6.006 Lecture #1

Summer

Course Overview:

EFFicient procedures for solving problems with large (trillion) inputs.

Classic Data Structures - Binary Search Tree, Hash Tables + classical algorithms - sorting, matching etc

Real implementations in Python

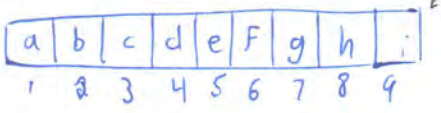
Content

8 modules:

- Algorithmic thinking: peek Finding
- Sorting and trees: Event simulation
- Hashing: Genome comparison
- Numerics: RSA encryption
- Graphs: Rubik's Cube
- Shortest path: Caltech → MIT
- Dynamic programming: Image compression
- Advanced Topics:

Peak Finder

One dimensional version



← Array of #'s

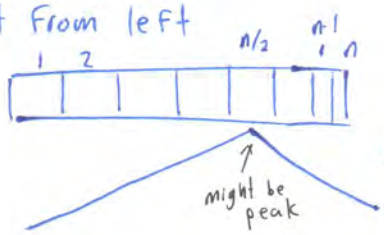
Position 2 is peek iFF $b \geq a$ and $b \leq c$

Position 9 is peak iF $i \geq h$

Problem: Find the peak, iF it exists

Straightforward algorithm:

Start From left



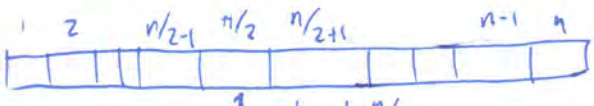
Look at $n/2$ elements

worse-case complexity

$\Theta(n)$

↖ Look at all elements, linear.

Recursive algorithm



Divide and 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 look at $n/2+1 \ldots n$ for peak

else: $n/2$ position is a peak.

(Argue that algorithm is correct)

Complexity:

$T(n) = T(n/2) + \Theta(1)$ ← comparison, left and write

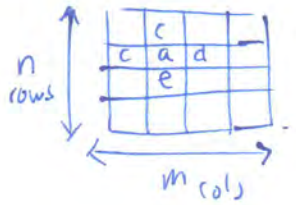
↑ "work done on input of size n"

base case: $T(1) = \Theta(1)$

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

- Faster and more efficient

2D Version



a is a 2D peak IFF

$a \geq b,\ a \geq d,\ a \geq c,\ a \geq e$

Greedy Ascent algorithm

$\Theta(mn)$ complexity

$\Theta(n^2)$ if $m = n$

Divide and conquer

- Pick middle column $j = m/2$

Find 1D peak at (i, j)

Use (i, j) as a start to find 1D peak on row i

Incorrect but efficient

Why it doesn't work.

Problem: 2D peak may not exist on row i

Attempt #2

Pick middle column $j = m/2$

Find global max on col j at (i, j)

Compare $(i, j-1), (i, j), (i, j+1)$

Pick left cols if $(i, j-1) > (i, j)$. Similarly for right

if $(i, j) \geq (i, j-1), (i, j+1) \Rightarrow (i, j)$ is a 2D peak

Solve the new problem w/ half the # of cols

When you have single col, find global max ← done

Complexity

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

($m/2$: half # of cols; $\Theta(n)$: Find global max)

$$T(n, 1) = \Theta(n)$$

$$T(n, m) = \underbrace{\Theta(n) \ldots + \Theta(n)}_{\log_2 m} = \Theta(n \log_2 m)$$

9/4/14

6.006 Intro to Algorithms

Fernando Trujano

Lecture #1

Algorithm - Finite, efficient precise sequences of instructions

Phonebook $A[1 \dots n]$ sorted

$M/2$ repeat

$O(1)$

$T(M)$ ← time with M entries $= T(\frac{m}{2}) + O(1)$

$= T(\frac{m}{4}) + O(1) + O(1)$

Peek Finding $A[1 \dots m]$ unsorted

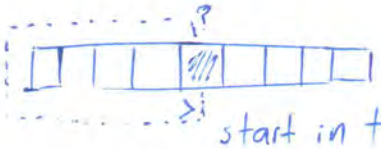
local max

15	13	5	8	3	2	1

$O(n)$

$A[i]$ Peek if no smaller than its neighbors

Find a peak



start in the middle, move in direction of positive gain

$O(\log n) \Leftarrow T(m) = T(\frac{m}{2}) + O(1)$ ← recursive; what you do on left/right border

* Disregard half of the problem in a constant # of steps

Divide and conquer

1) Divide the problem into DISJOINT parts
2) Conquer each part SEPARATELY
3) Occasionally, you combine

2D Peek Finding $A[1 \dots n, 1 \dots m]$

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

1) Run Peek algorithm in one direction $O(m)$

Global col max

10	13	5

$O(m \cdot n) \rightarrow O(m \log(n))$

6.006 Notes Chapter #3

Fernando Trujano

Order of growth

- Only consider the leading term in the Formula, ignore it's constant.

recurrance - running time of a recursive algorithm

logarithmic grows much slower than linear

Θ notation - asymptotically bounds a function From above and below.

O notation - only For asymptotic upper bound.

Ω notation - only For asymptotic lower bound

o notation - upperbound, not asymdotecally tight

ω notation - lower bound, ''

Modular Arithmetic: $a \bmod n = a - n\left[\frac{a}{n}\right]$

$a \equiv b \pmod n$ means $a \bmod n = b \bmod n$

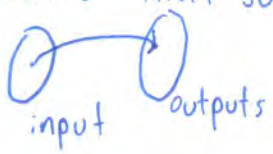
Today: { Models / cost } of computation.

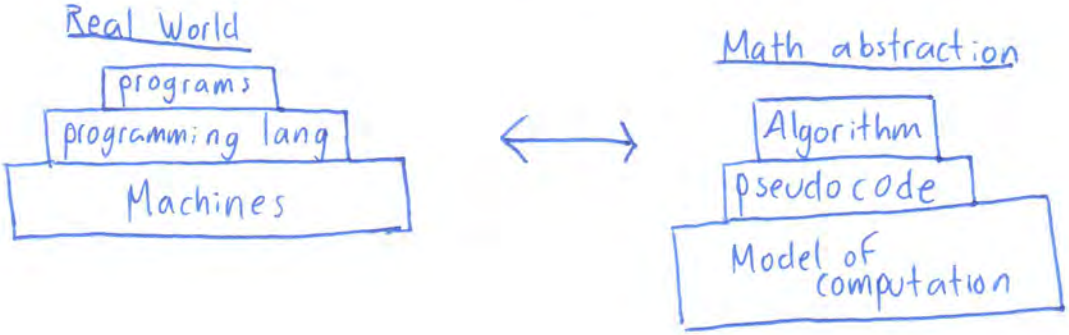
- what is an algorithm?
- what is time?
- models, RAM, pointer machine, python
  ex. doc distance

Guiding Principle

1) Work at the right level of abstraction
2) Asymptotic Analysis ("Big-Oh")
3) Worst-case analysis

Algorithm - computational procedure that solves a problem.





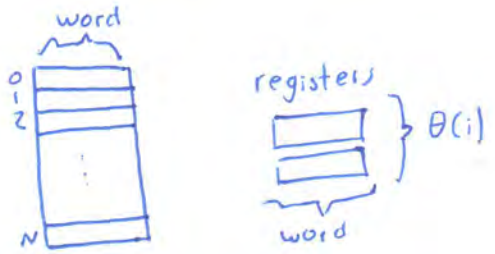
Model of computation

1. What are the basic operations
2. Cost of each operation (time)

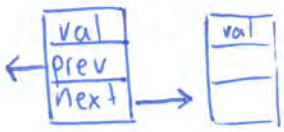
① Random Access Machine
② Pointer Machine
*③ Python Model

Random Access Machine (RAM)

- Modeled by an array
- Operations
  - load/store words } Θ(1) time
  - compute on two words }



Pointer Machine Model



- object with $\Theta(1)$ (constant) # of Fields
- dynamically allocate object in $\Theta(1)$ time

linked list?

Pros: Easy to add element $\Theta(1)$ time
Cons: slow to go to index
- has to traverse through list.

Python Model = RAM + Pointer

"list" = array

1. List[i] = L[j] + 5 - $\Theta(1)$ time
2. L.append(x) - $\Theta(1)$ time
3. L1.extend(L2) (concatenate) - $\Theta(|L2|)$ (length of L2)
4. L = L1 + L2 - $\Theta(|L1| + |L2|)$
5. x in L? - $\Theta(|L|)$
6. L.sort() - $\Theta(|L| \log |L|)$

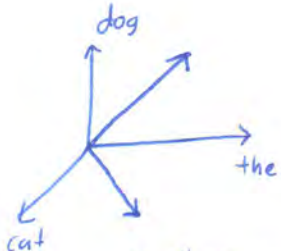
Example. Document Distance

$d(D_1, D_2)$ ← distance of Document

doc = list of words

= (D[w] = # occurances of w in D) (vector)

1 dimension per word



D1 "the cat"
D2 "the dog"

$$d = \frac{D_1 \cdot D_2}{\|D_1\| \|D_2\|} = \frac{\sum_w D_1[w] \cdot D_2[w]}{\|D_1\| \cdot \|D_2\|}$$

Goal: Given docs D1 and D2 as Files, compute the distance $d(D_1, D_2)$

① Split Document into words

② Compute Frequency

③ Dot Product

4/10/14 6.006 Recitation Fernando Trujano

1. Python Cost Model
   - ↳ Lists
   - ↳ Dictionaries
2. Document Distance
   - ↳ Definition
   - ↳ Implementation

Python Cost Model

Lists

Implemented as an array ↓ ordered

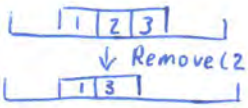
Takes one continious chunck in memory

Append: $\Theta(1)$ ← constant

Search: $\Theta(n)$ ← linear

Remove: $\Theta(n)$

For last element: $\Theta(1)$



-> has to rearange array

Insert: $\Theta(n)$

Splicing: $\Theta(b-a) = \Theta(n)$

L[a:b]

Dictionaries

Implemented as Hashtables ↓ unordered

Append: $\Theta(1)$

Search: $\Theta(1)$

Remove: $\Theta(1)$

Document Distance

D_i = vector of frequencies of words in doc i

Document 1 ("the dog")

Document 2 ("the cat")

D_1 = [["the", 1], ["dog", 1]]

D_2 = [["the", 1], ["cat", 1]]

$$d = \arccos\left(\frac{D_1 \cdot D_2}{\|D_1\| \, |D_2|}\right)$$

Inner product

$\sqrt{D_1 \cdot D_1}$ ← Norm

- Sum of products of Frequencies

"the" "cat" "dog"

$1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 = 1$

To compute d

1) Get vector From document
   get_word_counts(s)
2) Compute some inner products
   get_inner_product(v1, v2)
3) Do some computation

Take 1:

```
def get_word_counts(s):
    words = s.split(' ')                    O(n)
    counts = []
    For word in words:                      O(n)
        found = False
        for i in range(len(counts)):        O(n)
            if word == counts[i][0]:
                counts[i][1] += 1
                found = True
        if not found:
            counts.append([word, 1])
    return counts
```

Complexity: $O(n^2+n) = O(n^2)$

```
def get_inner_product(v1, v2)
    inner = 0
    for w1 in v1:
        for w2 in v2:
            if w1[0] == w2[0]: inner += w1[1] * w2[1]
    return inner
```

Complexity: $O(v1 \cdot v2)$

Take 1.5

```
def get_inner_product
    sort both lists #l    O(v1 log(v1) + v2 log(v2))
    Find intersections    O(v1+v2)
```

Take 2

```
def get_word_counts(s):
    words = s.split(' ')
    counts = {}
    For word in words:                    O(n)
        if word in counts:
            counts[word] += 1             O(1)
        else:
            counts[word] = 1
    return counts
```

Complexity: O(n)

Best possible solution. You at least have to read the whole input

```
def get_inner_product(v1, v2):
    inner = 0
    For w in v1:
        if w in v2:
            inner += v1[w] · v2[w]
    return inner
```

Complexity O(v1)

9/11/14 6.006 Lecture #3 Fernando Trujano

Sorting

- Why?
- Insertion sort
- Merge Sort (Divide + Conquer)
- Recurrance sorting

Why?

- Problems can be easier to solve with a sorted list.

	sorted	unsorted
search	$\Theta(\lg n)$	$\Theta(n)$
median	$\Theta(1)$	$\Theta(n)$
element distinctness	$\Theta(n)$	

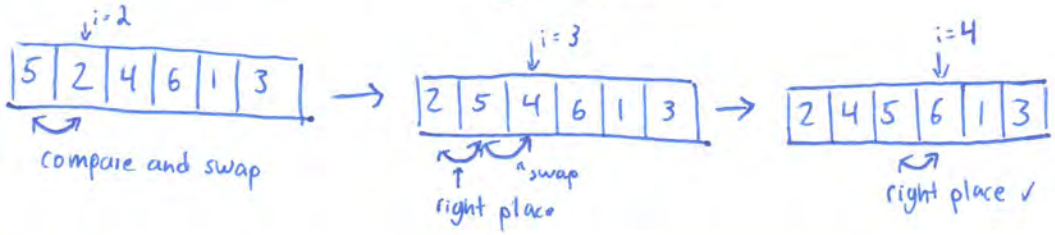
Insertion Sort

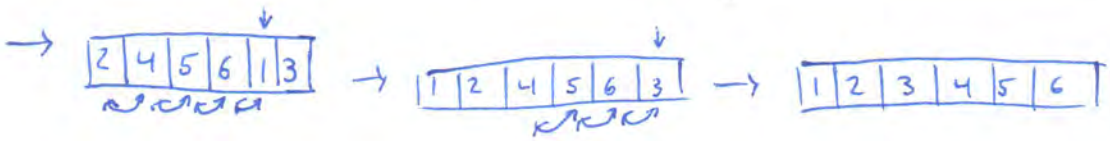
For i=1 to n

Insert A[i] into the "right position" into already sorted array A[1:i-1] by pairwise swaps down to the right position.

