Algorithms Classes at MIT: (post 6.006)

- 6.046: Intermediate Algorithms (more advanced algorithms & analysis, less coding)
- 6.047: Computational Biology (genomes, phylogeny, etc.)
- 6.854: Advanced Algorithms (intense survey of whole field)
- 6.850: Geometric Computing (working with points, lines, polygons, meshes, . . .)
- 6.849: Geometric Folding Algorithms origami, robot arms, protein folding, . . .
- 6.851: Advanced Data Structures (sublogarithmic performance)
- 6.852: Distributed Algorithms (reaching consensus in a network with faults)
- 6.853: Algorithmic Game Theory (Nash equilibria, auction mechanism design, . . .)
- 6.855: Network Optimization (optimization in graph: beyond shortest paths)
- 6.856: Randomized Algorithms (how randomness makes algorithms simpler & faster)
- 6.857: Network and Computer Security (cryptography)

Other Theory Classes:

- 6.045: Automata, Computability, & Complexity
- 6.840: Theory of Computing
- 6.841: Advanced Complexity Theory
- 6.842: Randomness & Computation
- 6.845: Quantum Complexity Theory
- 6.440: Essential Coding Theory
- 6.441: Information Theory

Top 10 Uses of 6.006 Cushions

10. Sit on it: guaranteed inspiration in constant time
    (bring it to the final exam)
9. Frisbee (after cutting it into a circle)*
8. Sell as a limited-edition collectible on eBay
    (they'll probably never be made again—at least $5)
7. Put two back-to-back to remove branding*
    (so no one will ever know you took this class)
6. Holiday conversation starter… and stopper
    (we don't recommend re-gifting)
5. Asymptotically optimal acoustic paneling
    (for practicing piano & guitar fingering DP)
4. Target practice for your next LARP*
    (Live Action Role Playing)
3. Ten years from now, it might be all you'll remember about 6.006
    (maybe also this top ten list)
2. Final exam cheat sheet*
1. *Three words:* OkCupid profile picture