★ First i-1 items already sorted

Example:





Worst-case running time: $\sum_{i=1}^{n} i = \Theta(n^2)$

Best case: $\Theta(n)$ (Already sorted array)

say cost(compare) >> cost(swap) ← (Objects are verry long)

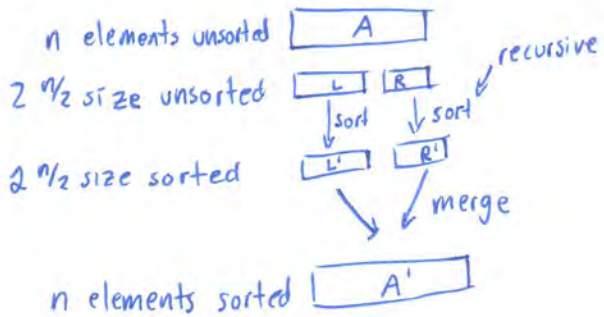
Do binary search to find right position.
Swap everything else to readjust.

compares $\sum_{i=1}^{n} \lg i = \Theta(n \log n)$

swaps $\Theta(n^2)$

Merge Sort (Divide and Conquer)

MergeSort(A):



Merge

L'	R'
20	12
13	11
7	9
2	1

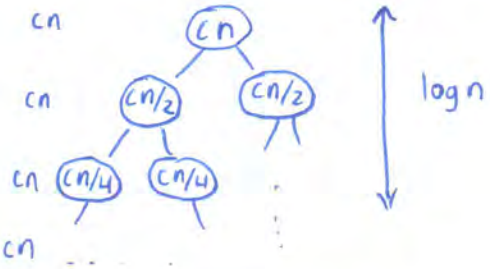
2 pointers
- compare
- write down element
- move pointer up

1 2 7 9 11 12 13 20

$\Theta(n)$ time for merge

$$T(n) = \underbrace{c_1 n}_{\text{divide}} + \underbrace{2T(n/2)}_{\text{conquer}} + \underbrace{c_2 n}_{\text{merge}}$$

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



$\approx cn \log n$

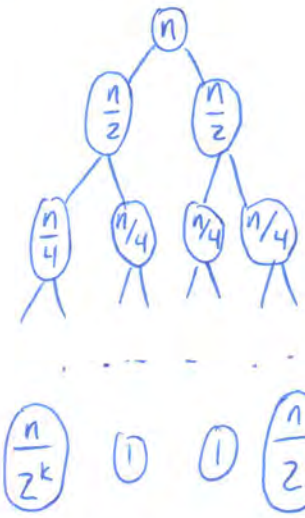
Insertion sort vs. Merge sort

1) Array is "almost sorted"

9/12/14 6.006 Recitation Fernando Trujano

Recurrance Solving

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



work
n
n
n
n

$= \theta(n \log n)$

Case 2

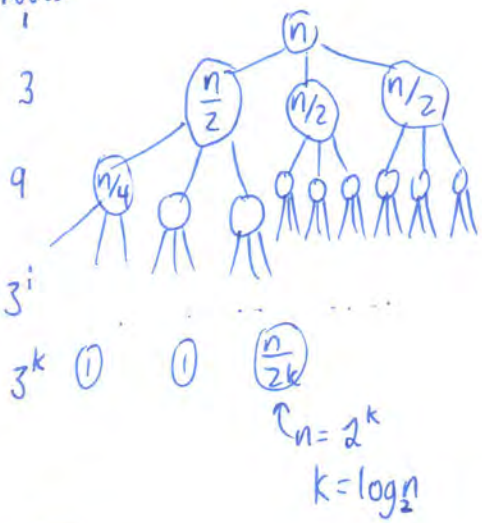
$n = 2^k$

$k = \log n$

$T(n) = 3T(\frac{n}{2}) + n$ ← work per step

$= \theta(n^{\log_2 3})$

Case 1



$n = 2^k$

$k = \log_2 n$

work
n
$3/2\, n$
$9/4\, n$
$(\frac{3}{2})^i n$

$(\frac{3}{2})^k n = 3^k = 3^{\log_2 n} = 3^{\frac{\log_3 n}{\log_3 2}} = (3^{\log_3 n})^{\frac{1}{\log_3 2}}$

change of base formula

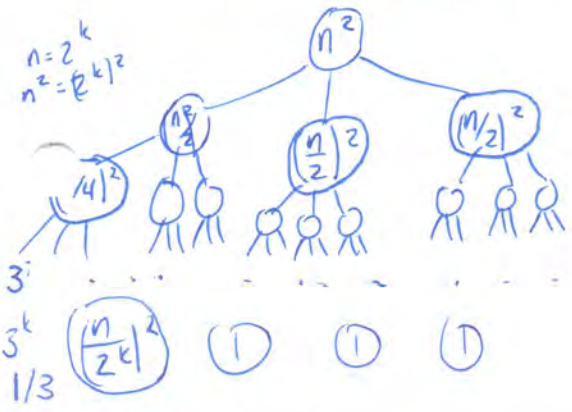
$= n^{\frac{1}{\log_3 2}}$

$= n^{\frac{\log 3}{\log 2}}$

$= n^{\log_2 3}$

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

$= \theta(n^2)$ Case 3



$n = 2^k$

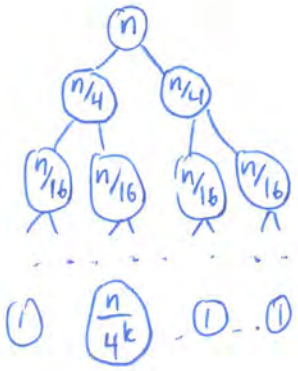
$n^2 = (2^k)^2$

work
n^2
$\frac{3n^2}{4}$
$\frac{9n^2}{16}$
$(\frac{3}{4})^i n^2$

$(3/4)^k n^2 = 3^k$

$T(x,y) = 2T\left(\frac{x}{4}, \frac{y}{2}\right) + x$ ← x dominates runtime ; ← branching factor $= \Theta(n)$

$T(n,n)$



work
n
$n/2$
$n/4$
$\frac{n}{2^i}$
$n/2^k$

Master Theorem

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$ ← must be in this form. a, b constant

$\varepsilon > 0$

$(\log n)^k$

Bottom level dominates ⇒ Case 1: $f(n) = O(n^{\log_b a - \varepsilon})$: $T(n) = \Theta(n^{\log_b a})$

No dominance ⇒ Case 2: $f(n) = O(n^{\log_b a} \log^k n)$: $T(n) = \Theta(n^{\log_b a} \log^{k+1} a)$

Top level dominates ⇒ Case 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$: $T(n) = \Theta(f(n))$

Example

$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2)$ $\quad n^{\log_2 4} = n^2$

1) Compare n^2 to $n^{\log_b a}$

$n^2 = n^2$: Tie ⇒ Case 2 $\quad k=0$

$= \Theta(n^2 \log n)$

$T(n) = 7T\left(\frac{n}{2}\right) + n^3$

1) $\log_2 7 \approx 2.8$ ← less than 3 so n^3 will dominate

$= \Theta(n^3)$

$T(n) = 2T(\sqrt{n}) + \log n$

* Can't use master theorem
* use change of variables

$n = 2^m$

$T(2^m) = 2T(\sqrt{2^n}) + \log 2^m$

$T(2^m) = 2T(2^{m/2}) + m$

$s(m) = T(2^m)$

$s(m) = 2s(\frac{m}{2}) + m$

* Can now invoke the master theorem

-> case 2

$T(n) = T(2^m) = s(m) = \Theta(m \log m)$

$m = \log n$

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

6.006 Lecture #4

Fernando Trujano

Data Structures (live!)

Dynamic Set + Ops

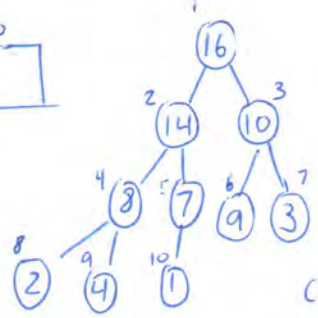
Priority Queue
- insert (s, x)
- Find max/min (s)
- extract max/min (s)

Heap Q: how to represent S?

A: Array $A = A[1 \ldots n]$ (var)

* visualised as a complete binary Tree *

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



left(i) = 2i
right(i) = 2i+1

Change of vars!

Heap $\triangleq \forall i: A[i] \geq \max\{A[i], A[2i+1]\}$ (makes it easier to maintain.)

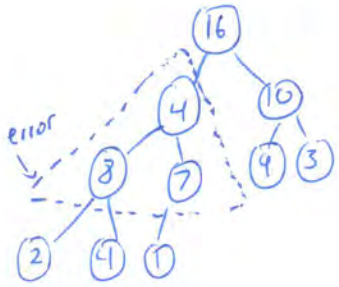
Great idea: Maintain a weak invariant

- Find max O(1) ✓
- extract max
- insert
- Build O(n)

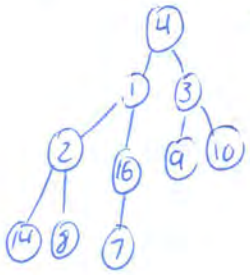
heepify: Given a heap with 1 single error restores the heap

- swap with biggest children
- new array can have at most one error. → heepify → recursion

O(log n)



Build Heep



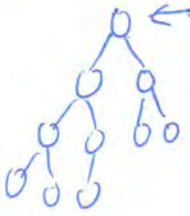
not a heep!

Do heapify for bad parents.

How long $n \log n$

$O(n)$

Extract Max



remove max → replace with element at the end of the list

→ heapify!

$O(\log n)$

Insert

Add element at the end → heapify

$O(\log n)$

6.006 Notes Chapter #4

Master Method

Master Theorem:

Recurrence in Form: $T(n) = aT(n/b) + f(n)$

Cases:

1) $f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$

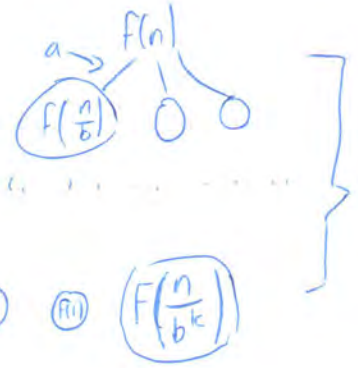
2) $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$

3) $f(n) = \Omega(n^{\log_b a + \epsilon}) \Rightarrow T(n) = \Theta(f(n))$

9/17/14 6.006 Recitation Fernando Trujano

Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + F(n)$$



$n = b^k$

$k = \log_b n$

$a^k \cdot c$ #nodes

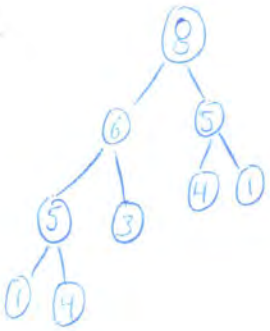
$= a^{\log_b n} \cdot c$

↓ Change of base

$= n^{\log_b a} \cdot c \Rightarrow \Theta(n^{\log_b a})$

Dominates

Heaps



8	6	5	5	3	4	1	1	4

left = 2i ← index

right = 2i + 1

Max: Look at root → 8 $O(1)$

Extract-max: Delete root → move last element to top (root) → heapify (takes logarithmic time, fixes a single error)

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

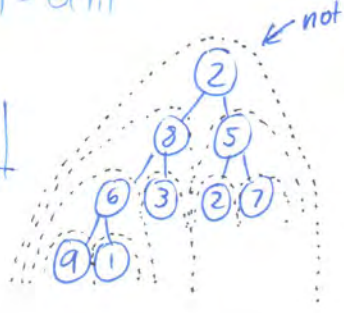
worst case: constant work for height of the tree ($\log n$)

Increase-key: $O(\log n) = \Theta(h)$ Go up the tree ↑

Insert: Add it to bottom (end of array) => heapify $O(\log n) = \Theta(h)$

Building max-heap:

2	8	5	6	3	2	7	9	1



Start at bottom and check that the heap condition holds For each sub-tree. IF it doesn't → heapify → Fixed.

Total work:

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot ch$$

ch: work per node

$\frac{n}{2^{h+1}}$: how many of the nodes at each level

Bottom level has 1/2 nodes
second to last has 1/4 of all the nodes
. . . .
$n/2^{h+1}$

$$= cn \sum_{h=0}^{\log n} \frac{h}{2^{h+1}} = \frac{1}{2} cn \sum_{h=0}^{\log n} \frac{h}{2^h} \leq \frac{1}{2} cn \sum_{h=0}^{\infty} \frac{h}{2^h}$$

$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$ b/c math

$$\leq cn = O(n)$$

Most of the nodes are near the bottom, where max heapify can Fix them in constant time.

Heap Sort

Take heap → extract max Until array is sorted

$O(n \log n)$

* Heaps are almost sorted. At least $(\log n)$ inversions

comparison sort = $\Omega(n \log n)$

k inversions:

Insertion sort $\Theta(nk)$

9/18/14

6.006 Lecture

Fernando Trujano

Heeps = Full binary tree s.t $\forall$ O 

Arrays $A[1 \ldots n]$ ← var = heep size

Find max, extract max, insert, construct

Application: Heepsort

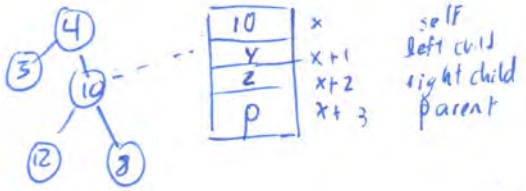
- Build a Heap O(n)
- Swap A[1] (= max) and A[n] O(1)
- discard n^{th} element From heep O(1)
- heepify O(logn)
- goto

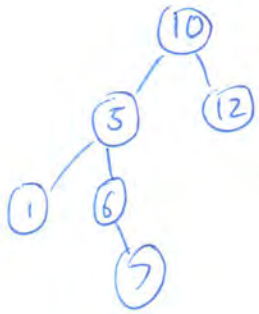
Total: O(n log n)

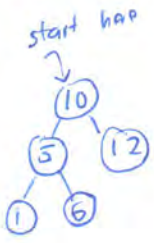
Useful For Finding k # of max

Arrays: continious memork 

LIST: 

Trees: 

Binary Search Trees 

Insert 10: 

Insert 12
" 5
" 1
6

Always inserting leafs!

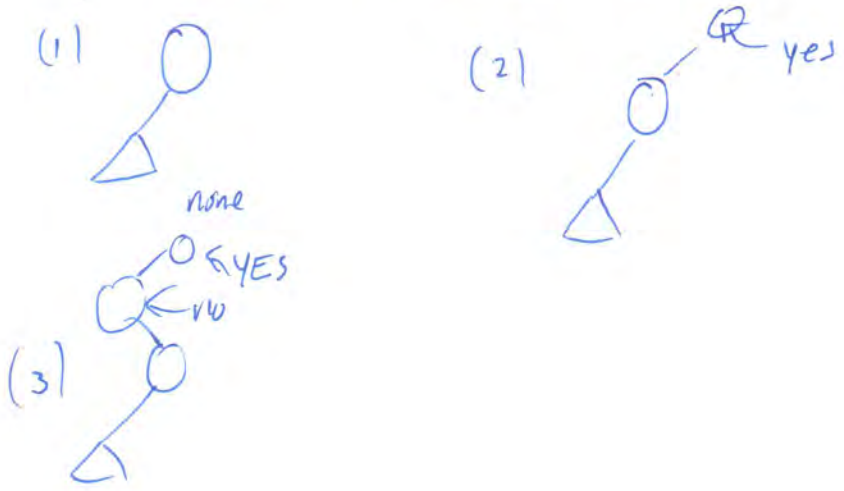
Operations:

Insert(k)	O(h)
Find(k)	O(h)
travel left — Find min(x)	O(h)
deletemin	O(h)

nextlarger(x) = smallest $\geq$ key[x]

If all distinct ✓

IF I have a righ child, then my succesor is the min of the right sub tree!



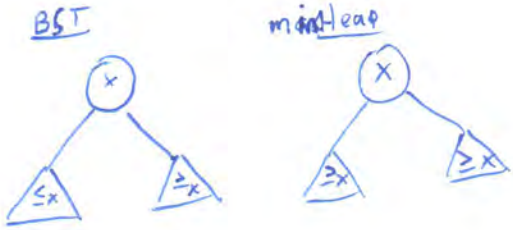
9/23/14

6.006 Lecture #6

Fernando Trujano

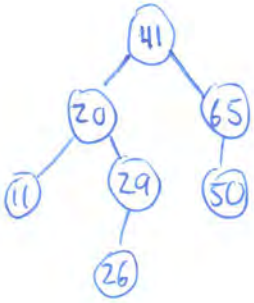
Balanced Binary Search Trees

- AVL Trees: def, rotations, insert
- heaps vs AVL trees



Recall BST

- rooted binary tree (2 leaves)
- each node has: key, left, right, parent



BST
- insert / delete
- Find min / max
- Find
- successor / pred

height of ~~tree~~ a node n h:

h_n = length of longest path from node to a leaf going down

height of tree:

height of root

max: $h = n-1$

min: $h = \log n$ or $\lceil \log(n+1) \rceil - 1$

AVL Tree

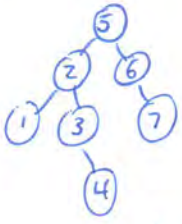
AVL property | For every node, heights of its right and left sub tree differ by at most 1.

$|h_\ell - h_r| \leq 1$

+ BST.

AVL trees have height $\Theta(\lg n)$

$h_{node} = \max(h_{node.right}, h_{node.left}) + 1$



height = 3
biggest height

Given n nodes, how large can h be?
<=> Given height h, how few nodes can I pack into tree?

Let N_h be min # nodes in AVL height h.

Recurance

$$N_h = N_{h-1} + N_{h-2} + 1$$
$$\geq N_{h-1} + N_{h-2}$$
$$\geq Fib_h = \frac{\varphi^h}{\sqrt{5}}$$

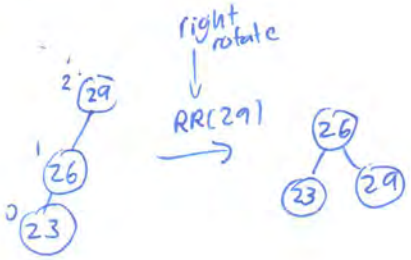
(magic)

or

$$N_h \geq 2\, N_{h-2}$$
$$\geq (2)^{h/2}$$

AVL Insert

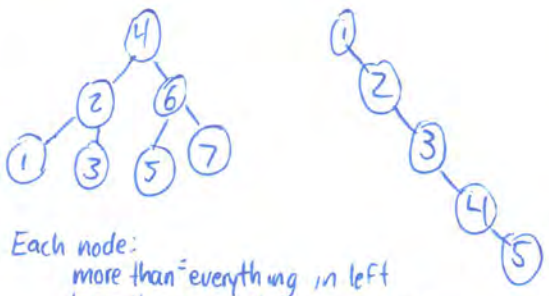
1) BSD insert
2) Fix AVL Property



6.006 Recitation

Fernando Trujano

Binary Search Trees



Each node:
more than = everything in left
less than = everything in right

Find(x): O(h)
insert(elt, x): O(h)
delete(x): O(h)

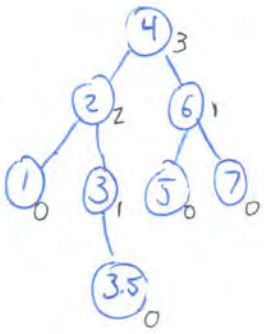
Goal - Minimize h => AVL Tree

Sorted Arrays/List vs Tree:
- Difficult to insert new keys in array

AVL Trees

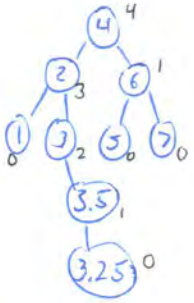
- Height of each child cannot differ by more than 1

$$height(x) = \max(height(left), height(right)) + 1$$
$$height(none) = -1$$

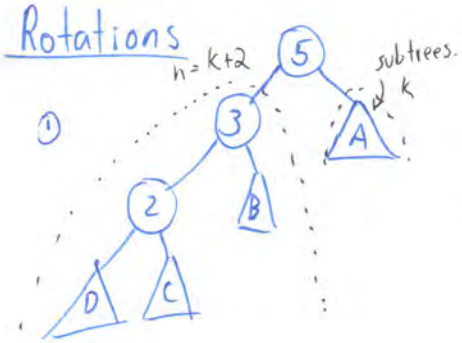


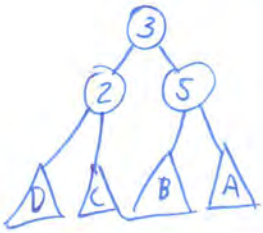
Insert

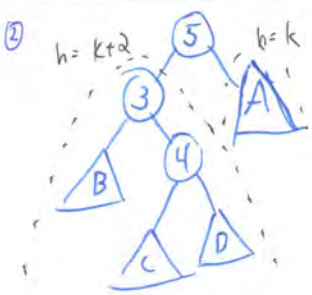
- Suppose I inserted 3.25 => height change of all ancestors
- Do a rotation to fix properties

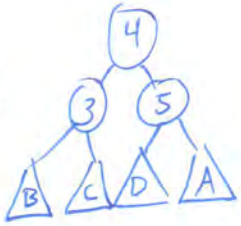


Rotations

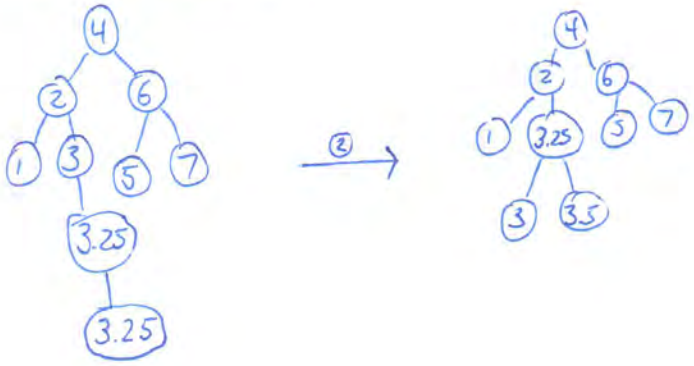








Back to insert example:



Other ideas to minimize n

- The lengths of all root-to-leaf paths differ by ≤ 1 : n
- At most one node has exactly one child: $\frac{n}{2}$

- Parents and children have heights that differ by at most 2 : $\log_\phi n$
  - Like AVL property

- Comparison Model
- Lower Bounds
  - search $\Omega(\lg n)$
  - sort $\Omega(\lg n)$
- O(n) sorting
  - counting sort
  - radix sort

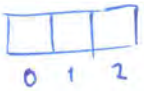
Searching Lower Bound

- given n items (random)
  any algorithm that searches for an item x _ takes $\Omega(\lg n)$ time in worst case.

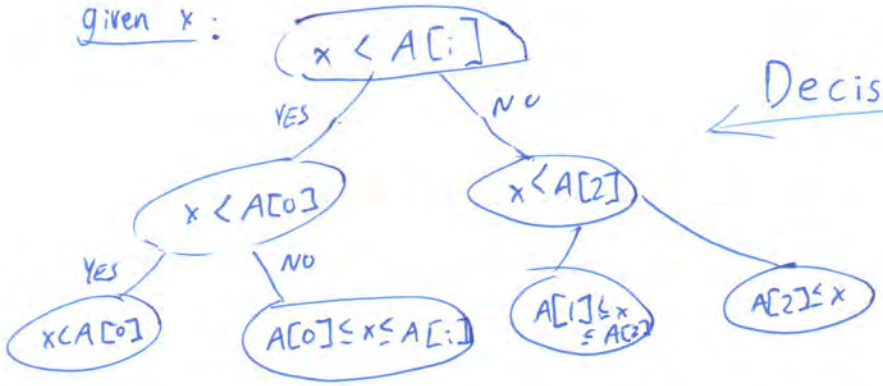
Comparison Model

- input items are black boxes
- only operations are comparisons ($\geq, \leq, >, <, =$)
- time cost = # comparison

eg: binary search n=3



given x:



Decision Tree.

* Any algorith can be written like a tree

decision trees	≡	Algorithm (in comparison model)
internal nodes		decisions / comparisons
leaf		outcome / result
path (root → leaf)		Execution
length of path		running time
height of tree		worst case ↑

Proof (for searching lower Bound)

≥ n possible results

≥ n leaves

height = worst case running time ≥ lg n □

Sorting Lower Bound

- any algorithm that sorts a given n-item list in comparison model takes $\Omega(n \log n)$ time in worst case.

Proof

- n! outcomes

≥ n! leaves

height ≥ $\lg(n!) \approx \Omega(n \log(n))$ because math.

buildheap = O(n)

build BST / AVL = $\Omega(n \lg n)$ / $O(n)$ PROVE!

Integer sorting in linear time

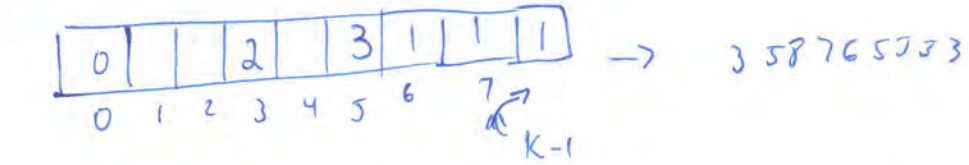
- Assume n integers ∈ {0, 1, ..., k-1} (and each fits in a word)

- can add, mult, index into array...

Thm For k not too large, can sort in $O(n)$ time.
$k \leq n^{O(1)}$

Counting Sort

3 5 8 7 6 5 5 5 3



stable sort:
2 items with the same key will appear in the same order that they appeared in the original.

$$O\left(\sum |L_i| + 1\right) = O(n+k)$$

Radix Sort

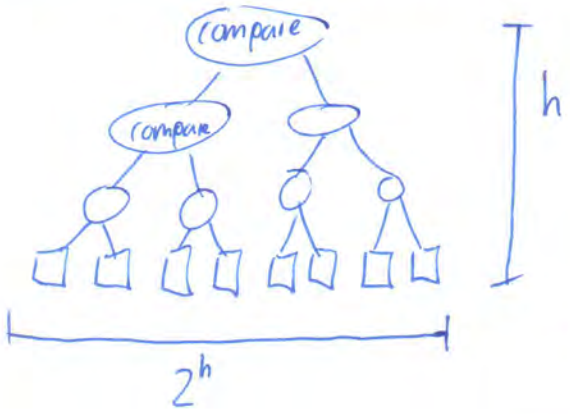
For $k \leq n^c$.
sort in time $O(nc)$

- More in recitation.

9/26/3 6.006 Recitation Fernando Trujano

Search

outcomes: n



runtime: h

#outcomes: n

#distinguishable outcomes: 2^h

$2^h \geq n$

runtime = $h \geq \log(n)$

Sort:

n! permutations of list.

runtime: h

#outcomes: n!

#distinguishable outcomes: 2^h

$2^h \geq n!$

runtime = $h \geq \log(n!) = n \log n$

Counting Sort

Sort with inputs bounded by $\Theta(n)$

Example: range of values 1 ... 2n

1, 5, 4, 10, 9, 18, 10, 2, 3, 7

given: Possible values 1-2(10) => 1-20

1) Write down all of numbers in range
2) Go through given list, add tally every time to appropriate "slot"/number
3) Go through created array and to create sorted list.

Time: $\Theta(n)$

Ex: Given: 1 ... $\frac{n}{2}$ (1,a), (3,b), (2,c) (2,d) (1,e) (3,F)

1)
2) 1 ↓ a ↓ e, 2 ↓ c ↓ d, 3 ↓ b ↓ F

=> 3) (1,a) (1,e) (2,c) (2,d) (3,b) (3,F)

Stable sort: If two elements are tied in initial list, they will remain in the same order on the sorted list. Ex: Counting Sort

Stable sort	Not stable sort
Counting sort	Heap Sort
Merge sort	Bogo Sort ← Brute Force.

Left ties broken by original position. left wins.

Radix Sort

Required range: $O(n^c)$

Ex: $1 \ldots n^3 - 1$

120 37 61 40 448 2 20 54

1) Write #'s in base n and pad with 0's

170 045 075 050 702 002 024 066

2) Counting Sort by Least significant digit

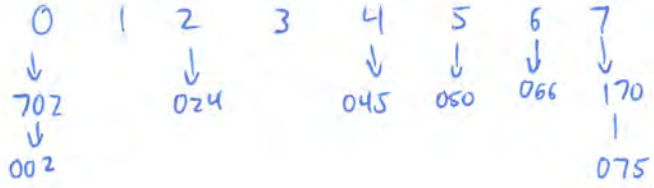
i) Write all numbers in range. (Base 8 so 0-7)

ii)

0	1	2	3	4	5	6	7
↓		↓		↓	↓	↓	
170		702		024	45	066	
↓		↓			↓		
050		002			075		

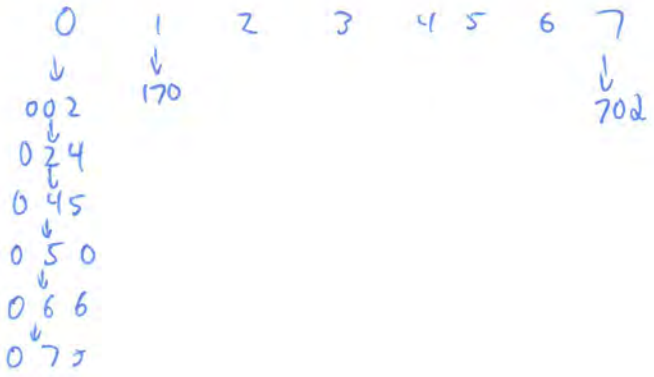
iii) 170 050 702 002 024 045 075 066

3) Counting sort by 2nd Least significant digit



702 002 024 045 050 066 170 075

4) (ounting sort by most significant digit



002 024 045 050 066 075 170 702

5) (onvert back to base (10)

2 20 37 40 54 61 120 446

Time

(counting sorts in linear n time : $\theta(nc) = \theta(n)$ (c constant)

comparisons can take nlogn time for very large numbers. so merge sort typically $\theta(n^2 \log^2 n)$ (c going through each bit)

11 0101 10
01 0101 11

would be $\theta(n^2 \log^2 n)$

9/30/14

6.006 Lecture

Fernando Trujano

Algorithms about numbers

- int Representation 10101

$a, b \to a \cdot b$ $\quad |a| = |b| = k$. Traditional alg $= k^2$

- Exponential revolution

$a, b \to a^b$

a, b, c $\quad |a| = |b| = |c| = k \therefore a^b \bmod c$

Modular Exponentiation

$a^{2^k} = a \to a^2 \to a^4 \to a^8 \to \dots \to a^{2^k}$ (mod c; k)

$a^{10100110\dots0} ??? = a^{10\text{—}0} \cdot a^{1100\dots0} \cdot e$

$a^1 = a$

$a^{10} \quad a^2$

$a^{100} \quad a^4$

$a^{1000} \quad a^8$

$a \quad \vdots$

$a^{1000\dots0} \quad a^{2^k}$

$\approx 1.5k$ modular mults $\quad O(k^3)$

Groups

1) associative
2) identity
3) inverse

Z_n add group mod n

Z_m^* mult group mod n

$Set = \{ 1 \le x \le n : gcd(x, m) = 1 \}$

Z_{15}^* 1 2 4 7 8 11 13 14

≈ relatively prime to 15

$|Z_m^*| = \varphi(n)$ ← Euler's Totient Function

$Z_{p^k}^*$: 1 2 3 p^k

take out : p 2p 3p $p^{k-1}p$ ⟶

P prime $\quad \varphi(p) = p - 1$

$\varphi(p^k) = p^k - p^{k-1}$

$gcd(a,n) = 1$ ← inv(e) mod n

$ax + ny = 1$

$ax = 1 + ny$

$ax = 1 \bmod n$

← Can get x and y with the PULVERIZER

DeF (recall)

cyclic group

$\exists_g \in G$

$g \to g^2 \to g^3 \to$

Z_p^* is cyclic

10/1/14

6.006 Recitation

Fernando Trujano

a, b, c — length of bits required to represent #

$|a| = |b| = |c| = n$

$a < 2^n$
$b < 2^n$
$c < 2^n$

want: $a^b \bmod c$ ← keeps # bounded

simpler problem

$b = 2^k$ for some k

$a^{1000000}$ ← binary
k

$a^2 = a^{10} \bmod c$ ← binary
$(a^{10})^2 = a^{100} \bmod c$
$(a^{100})^2 = a^{1000} \bmod c$
⋮ ⋮
Takes k squarings

Time: $\Theta(kn^2)$

harder problem

$a^b \bmod c$

$b = 11000\ 10\ 11\ 10\ 1\ 1$

		Time
$2^{k_1} =$	$1000000000000\ 0$	$k_1 n^2 \Rightarrow \leq n^3$
$2^{k_2} =$	$10000\ 000\ 0000\ 0$	$k_2 n^2 \Rightarrow \leq n^3$
$2^{k_3} =$	$1000\ 0000\ 0$	
⋮	Could be as many n. (One for each 1)	

$\leq n \Rightarrow \Theta(n \cdot n^3)$
$= \Theta(n^4)$
time

Optimize. Don't recompute everything every time. Build a list of $a^{10}, a^{100} \ldots$ the first time and look back at it after. => Dynamic Programming
$\Rightarrow \Theta(n^3)$

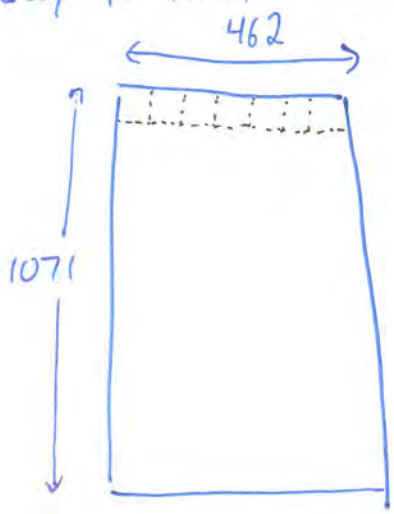
Euclid's Algorithm

$\gcd(a, b) = \gcd(a-b, b)$ ← Core idea
↳ can repeat multiple times

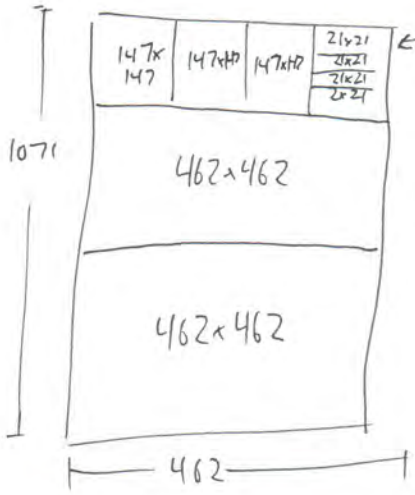
Example

$\gcd(1071, 462) = \gcd(1071 - 2(462), 462) = \gcd(462, 147)$
$= \gcd(462 - 3(147), 147) = \gcd(21, 147) = \gcd(21, 147 - 7(21))$
$= \gcd(21, 0)$
return 21

One way to think about EA



What is the biggest square that can tile this rectangle?



can tile 147x147 and 462x462
=> can tile 1071x462

Extended Euclid's Algorithm

Bezout's equation: a, b s.t gcd = g

$g = sa + tb$

how to find s and t?

work in reverse

Pulverizer!

Runtime of EA

$gcd(a,b) \rightarrow gcd(a \bmod b), b \Rightarrow \Theta(\log(\min(a,b)))$

$gcd(x,y)$ $\frac{x}{y} =$

$\Theta(\log_\phi n) = \Theta(\log n)$

Chinese Remainder Theorem

n_1, n_2 and $\gcd(n_1, n_2) = 1$ ← coprime

$\Rightarrow x \equiv a_1 \pmod{n_1}$

$x \equiv a_2 \pmod{n_2}$

Bezout's equation: $1 = sn_1 + tn_2$ → take mod n_1 of both sides

$1 \pmod{n_1} = tn_2 \pmod{n_1}$

$1 \equiv tn_2 \pmod{n_1}$ Similarly $1 \pmod{n_2} \equiv sn_1$

$x = a_2 sn_1 + a_1 tn_2$

$x \pmod{n_1} = a_1$

---

generalized Chines Remainder Theorem

n_1, n_2, n_3 $\gcd(n_i, n_j) = 1$ → pairwise coprime

$x = a_1 \bmod n_1$

$x = a_2 \bmod a_2$

$x = a_3 \bmod n_3$

$x = a_1 rn_2n_3 + a_2 sn_1n_3 + a_3 tn_1n_2$

$\gcd(n_1, n_2n_3) = 1$

$1 = sn_1 + rn_2n_3$

$1 \pmod{n_1} \equiv rn_2n_3 \pmod{n_1}$

10/2/14 6.006 Lecture Fernando Trujano

Number Theoretic Algorithm

Squares mod p (quadratic residues) mod p

a is sq mod if $\exists x \in \mathbb{Z}_p^*$ s.t $x^2 \bmod p = a$

when p is prime

g^i Then g^{2i} is a sq mod p:

Proof $(g^i)^2 \bmod p = g^{2i}$ ← generator

Q: given p (Primo) and $a \in \mathbb{Z}_p^*$, is "a" a sq mod p?

Binary Search? Does not work because of the mod

Proposal: Find g, Find $\text{index}_g(a)$ $\times$ $g^x = e$

? ? ← impossible?

Euler: Q $a^{\frac{p-1}{2}} \bmod p = \begin{cases} +1 & \text{sq} \\ -1 & \text{not sq} \end{cases}$

Proof: Assume a is a sq $a = b^2$ $a = g^{2i}$

$$(g^{2i})^{\frac{p-1}{2}} = g^{i(p-1)} = (g^{p-1})^i = 1^i = 1 \checkmark$$

other case

$$(g^{2i+1})^{\frac{p-1}{2}} = \underbrace{(g^{2i})^{\frac{p-1}{2}}}_{1} \cdot g^{\frac{p-1}{2}} = g^{\frac{p-1}{2}} = -1$$

$g^{2i} \cdot g$

$\mathbb{Z}_p^*$ cycle $p-1$ all mod 11 $\mathbb{Z}_{11}^*$

$2^1 = 2$ $2^2 = 4$ $2^3 = 8$ $2^4 = 5$ $2^5 = 10$ $2^6 = 9$ $2^7 = 7$ $2^8 = 3$ $2^9 = 6$ $2^{10} = 1$

Q: given p (Prime) a (sq mod), find $\sqrt{a} \bmod p$

A: for half of p's $= i$ a for $p = 3 \bmod 4$

$p = 3 + 4k$

$a = x^2 \ldots$

$a = g^{2i}$

$a^{\frac{p-1}{2}} = 1$

$a^{1+2k} = 1$

$(a^{1+k})^2 = a^{2+2k} = a$

Data Structures

Max Heap: All children ≤ parents

Binary Search Tree: 

AVL Trees: Balanced BST. $h_{left} - h_{right} \leq 1$
Augmenting each node with size.

Numerics

n has $\log n$ digits
Efficient = easy = polynomial time

Euclidean Algorithm: $GCD(n, m) = GCD(m, n \bmod m)$

Extended Euclidean Algorithm = Pulverizer.
- used to Find inverses fast

$\mathbb{Z}_n$ addition mod n

$\mathbb{Z}_n^*$ multiplication. Set of # relatively prime to n

Chinese Remainder Theorem

$x \in \mathbb{Z}_n \qquad n = p_1 p_2 \cdots p_k$ (Relatively prime)

CRT

$X \leftrightarrow (x_1, x_2, x_k)$ where $x_i \equiv x_i \bmod(p_i)$

useful for square roots

3/3

- Factorization: Probably hard!

is 3 a square mod 4931?

x^2

Find int x: $x^2 = 3 \pmod{4931}$

$$3^{\frac{p-1}{2}} = \begin{cases} +1 & \text{square} \\ -1 & \text{not square} \end{cases}$$

$$3^{2465} \bmod 4931$$

$3^2 = 9$	$3^{32} = 2804$	$3^{512} = 1680$
$3^4 = 81$	$3^{64} = 2402$	$3^{1024} = 1868$
$3^8 = 1630$	$3^{128} = 334$	$3^{2048} = 3207$
$3^{16} = 4022$	$3^{256} = 3074$	

* All mod 4931

$2465 =_{\text{binary}} 100110100001$

(2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1)

Multiply wherever there are ones

$1 = 3207 \cdot 3074 \cdot 334 \cdot 2804 \cdot 3 \bmod 4931$

$3^{2465} \equiv 1 \bmod 4931$

HOW THE FUCK DO I CALCULATE 1

$3^{\frac{p-1}{2}} = 1$

$\Rightarrow$ 3 is a sq.

Find roots of 3?

$3^{\frac{4931-1}{2}} \equiv 1 \bmod 4931$

$3^{\frac{4(1232)+3-1}{2}} \equiv 1 \bmod 4931$

$3^{2(1232)+1} \equiv 1 \bmod 4931$

$3^{2(1232)+2} \equiv 3 \bmod 4931$

$\left(3^{1233}\right)^2 \equiv 3$

3^{1233}: square root!

Looks a lot like 2465...

$1233 \equiv_{\text{binary}} 10011010001$

$1868 \cdot 334 \cdot 2402 \cdot 4022 \cdot 3 = 1291$

$\Rightarrow 1291^2 \equiv 3 \bmod 4931$

$4931 - 1291 = 3640^2 \equiv 3 \bmod 4931$

Runtime $n = |p|$, $n = \Theta(\log p)$

1) Is 3 a square $\quad 3^{\frac{p-1}{2}} \quad \Rightarrow \Theta(n)$ mults $\Rightarrow \Theta(n^3)$ (each mult $\Theta(n^2)$)

2) Sqrt of 3 $\quad 3^{\left(\frac{p-1}{2}+1\right)/2} \quad \Rightarrow \Theta(n^3)$

$= \Theta(n^3)$

Exam Prep:

Which operations supported efficiently by min heaps?

= logarithmic time or better

- ✗ Find(x)
- ✓ Findmin()
- ✓ Insert(x, elt)
- ✗ Delete(x) ← you have to find it first
- ✓ Extract-min()

Where can I find the max?

Any of the leaves => last 8 elements.

6.006 Lecture

Fernando Trujano

Review Session 1

T/F : Negative points if wrong

Algorithms: English or pseudocode

Asymptotics

$O \quad \Omega \quad \Theta$

$f(n) = O(g(n))$
$\Rightarrow f(n) \leq c \cdot g(n)$
$\Rightarrow g(n) = \Omega(f(n))$

$f(n) = \Theta(g(n))$
$\Rightarrow f(n) = O(g(n))$
$f(n) = \Omega(g(n))$

$\binom{n}{k}$ "n choose k" Always $\leq 2^n$

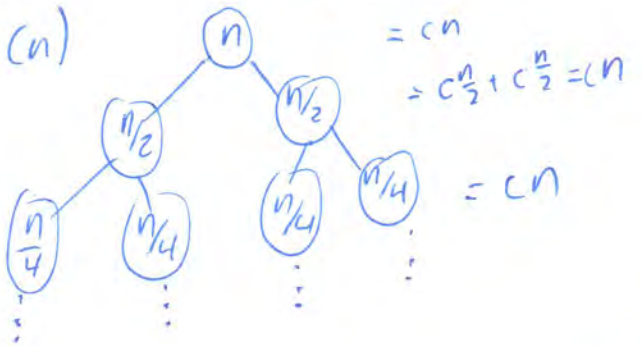
$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k$$

Recurrance

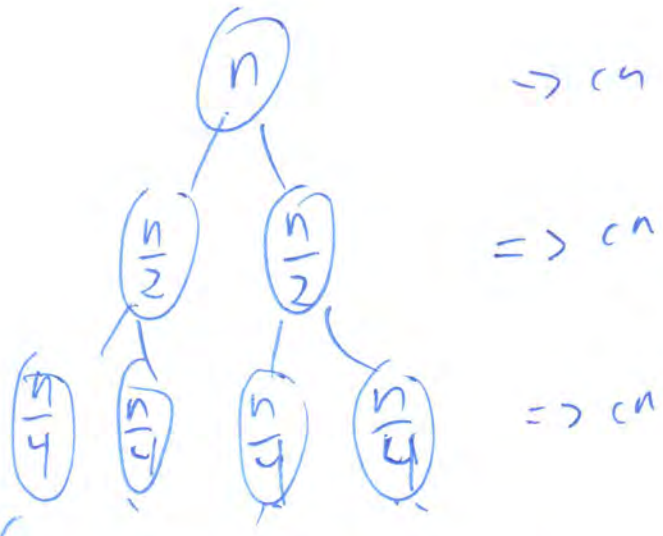
$T(n) = aT(n/b) + f(n)$ ← work

Ex: $2T(n/2) + O(n)$

Tree Method



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

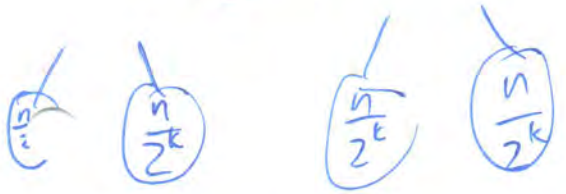


=> cn

=> cn

=> cn

Keep going until work is 1



=> cn

that is when

$$2^k = n$$

$$\Rightarrow \log_2 n = k$$

cn work per for $\lg n$ levels

$$\Rightarrow \theta(cn \log n) \Rightarrow \theta(n \log n)$$

Master Theorem : On cheat sheet

- use change of variables if not in form $T(n) = aT(n/b) + f(n)$

Asymptotic Relationships

Small

$1/n$
$1/5$
$\log(\log(n))$
$\log(n)$
$n^{1/100}$
$10^{100}n$
$n\log n, \log(n!)$
$n^{100}, \binom{n}{100}$
$3^{\sqrt{n}}$
$2^n, 2^{n+1}$
$n2^n$
3^n
$4^n, 2^{2n}$
$n!$

Big

Explanations for the most common errors:

- $\log n = o(n^x)\ \forall x > 0$. Using L'Hôpital's Rule:

$$\lim_{n\to\infty} \frac{n^x}{\log n} = \lim_{n\to\infty} \frac{\frac{d}{dn}n^x}{\frac{d}{dn}\log n} = \lim_{n\to\infty} \frac{xn^{x-1}}{1/n} = \lim_{n\to\infty} xn^x - \infty$$

$\log(n^2) = 2\log(n)$

- $2^n = o(3^n)$:

$$\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$$

$\log(2^n) \Rightarrow \Theta(n)$

- By Stirling's approximation, $n! = \Theta(n^n)$, so $\log n! = \Theta(\log n^n) = \Theta(n \log n)$.
- $\binom{n}{100}$, or "n choose 100", can be expressed as

$\log_2 n = k$
$\Rightarrow 2^k = n$

$$\binom{n}{100} = \frac{n!}{100! \cdot (n-100)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-99)}{100!}$$

Now, observe $(n-100)^{100} \le n \cdot (n-1) \cdot \ldots \cdot (n-99) \le n^{100}$. Because $(n-100)^{100} = \Theta(n^{100})$, it follows that $n \cdot (n-1) \cdot \ldots \cdot (n-99) = \Theta(n^{100})$ and therefore $\binom{n}{100} = \Theta(n^{100})$.

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{ne}{k}\right)^k$$

$f(n) = [g(n)]$

O, Θ, Ω or $\asymp$	$f(n)$	$g(n)$
O	n^2	n^3
Ω	$n \lg n$	n
Θ	1	$2 + \sin n$
Ω	3^n	2^n
Θ	4^{n+1}	2^{2n+2}
O	$n \lg n$	$n^{1.0001n}$
Θ	$\lg\sqrt{10n}$	$\lg n^4$
O	$n!$	$(n+1)!$

Master Theorem

The master method provides a simple method for solving recurrences of the form

$T(n) = a(n/b) + f(n)$.

a= # of subproblems
(n/b) = size of each subproblem
f(n) = how long it takes to divide into subproblems and combine the results

where $a \ge 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The master method can be broken down into three cases depending on how the function $f(n)$ compares with the function $n^{\log_b a}$. The three cases can also be interpreted as different weightings of the recursion tree associated with the recursion. ① Calculate $n^{\log_b a}$ ② Compare with $f(n)$

1. **Case 1:** If $f(n) = \Theta(n^{\log_b a - \varepsilon})$, then the amount of work per level geometrically increases as we go down the tree. The work at the leaf level dominates and $T(n) = \Theta(n^{\log_b a})$.

2. **Case 2:** If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then the amount of work per level have about the same cost (i.e. work per level does not *polynomially* increase or decrease, though work per level still may increase or decrease at some slower rate). We have to take the work of all levels into account and $T(n) = \Theta(n^{\log_b a} \log^k n \log n) = \Theta(n^{\log_b a} \log^{k+1} n)$. $T(n) = \Theta(f(n)\log n)$

3. **Case 3:** If $f(n) = \Theta(n^{\log_b a + \varepsilon})$, then the amount of work per level geometrically decreases as we go down the tree. The work at the root level dominates and $T(n) = \Theta(f(n))$. Note: the recursion must additionaly satisfy the "regularity" condition: $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n.

To use the master method, simply determine which case (if any) of the master theorem applies, by comparing $f(n)$ with $n^{\log_b a}$. If the function $n^{\log_b a}$ is the larger, consider case 1. If $f(n)$ and $n^{\log_b a}$ are the same, consider case 2. And if $f(n)$ is the larger, consider case 3.

Change of Variables (Solving Recurrences)

For some particularly tricky recurrences, it may be necessary to combine the existing methods we've looked at with a method known as **change of variables**. The idea behind change of variables is to rewrite a tricky recurrence in terms of a new variable that is somehow related to the old variable, solve the new recurrence, then change back to the original variable.

Consider, for example, the following recurrence containing a square root:

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

None of our existing methods provide an easy way to solve this recurrence. To solve this we can perform a change of variables by defining a new variable m as $n = 2^m$. Substituting this in gives:

$$\begin{aligned} T(2^m) &= 2T(\sqrt{2^m}) + \Theta(\log 2^m) \\ &= 2T(2^{(m/2)}) + \Theta(m) \end{aligned}$$

This recursion still isn't in the standard form that we expect for Master Theorem, so we define a new recurrence, S as $S(m) = T(2^m) = T(n)$. This gives us:

$$\begin{aligned} S(m) = T(2^m) &= 2T(2^{(m/2)}) + \Theta(m) \\ &= 2S(m/2) + \Theta(m) \end{aligned} \qquad \begin{matrix} a=2 \\ b=2 \end{matrix}$$

Suddenly, we have a form we can work with using any of the previous methods we've learned. Using Master Theorem, for example, gives us $S(m) = m \log m$. Now we simply have to substitute back to make this meaningful in the context of T and n:

$$\begin{aligned} S(m) &= \Theta(m \log m) \\ T(n) &= \Theta(\log n \log\log n) \end{aligned}$$

Exercise 3 – Suppose you are given $T(\log_2 n) = 2T(\log_4 n) + \Theta(\log\log n)$). Compute a closed form for $T(x)$ using change of variables.

Notation

Recurrences: Merge Sort: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
Insertion Sort: $T(n) = T(n-1) + O(n) = O(n^2)$ 2D Peak finding: $T(n) = T(n/2) + \Theta(n) = \Theta(n)$

Data structure to find how many ints in range (a,b).
in range in $4k$
A: same preprocessing as counting sort. $c_i = C'(b) - C'(A)$

Sq mod p

Is 3 a square mod 4931?

Find int x: $x^2 \equiv 3 \pmod{4931}$

$3^{\frac{p-1}{2}} = \begin{cases} +1 & \text{a square} \\ -1 & \text{not a square} \end{cases}$

$3^{2465} \bmod 4931$

$3^4 = 9$ $3^{32} = 2581$ $3^{256} = 1620$
$3^8 = 81$ $3^{64} = 2402$ $3^{512} = 1363$
$3^{16} = 1135$ $3^{128} = 334$ $3^{1024} = 3207$
$3^{32} = 4022$ $3^{256} = 3074$ *All mod 4931

Multiply powers (have one prop)

$2465 = 100110100001$

$\equiv 3207 \cdot 3074 \cdot 334 \cdot 2581 \pmod{4931}$

$3^{2464} \equiv 1 \pmod{4931}$

$3^{\frac{2464}{2}} = 1$

$\Rightarrow 3$ is a sq.

Find roots of 3?

$3^{\frac{4930}{2}} \equiv 1 \pmod{4931}$

$3^{\frac{2(1233)+2}{2}} \equiv 1 \pmod{4931}$

$3^{2(1233)+1} \equiv 1 \pmod{4931}$

$3^{2(1233)+2} \equiv 3 \pmod{4931}$

$\left(3^{\frac{2468}{4}}\right)^2 \equiv 3$

Square root!

$1233 = 10011010001$

$1368 \cdot 324 \cdot 2402 \cdot 4022 \cdot 3 = 1291$

$\Rightarrow 1291^2 \equiv 3 \pmod{4931}$

$4931 - 1291 = 3640$, $3640^2 \equiv 3 \pmod{4931}$

roots = 1291, 3640

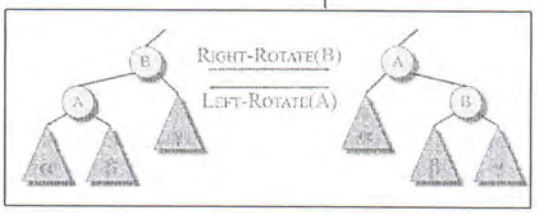
√ mod p

$a^{\frac{p-1}{2}} \equiv 1 \mod p$
$a^{\frac{3+4k-1}{2}} \equiv 1 \mod p$
$a^{1+2k} \equiv 1 \mod p$
$a^{2+2k} \equiv a \mod p$
$(a^{1+k})^2 \equiv a \mod p$
$a^{1+k} \equiv \sqrt{a} \mod p$

Verify BST

```
def verify(node):
    return helper(node, -∞, ∞)
def helper(node, lo, hi):
    if not(node.left is none or helper(node.left, lo, node.key)
        return False
    if not(node.right is None or helper(node.right, node.key, hi)
        return False
    if (node.key < lo or node.key > hi):
        return False
    else: return True.
```

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



GCD + Pulverizer

$$\begin{aligned} \gcd(259,70) &= \gcd(70,49) && \text{since } 259 \text{ rem } 70 = 49 \\ &= \gcd(49,21) && \text{since } 70 \text{ rem } 49 = 21 \\ &= \gcd(21,7) && \text{since } 49 \text{ rem } 21 = 7 \\ &= \gcd(7,0) && \text{since } 21 \text{ rem } 7 = 0 \\ &= 7. \end{aligned}$$

x	y	$(x \text{ rem } y)$	$=$	$x - q \cdot y$
259	70	49	$=$	$259 - 3 \cdot 70$
70	49	21	$=$	$70 - 1 \cdot 49$
			$=$	$70 - 1 \cdot (259 - 3 \cdot 70)$
			$=$	$-1 \cdot 259 + 4 \cdot 70$
49	21	7	$=$	$49 - 2 \cdot 21$
			$=$	$(259 - 3 \cdot 70) - 2 \cdot (-1 \cdot 259 + 4 \cdot 70)$
			$=$	$\boxed{3 \cdot 259 - 11 \cdot 70}$
21	7	0		

	Time complexity								Space complexity
	Average				Worst				Worst
	Find Min	Search	Insert	Delete	Find Min	Search	Insert	Delete	
Array	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
(Min/max) Heap	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$
BST tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

There are two kinds of binary heaps: max-heaps and min-heaps. Max-heaps have the property that $A[\texttt{Parent(i)}] \geq A[i]$, and min-heaps have the property that $A[\texttt{Parent(i)}] \leq A[i]$.

Max-heaps have the following operations: Heaps

`Max-heapify(A, i)`
Description: Assuming that binary trees rooted at the left and right children of i are max-heaps (but the tree rooted at i may not be), return A such that the binary tree rooted at i is a max-heap.
Approach: Compare the value $A[i]$ with its left and right children. If it is smaller than either of its children, exchange it with the larger of its two children. Continue comparing the value $A[i]$ with its left and right children and making exchanges with the larger of its children, until the value $A[i]$ is greater or equal to both of its children.
Analysis: Since $A[i]$ may need to be checked with each value in some branch of the tree, the runtime is $O(h) = O(lgn)$.

Heaps: parent(i) = $\lfloor i/2 \rfloor$, left(i) = $\lfloor 2i \rfloor$, Right(i) = $\lfloor 2i+1 \rfloor$

`Build-max-heap(A)`
Description: Given an array $A[1..n]$, build a max-heap.
Approach: Starting from the parent of the last leaf and traversing backwards in the array (traversing left and up in the binary heap), call `Max-heapify` on each index of the array.
Analysis: Using the recurrence $T(n) = 2T(n/2) + lgn$ to model the operation, the runtime is $O(n)$.

`Heapsort(A)`
Description: Given an unordered array $A[1..n]$, sort A using a max-heap.
Approach: Build a max-heap from A. Remove the maximum element by swapping it with the last leaf and placing it at the end of our sorted array. Calling `Max-heapify` on the root will result in a max-heap with the second largest element at the root. Continue the process of exchanging the root with the last leaf, removing the current maximum,placing it in our sorted array, and calling `Max-heapify` on the root.
Analysis: We make $n-1$ calls to `Max-heapify` (one call after each element in A that is visited), which has running time $O(lgn)$, so the total runtime of `Heapsort` is $O(n \lg n)$.

`Heap-maximum(A)`
Description: Return just the value of the maximum element of the max-heap A.
Approach: The maximum element is the root of the max-heap or $A[1]$.
Analysis: $O(1)$.

`Heap-extract-max(A)`
Description: Return the value of the maximum element of the max-heap and also remove the element from the max-heap.
Approach: The maximum element is the root of the max-heap. To remove it, we exchange it with the last leaf, remove the element, and `Max-heapify` the new root.

BST Operations

Binary Search Trees

Binary search trees have the following operations that allows users to search for elements and manipulate the tree.

`search(k)`
Description: Find key k. Return the node that contains k if it exists or NIL if it is not in the tree.
Approach: Starting at the root, check to see if the node we're at contains key k. If so, return that node. If not, traverse to the left child if k is smaller or traverse to the right child if k is larger and repeat. If we reach NIL before finding k, then k is not in the tree and we return NIL.
Analysis: Searching could potentially go along the longest branch of the tree, so the runtime of `search` is $O(h)$ where h is the height of the tree.

`insert(k)`
Description: Insert key k into the tree.
Approach: Starting at the root, compare the key of the node we're at to k. Traverse to the left child if k is smaller or traverse to the right child otherwise and repeat until we reach NIL. Once we reach NIL, replace the NIL node with a node containing k, thus inserting k into the tree.
Analysis: Searching where to insert k could potentially go along the longest branch of the tree, so the runtime of `insert` is $O(h)$ where h is the height of the tree.

`find-min(x)`
Description: Find the node with the minimum key of the tree rooted at node x and return it.
Approach: Starting at x, keep traversing to the left child until we reach a node with no left child. This node contains the minimum key, so we return it.
Analysis: The left-most branch could potentially be the longest branch of the tree, so the runtime of `find-min` is $O(h)$ where h is the height of the tree.

`next-larger(x)`
Description: Find the node that contains the next larger key relative to the key at node x.
Approach: If x has a right child, return find-min(x.right). Otherwise, traverse upwards through the tree in x's ancestry line until we reach a node with a larger key than x's key and return this node.
Analysis: We could potentially go through the longest branch of the tree up x's ancestry line or down x's right subtree so the runtime of `find-min` is $O(h)$ where h is the height of the tree.

`delete(x)`
Description: Remove node x from the tree while maintaining the tree's properties
Approach: If x has no children, just remove it by replacing x with NIL. If x has one child, splice out x by linking x's parent to x's child. If x has two children, splice out x's successor and replace x with x's successor.

Finding the next larger element: successor(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);
```

Radix Sort

- imagine each integer in base b
- $\Rightarrow d = \log_b k$ digits $\in \{0, 1, ..., b-1\}$
- sort (all n items) by least significant digit $\rightarrow$ can extract in $O(1)$ time.
- ...
- sort by most significant digit $\rightarrow$ can extract in $O(1)$ time.
- use counting sort for digit sort.
  - $\Theta(n+b)$ per digit
  - $\Theta((n+b)d) = \Theta((n+b)\log_b k)$ total time
  - minimized when $b = n$
  - $\Theta(n \log_n k)$
  - $O(nc)$ if $k \leq n^c$

Numerics: $\mathbb{Z}_n$: addition mod n $\phi(n) = |\mathbb{Z}_n^*|$
$\mathbb{Z}_n^*$: Multiplication. Set of # relatively prime to n

Chinese Remainder Theorem: Relatively prime
If $x \in \mathbb{Z}_n$ $n = p_1 p_2 \cdots p_k$
Then $\Rightarrow$ $x \leftrightarrow (x_1, x_2 \ldots x_k)$ where $x = x_i \pmod{p_i}$

Multiplication $O(k^2)$. 'shift and add' (k = # digits) } Polynomial in size of input
Division $O(k^2 \lg k)$ 'Binary search'

$\binom{n}{k}$ = c^n

Have you seen my integer?
Input: Sorted array N-1 distinct int in range $\{1 \ldots N\}$. So exactly one int $a \in \{1 \ldots N\}$ is not in A.
Goal: Find a? in $O(\log n)$
Solution: Binary search for missing value as follows: guess index i to be the location of missing num. If A[i+1]-A[i-1] = 2 return i. Otherwise, if A[i] = i, missing num must be > i. If A[i] = i+1 missing num < i. Perform recursion on appropriate half of array.

10/14/14 6.006 Lecture #11 Fernando Trujano

Hashing I

- Dictionaries
- Prehashing and hashing
- How to resolve collisions - chaining
  Analysis: Simple Hashing Assumption
- Good hash Functions

Dictionaries (ADT)

keys are unique ← convention

Set of items = (key, value)

insert(item)
delete(item)
search(key)
} → We want in O(1) time (probabilistically)

Balanced BST (ex AVL)
- O(log n)
+ supports succ, predecessor

Dictionaries
+ O(1)
- no succ, pred
  only ins, del, search

Python

```
D[key] = value  // insert
del D[key]      // delete
D[key]          // search
```

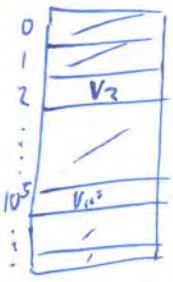
Applications

- phonebook
- doclist
- network routers
- substring search
- File and directory
- cryptography

Simple idea

Direct access table

item = (k, v) Ex $(2, v_2)$ $(10^5, v_{10^3})$



Problem

① Keys may not be integers
② Horrible waste of space

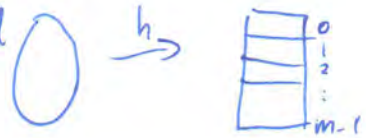
Solutions

①: Map keys to integers. Prehashing

- In theory, everything is a string of bits ☐
  injectivity: prehash(x) ≠ prehash(y) <=> x ≠ y
- In python: hash(object) → integer

② Hash Functions

Goal: Reduce all keys (universe of keys) down to a reasonable size.

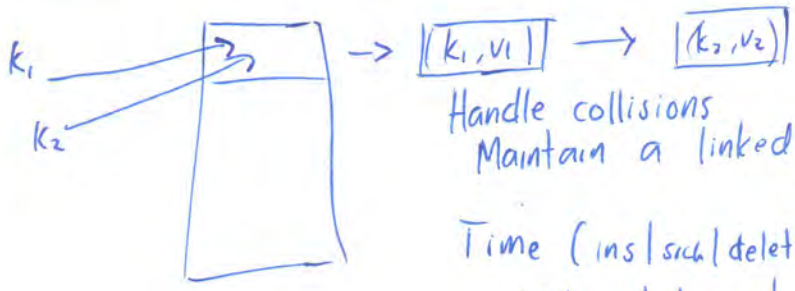


$h: \mathcal{U} \to \{0, 1, 2 \dots m-1\}$

Problems with goal

① Collisions
② What if tons of items map into same location

Hashing by Chaining



Handle collisions
Maintain a linked list at each element.

Time (ins | srch | delete)
$\theta(1) + |chain|$

Goal: Control $|chain|$

Good hash functions

① $h(k) = 0$
② $h(k)$ = random # X
③ $h(k) = k \bmod m$ // division Method
$(1, v_1), (2, v_2) \ldots (n, v_n)$: $|chain| = \lceil n/m \rceil$
$(1, v_1), (m+1, v_2) \ldots$: $|chain| = n$ ← bad X

Universal Hashing

$$h(k) = [ak + b \bmod p] \bmod m$$

(p: some prime > 0)

a, b: random #'s $0 \ldots p-1$

worst case
for any $k_1 \neq k_2$ (collision)
Probability $[h(k_1) = h(k_2)] = 1/m$

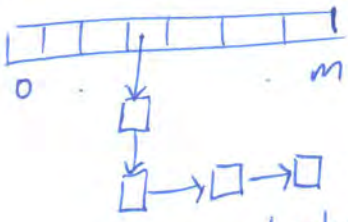
10/15/14 6.006 Recitation Fernando Trujano

Hashing

Hash tables

$h(i) \Rightarrow 0, m$

Put i in position $h(i)$



Linked lists if collisions

Runtimes

Insert: $\Theta(1)$ (amortized / average) expected time, $\Theta(n)$ worst case

Delete: " " " " " "

Find: " " " " " "

Family H of hash Functions

Idea: choose randomly from H to minimize collisions

Universal Family of hash functions

$$\forall x, y \quad x \neq y \quad \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}$$

Probability

H: for some a, b (Random), p (large prime)

$$h(x) = ((ax + b \bmod p) \bmod m)$$

$h(x) = h(y)$

$ax + b \equiv ay + b + im \pmod p$

$a(x - y) \equiv im \pmod p$

$a \equiv im(x-y)^{-1} \pmod p$

$a \equiv im\,(x-y)^{-1} \pmod{p}$

$p-1$ ↑ $\lfloor p/m \rfloor$ So collision prob $\approx \frac{\lfloor p/m \rfloor}{p-1} \to 1/m$ for large p

---

$\Pr[h(x) = h(y) = h(z)] \leq \frac{1}{m}$

$m = \Theta(n)$

$d = \{\}$ prob $\frac{1}{m}$: $\Theta(n)$
prob : $\Theta(1)$

Expected Runtime: $\Theta(1) + \frac{\Theta(n)}{m}$
$= \text{constant}$

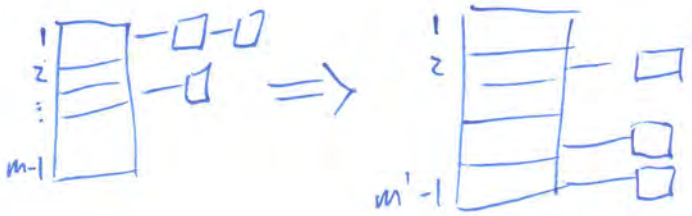
Hashing II

n items, table size m

Table resizing

Invariant: $m \geq n$

if $n > m$ at any point, grow table to size m'



1) Choose new hash for

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

2) For every item in T
insert to T'

$$RT = \Theta(n+m+m') \text{ time}$$

what is m'?

$m' = m+1$?

Imagine inserting n elements into the empty table

Cost: $\Theta(1+2+3+\ldots n) = \Theta(n^2)$ ✗

$m' = 2m$?

inserting n elements into empty table

Cost: $\Theta(1+2+4+8+16+\ldots n) = \Theta(n)$ ✓

Cost? may be fast
We don't know - may have to pay a lot to resize table

Solution: Don't worry about individual insertions.
Analyze the average insertion

Amortization

- operation takes "$T(n)$ time" Amotized if k ops take $\leq k \cdot T(n)$ time

So what happens with deletions?

- See recitation

String Matching

Given strings s and t, does s occur as a substring of t?

Ex: s = "your name"
t = INBOX

Obvious Algorithm: Check each substring

$t[i : i + |s| - 1] = s$

if match: YES

Running Time

$$RT = \Theta((|t| - |s|) \cdot |s|)$$

($|t| - |s|$: # iterations; $|s|$: cost of each iteration)

$$= \Theta(|t| \cdot |s|$$

Happier would be $\Theta(|t| + |s|)$ ← would have to pay at least this much.

Can do this with randomness!!!

Instead of comparing s to each substring of t compare the hashes!

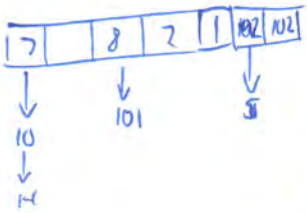
t = abcde

So we need a hash function that $h(abcd) \rightarrow h(bcde)$ in constant time!

next class

Table Resizing

Needed because table might become full and linked list will be longer.



more elements than slots.

I'll choose to resize now

to a size 14 table.

↳ 2m

← will need a new hash function

Recall Hash Family: $H : ((ax+b) \bmod p) \bmod m)$

↑ size of hash table

Now reinsert elements in new hash table

Yay! Now what if the user deletes a bunch of elements?

Hash Table now:

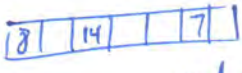
Unused Space!

Resize HashTable!

I'll chose to resize

↳ Ideally I should resize when?

and reinsert



and repeat...

Hash Table n slots, $\frac{n}{2}$ elts

$\frac{n}{2}+1$ inserts

K = insert time

cost: $K(n+1)$

Table resize

$2n$ slots

$n+1$ elts

Avg/insert: $\approx \frac{Kn}{n/2} = 2K = \Theta(1)$

$d/4$ deletes (chosen)

Table resize

$n/2$ slots

$n/4$ elements

cost: $K\frac{n}{4}$

Avg/delete $\approx K\frac{n}{4}/\frac{n}{4} = K = \Theta(1)$

Hash tables support all operations in
$\Theta(1)$ amortized expected time

Best World	Middle World	Worst World

Simple Uniform Hashing Assumption (SUHA)

Each hashed key has an equal chance of hashing to any given bin independent of where all the previous keys hashed.

Find Substring

MISSISSIPPI ← Large input string, n, Naive approach $O(mn)$

Look For "SIP", m

$\downarrow$

4

New Hash function:

M=0 I=1 S=2 P=3

Ex "SIP" = $213_4 = 39$ (base 4)

$39 \bmod 5 = 4$ ← chosen

Start at "MIS"

$\downarrow$

$012_4 = 6 \equiv 1 \pmod 5$ ← Not a match

can now compute h("ISS") using h("MIS") w/o recomputing everything

MIS → 1
IS → 1
IS → 4 - times 4 (since base 4 - shift everything)
ISS → 1

Repeat:

Pop → ISS → 1 ← Check if its = h("SIP")

SS → 0

SS → 0 2×4

SSI → 1 ← Check

Repeat

SSI → 1

SI → 4

SI → 16 mod 5 = 1

SIS → 3 ← Check

Now we get O(n+m)

10/21/14 6.006 Lecture #14 Fernando Trujano

Hashing III

- Rolling Hash
- Open Addressing
  (how to handle collisions part 2)

Rolling Hash Function

- append(s,c): appends char c to end of s
- deleteFirst(s,c): deletes first char of s if it's c

Ex: $h(s) = h(s_{k-1}, s_{k-2}, \ldots s_0)$

$= h(s_{k-1}\alpha^{k-1} + s_{k-2}\alpha^{k-2} + \ldots s_0)$ α = base (eg 27) English with space

$= (s_{k-1}\alpha^{k-1} + \ldots + s_0) \bmod m$

Compute $h(\text{append}(s,c), h)$: $h \cdot \alpha + c \bmod m$
(h = $h(s)$)

$h(\text{deleteFirst}(s,c), h)$: $h - c \cdot \alpha^{k-1} \bmod m$

Don't have to recompute entire hash everytime.
↳ very useful for string matching (see recitation)
Karp Robin

$$O(|s| + (|t| - |s|) \cdot (O(1) + |s| \cdot \tfrac{1}{|s|}) + O(|s|)$$

- $|s|$: hashs first
- $(|t| - |s|)$: # iterations
- $O(1)$: updating hash
- $|s| \cdot \frac{1}{|s|}$: grab false match
- $O(|s|)$: ↳ compare strings if yes

$= O(|t| + |s|)$ in expectation

Handle collisions in hash table:
can use chain (linked list)
or Open Addressing.

Open Addressing

no chaining, pointers, linked list ← Waste of space

Probing Hash Function

Hash function with two arguments.

key ↙ ↘ $n \in \{0, 1 \ldots m-1\}$

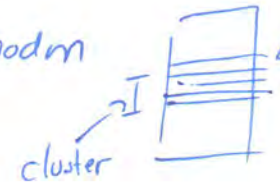
specifies an order of the slots to probe (ins/search) for key

$h(k, 0)$ ↳ slot full $h(k, 1)$ ← should return a different hash!

Linear Probing: $h(k, i) = h'(k) + i \mod m$

Problem: "Rich gets richer"

Clustering



Double Hashing: $h(k, i) = h_1(k) + i \cdot h_2(k) \mod m$

10/22/14

6.006 Recitation

Fernando Trujano

Recall substring finding algorithm.

average case = average runtime on worst input

worst case = worst runtime on worst input

deterministic algorithm: average case = worst case

Last recitation's algorithm was deterministic.

Upgrade Idea:

Pick at random from a universal family of rolling hash function

easy to pop and add a character w/o recalculating entire hash.

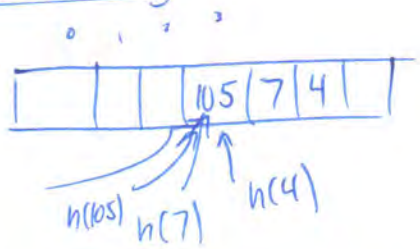
$h(x) = ((ax+b) \bmod p) \bmod m$

not a rolling hash function.

$x' = ax - u + v$ where x' relates to x

$h(x') = ((ax - au + av + b) \bmod p) \bmod m$

Open Addressing

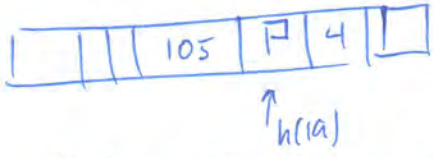


h(7)

so Find(7) start at bucket 3 and walk the cluster until Find 7.

Find(1) hash (1) - walk until find empty space

delete(7): hash(7) Find(7) delete it and replace it with a Flag ← Tells other methods to keep going.



aka space is not empty

insert(19): hash 19. Can replace Flag.

6.006 Notes

Fernando Trujano

Hashing

Universal Hashing

$$h(k) = [(ak + b) \bmod p] \bmod m$$

- p: large prime
- a, b: Randomly chosen at setup time, $0 \le a, b \le p$, $a \neq 0$
- m: size of the table

$$\text{Prob}_{a,b}\left(\underbrace{h(k_1) = h(k_2)}_{\text{hash collision}}\right) = 1/m \qquad k_1 \neq k_2$$

Expected Search Time

$= O(1 + \#\text{ keys that collide w/ } k)$

$= O\left(1 + (n-1)/m\right)$

($n-1$: All keys but k; $1/m$: chance of collision)

$\alpha = n/m$ = load factor

$= O(1 + \alpha)$

$= O(1)$ if $m = \Theta(n)$

(m: size of table)

need to ensure that $m = \Theta(n)$: Table Doubling

- Create a new table when current one too full
- Rehash all elements into new table
- Still $O(1)$ amortized time since we rarely have to do this

Rolling Hash

- Calculate a new hash based on previous hash w/o recomputing everything
- Remember DNA Pset problem. (Sub-string matching)

Open Addressing

For collision resolution

- No linked lists (→ no chaining) instead, several hashfunctions $h_1(k), h_2(k) \ldots h_{m-1}(k)$
- Works like regular hashing but: If $h_1(k)$ causes collision, we try $h_2(k)$ (← probe) and so on. This way each entry on table only has 1 value.
  - Should still aim $\alpha \leq 1/2$

10/23/14

6.006 Lecture

Fernando Trujano

Graphs

$G=(V, E)$

$|V| = n$

$|E| = m$

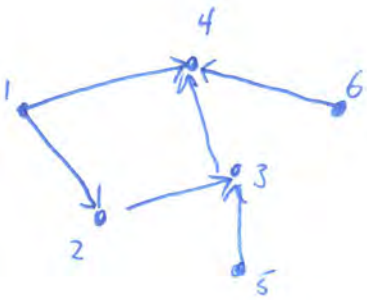
V is a set of vertices

$E \subseteq \{\{a,b\} \mid a,b \in V\}$ or $V \times V$ ← only one

$V = \{1, 2, 3 \ldots n\}$

Why graphs? Functions $F: X \to X$

Relations more



Representation

1 points to 2

Adjacency Matrix

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	1	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	0	0
5	0	0	1	0	0	0
6	0	0	0	1	0	0

$(i,j) \in E$

← 4 points to nothing

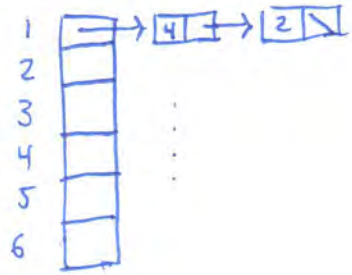
↑ no one points to 1

* Very expensive to make

$m \leq O(n)$

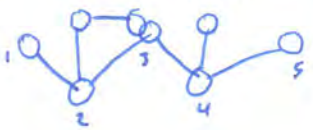
$\frac{m(m-1)}{2}$

Adjacent List



Basic Notions

Path $P \equiv S_0, S_1, \ldots S_k$ s.t $\forall n \{S_i, S_{i+1}\} \in E$



Ex: $1 \to 2 \to 3 \to 4 \to 5$

'0' ← path of length 0 from 1 to 1

Basic Notions (cont)

Connected : For any pair of vertices a, b, there is a path from a to b

connected

not connected

Cycle : A path from x to x of length > 0 in which every vertex appears at most once. ← not thorough enough

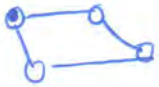
Sequence of vertices $v_1, v_2, \ldots (v_k)$ $k > 1$ ← v_1

such that $\forall i < k \ \{v_i, v_{i+1}\} \in E$

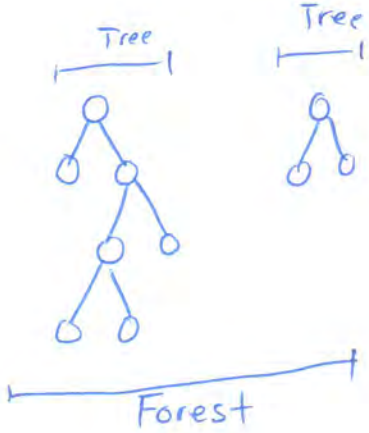
and $(v_k, v_1) \in E$

∴ still not complete.

Cycle



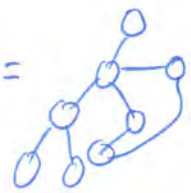
Tree : A connected acyclic graph



Subgraph $G = (V, E)$

$G'(V', E')$ s.t. $V' \subseteq V \wedge E' \subseteq E$ (subset of)

$G =$



Possible $G' =$

Algorithms

Is a graph connected?
What is the shortest distance? } > Solve both problems with one algorithm.
s = start point

Breadth First Search (BFS)

Initially s is marked "0", all other verts "∞"

1. $i \leftarrow 0$ (ie $i = 0$)
2. IF no vertex is marked "i", STOP
3. For all vertex u marked "i" and all edges (u, v),
   IF v marked "∞" mark it "$i+1$"
4. $i \leftarrow i+1$

Thm: This algorithm returns the shortest path from s to any vertex ~~to s~~

Proof (By contradiction)

By induction

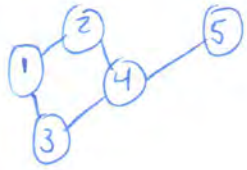
Complexity ??

(next class)

(spoiler) Linear Time.

10/24/14 6.006 Recitation Fernando Trujano

Graph Representation



# of nodes: n
edges: m

Adjacency Matrix

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	0
4	0	1	1	0	1
5	0	0	0	1	0

isNeighbor(u,v): $\Theta(1)$

Space: $\Theta(n^2)$

↳ symmetric

Adjacency List

1: [2, 3]
2 [1, 4]
3: [1, 4]
4: [2, 3, 5]
5: [4]

isNeighbor = Θ(neighborhood of node)

Space: $\Theta(n+m)$

Adjacency List (using HashSet)

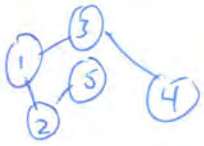
1: {2, 3}
2: {1, 4}
3: {1, 4}
4: {2, 3, 5}
5: {4}

isNeighbor = $\Theta(1)$ * amortized expected

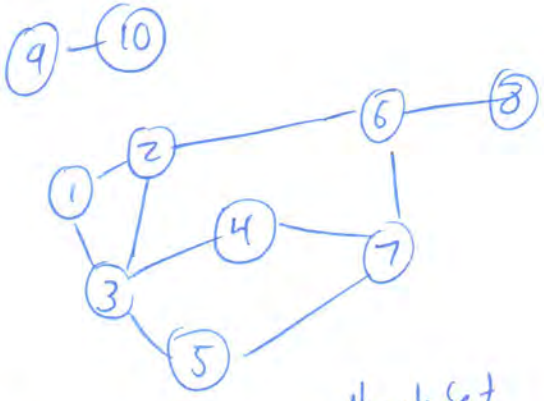
Space: $\Theta(n+m)$

cycle 

not cycle 

Tree 

Directed Acyclic Graph (DAG) 



BFS

HashSet

off limits = {}

queue = [ ]

To expand a node:

For each of the node's neighbors:

if neighbor isn't off-limits

Make neighbor off-limits, return False

Add neighbor to queue

BFS

while non empty:

expand first node in queue

if this node is the one we are looking for: return True.

Sample

queue = [1] off limits = {1}

[2, 3] {1, 2, 3}

[3, 4, 6] {1, 2, 3, 4, 6}

[4, 6, 5]

[6, 5, 7]

[5, 7, 8]

[7, 8]

[8]

Runtime: $\Theta(m)$ (# of edges) *Amortize expected

How to make it faster?

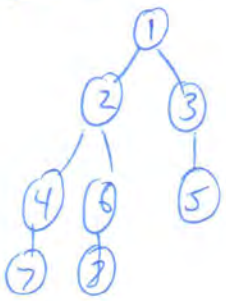
Augment the node

```
class Node :
    self.offlimits = False
```

Now to check Node, simply access node.offlimits instead of searching through the Hash map.

Runtime: $\Theta(m)$

BFS Tree



Have parent pointer in each node.

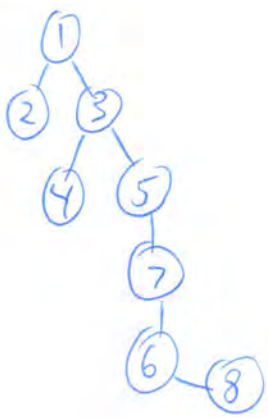
Path from root to any is guaranteed to be the shortest path.

* Not necessarily from any node to any node.

Depth First Search (DFS)

Like BFS, but expand last node in queueue

DFS tree



* No guarantees of shortest path.

BFS++

Find a spanning tree of the graph
↳ subgraph of G that is a tree
↳ with same vertices.
So, can only remove edges

start at s, do BFS
if it is the first time reaching neighbor, mark it's edge

Euler cycle - travel each edge exactly once

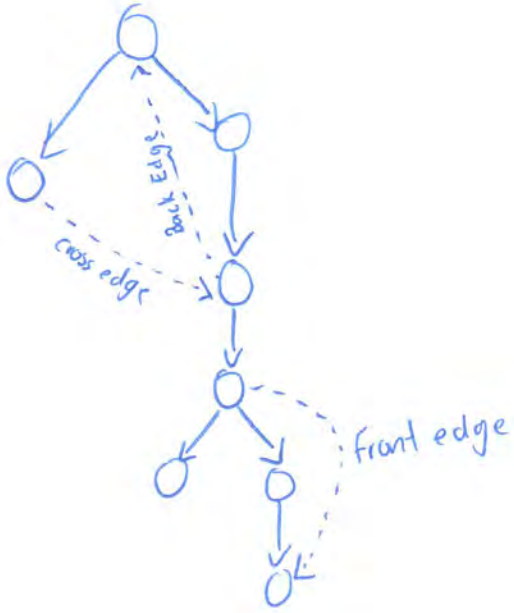
Thm: If Every even degree => Euler cycle

10/29/14

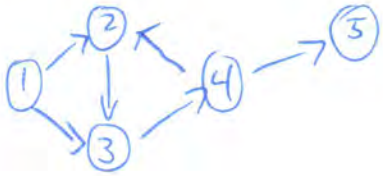
6.006 Recitation

Fernando Trojano

* Problem Set 4 Part B works !!! 40/40



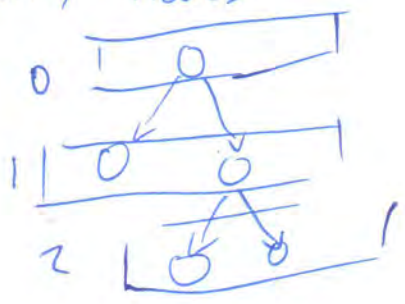
Find cycle in graph:



DFS TREE

IF there is a back edge, then there must be a cycle!

How to classify nodes



n = # nodes
m = # edges

normal DFS $\Theta(m)$

our mod DFS $\Theta(mn)$

Another way to find cycles

Get adjacency matrix A

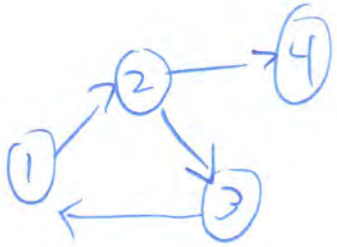
A^2 = Paths of the length 2 from i to j

A^n

So if there is a 1 in the diagonal, there is a cycle of length n

$\Theta(n^3)$ matrix mult

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



10/30/14 6.006 Lecture #17 Fernando Trujano

DFS

- Explore the maze walking from vertex to vertex always using new edges and keeping track of vertex from which you discover a new one.
- When you get stuck in a vertex, backtrack till you find a new edge + traverse it.
- IF you are really stuck, HALT (For later use)
- NUMBER all vertices in the order of discovery

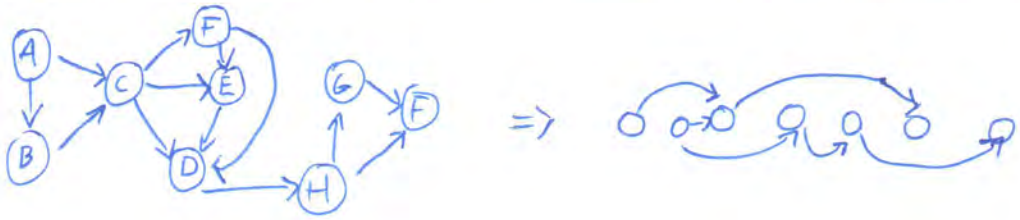
Complexity? $O(n+m)$

6.006 Recitation

Fernando Trujano

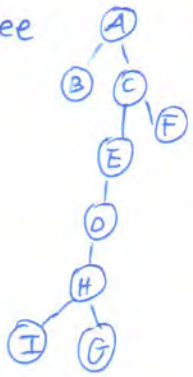
Topological Sort

Arrange nodes left to right such that all edges point left to right



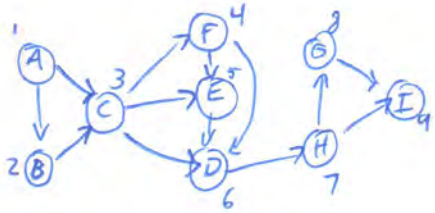
How to topologically sort graph?

- Make sure there are no cycles (DFS w/ no back edges)
- Build DFS Tree



Start at i = # of nodes
Do DFS until you have to backtrack
  Assign current node i
  i--
Repeat untill all nodes labled

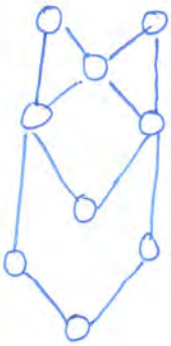
ie, node is inactive



Works b/c a node is not inactive unless all of it's children are inactive
so $9 \rightarrow 8$ could never happen.

Eulerian Tours

- Cycle that goes through each edge exactly once
  - cannot happen if a node has an odd degree



How to Find if this graph has a Eulerian Tour.
- Does not matter where you start

For each edge: BFS

?

Spanning Trees

⊕ Trees from BFS and DFS are Spanning Trees!

⊗ Adding back an edge into a spanning Tree creates exactly 1 cycle.

Number of spanning trees in complete graphs

knodes # spanning trees = k^{k-2}

Notions of Connectivity

Strong: Any vertex can get to every other 

normal: 

weak 

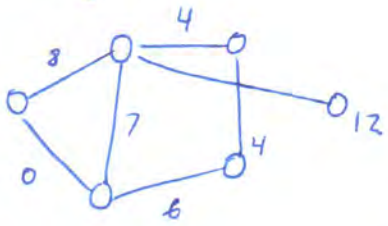
Minimum Spanning Tree

Spanning Tree that minimizes cost.

Kruskwell algorithm: Add lowest cost edge that does not create a cycle

Runtime: $\Theta(m^2)$ m = # edges

Weighted graphs



Dijkstra's Alg

see notes.

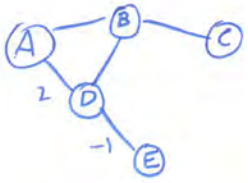
10/5/14

6.006 Recitation

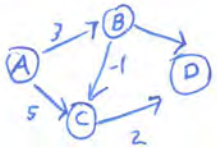
A* without heuristic = Djikstra's

Expected Runtime: average runtime on worst-case input

Negative path costs:



Messes everything up on undirected graphs



Not so bad for directed -- unless there is a negative cycles and cycle is connected to destination point. And start pont is connected to cycle.

So. Undirected

neg weight edge that is connected to sorce

=> No shortest path

Directed

neg wt. cycle => no shortest path

iff (s) -> cycle -> (t)

Djikstra's

(+) Does not work w/ negative weight edges! (See Bellmaid-Ford for tha

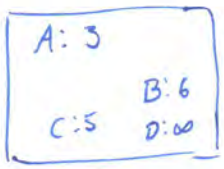
n = # vertices
m = # edges

At each step

- Find vertex ✓ with the minimum distance from the source From among all vertices that we're not sure about yet, make ourselves sure $\Theta(n)$
- For all neighbors of v, decrease their temporary label if need be. $\Theta(m)$
- Repeat until every node has a permanent label. $\Theta(n)$

Which data structure to use?

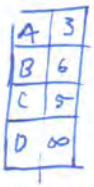
$\Theta(n)$ Extract min
$\Theta(m)$ Decrease-key



← data structure

Array

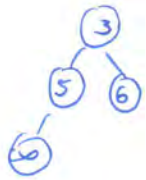
Extract min: $\Theta(n)$
Decrease key: $\Theta(1)$
Overall Runtime: $\Theta(n^2+m) = \Theta(n^2)$



Min Heap

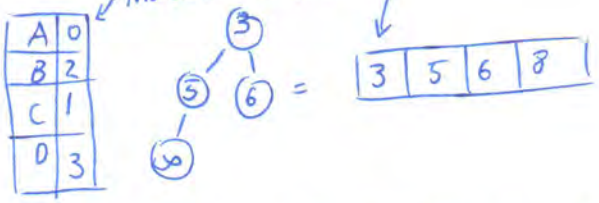
Extract min: $\Theta(\log n)$

Decrease key: hard to search in heaps.



Heap and array? Link them!

← index in array where A is



Could also have pointers in array directly to elements in the heap.

Other Approach: Never delete,

Runtime with either of these methods

Extract-min: $\Theta(\log n)$
Decrease-key: $\Theta(\log n)$

So, Overall Runtime: $\Theta(m \log n + n \log n)$

$\Rightarrow \Theta(m \log n)$

Magical Data Structure

Extract min: $\Theta(\log n)$ amortized
Decrease-key: $\Theta(1)$
Overall Runtime: $\Theta(n \log n + m)$

Magical Data Structure exists: It's called a Fibonacci Heap

6.006 Recitation

Fernando Trujano

1. Dijkstra w/ Negative weights
2. Bellman-Ford
3. DAG
4. Constraint Solving

1. Dijkstra w/ Negative weights

Def Relax (u,v)

```
if d[v] ≥ d[u] + w(u,v)
    d[v] = d[u] + w(u,v)
    #update the distance
```

Dijkstra will fail with negative weights

Runs in $O(|V| \log(|V|) + |E|)$

2. Bellman-Ford

```
INITIALIZE(G)   ← set d[s] = 0, all other ∞
repeat |V|-1 times          } iteration   } O(VE)
    For e ∈ E
        relax(e)
```

Check for negative-weight cycle reachable from source
if ↗ return "Bad Graph!"
else return shortest-path

- Assume no negative weight cycles
- Want to prove that Bellman-Ford returns shortest path

After First iteration: (one edge)

All shortests paths of length 1 discovered

Facts

- Sub path of shortest path is also shortest path
- Most possible edges in shortest path = $|V|-1$
  ↑ since no cycles in shortest path

Induction

Assume that after k iterations, all shortest paths of at most k edges are discovered

Want to prove: After k+1 iterations " " k+1 " "

.
.
.

We can check for negative weight cycle by doing one more iteration and checking for updates.

DAG (Directed Acyclic Graph)

What order to relax the edges in?

Relax edges in order of topological sort $O(|V|+|E|)$

Solving shortest Path Problem

Unweighted (Directed/Undirected)	BFS	$O(\|V\| + \|E\|)$
Weighted w/ nonnegative edge weights (Directed/Undirected)	Dijkstra	$O(\|V\| \log \|V\| + \|E\|)$
DAG	DAG-SP (relax in correct order)	$O(\|V\| + \|E\|)$
General	Bellman Ford	$O(VE)$

11/11/14

6.006 Quiz 2 Review

Fernando Trujano

Hashing

- Dired Acess Table

- Chainging
  - easy way to handle collisions

- Open Addressing

  Double Hashing (# of tries)

  $h_m = h_1(k) + i h_2(k) \bmod m$

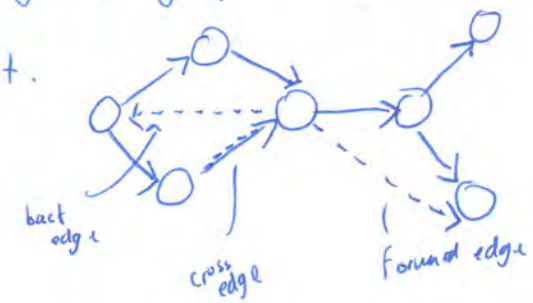
  Rolling Hash - substring matching

Graphs

Representation: Picture, list, Matrix

BFS: shortest path unweighted graph. Explore each level

DFS: explore till end First.



Djikstra

- weighted, directed
- Priority Queue
- $O(V \log V + E \log V)^*$
- does not work with neg weight
  - could try to change graph to make this work

Bellman - Ford

Negative weights, negative cycles

Questions:

# of probes : n keys, m slots

$\text{Prob}(\text{1st probe } \cancel{\text{fails}}^{\text{yay}}) = \frac{m-n}{m}$

$\text{Prob}(\text{2nd } \cancel{\text{fails}}^{\text{yay}} \mid \text{fails}) = \frac{m-n}{m-1}$

(given)

$\text{Prob}(i^{th} \text{ probe } \cancel{\text{fails}}^{\text{yay}} \mid \uparrow) = \frac{m-n}{m-i+1} \geq \frac{m-n}{m}$

$p \geq \frac{m-n}{m}$

$\frac{1}{p} = \frac{1}{\frac{m-n}{m}} = \frac{m}{m-n} = \frac{1}{1-n/m} = \frac{1}{1-\alpha}$ ← load factor

11/12/14

6.006 Recitation

Fernando Trujano

Quiz II Review

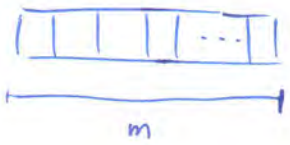
Bidirectional BFS: BFs on both start (s) and goal (t). Faster on average, not on worstcase

Dijkstra's:

Take lowest ~~pe~~ temporary label, make it permanent expand node
↳ relax all edges

Hashing:

Simple Uniform Hash Function: Equal probability to hash everywhere



Odds that k elements hash to the same value: $\frac{1}{m^{k-1}}$

Universal hash family function. $\forall x \neq y$

$$Pr[h(x) = h(y)] \leq \frac{1}{m}$$

	collisions
$h_1(x)$	1, 2, 3, 4, 7, 10, 15
$h_2(x)$	1, 5, 9, 13, 17
$h_3(x)$	1, 6, 11, 16, 20
$h_4(x)$	
⋮	
m	

$$Pr[1 \text{ and } 2 \text{ will collide}] = \frac{1}{m} = Pr[h_1] \quad (\text{picking } h_1)$$

$$Pr[h(1) = h(\cancel{4}) = h(7)] = \frac{1}{m} = Pr[h_i] \quad (\text{Picking } h_i)$$

H contains g different hash functions

h_1	hashes 1, 4, 9, 22, 33 to same value
h_2	" 1, 5, 13, 23, 49 "
h_3	. . . . .
⋮	
h_g	

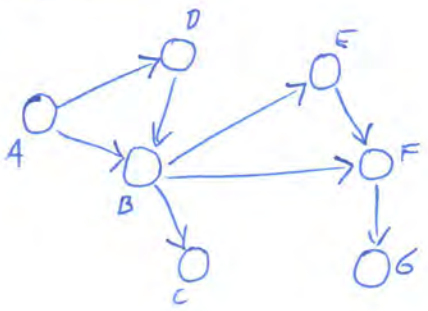
Devil picks worst case input to maximize collisions.

Ex 1, 4, 9, 22, 33

Prob that they will all collide $= \frac{1}{g}$

11/2

Topological Sort



Ordering of the nodes such that edges will go left to right.

* Can't have cycle

Run DFS From source ← Any vertex with no arrows pointing into it.

When node has no active children, add it to end of array

A	D	B	C	E	F	G

Runtime:

Pseudo-Code

d[v] = current calculated distance from s to v.

w(u,v) = Edge weights of (u,v)

π(v) = current parent of v that lead to the shortest path.

```
Relax(u,v):
    if d[v] > d[u] + w(u,v):        // Found a shorter path
        d[v] = d[u] + w(u,v)        // Update path
        π[v] = u                    // Update parent

Initialize (V, E, s):
    for v ∈ V:
        d[v] = ∞ , π(v) = null
    d[s] = 0, π[s] = s

Dijkstra (V,E,s):                   // use on weighted w/ non-neg edges
    Initialize (V,E,s)
    Q.push(all v∈V)                 // Q is a min priority queue
    while Q not empty:
        u = Q.pop                   // u has smallest d[u] value
        for each neighbor v of u:   // For v still in Q
            relax(u,v)
    return d, π
    // Runtime: O(V (extract-min) + E(decrease-key))
    // Look through each edge once and relax each edge once
    // O(V log V + E) using Fibonacci Heaps

Bellman-Ford (V,E,s) // use on weighted w/ neg edges
    Initialize (V,E,s)
    for i=1 to |V|-1:               // Relax all edges |V|-1 times
        for each edge (u,v)∈E :
            relax (u,v)

    for each edge (u,v) ∈ E :       // check edges one more time
        if d[v] > d[u] + w(u,v)     // Shorter path Found
            return False            // Report negative cycle
    return d, π
    // Runtime: O(VE)
```

Bellman-Ford calculaes the shortest path from the source to every vertex in G.

```
BFS(G,s):                           // Will not necessarily find shortest path
    Q = queue()
    V = {} // Hashset of visited nodes
    V.add(s)
    Q.add(s)
    while Q not empty:
        t = Q.dequeue
        if t == goal: return t
        for all adjacent edges in t:
            v = adjacent vertex
            if v not in V:
                V.add(v)
                Q.add(v)
    return None
    // Runtime O(V+E)

DFS: (G,s)
    S = stack()
    S.push(s)
    while S not empty
        v = S.pop()  // most recently added
        if v not discovered:
            v.discovered()
            for all neighbors of v:
                S.push(neighbor)
```

BFS/DFS.
Trees are dependent on start vertex (root) and order of visits.

Types
Tree: edge visits new vertex
Back: leads to ancestor
Forwards: " " decendent
cross: Links two branches

	BFS undirected	BFS Directed	DFS undirected	DFS Directed
Tree	✓	✓	✓	✓
Back	X	✓	✓	✓
Forward	X	X	X	✓
Cross	✓	✓	X	✓

Graph Theory

Chromatic #: min val of k for coloring problem

Basic Coloring Algorithm:
1) Order nodes by largest degree first
2) Order colors c_1, c_2, c_3
3) For all nodes Assign v_i the lowest legal color
will use at most max(degree)+1 colors

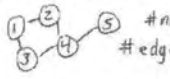
Handshake Lemma: $\sum_{v \in V} \deg(v) = 2|E|$

Definitions: Odd length cycle --> not bipartite
- Walk: Sequence of vertices in G
- Path: Walk with all different V's
- Cycle: Walk where $V_0 = V_k$ and $V_0, V_1 \ldots V_{k-1}$ are different.
- Connectivity: Vertices - if there is a path from v to v
  - Graph: Every pair of vertices is connected
  - ↳ indegree must = out degree + strongly connected
- Tree: Acyclic connected graph
- Spanning Tree: Subgraph (that is a tree) with the same set of vertices.
- A graph of n-nodes is a tree if:
  connected, no cycles and n-1 edges.
- Hamiltonian cycle: Cycle that visits each node exactly once.
- Euler Walk: Walk that traverses every edge once
- Euler Tour: Euler walks that ends at start node
  ↳ cant exists if a vertex has odd deg

Kraskals Algorithm: Find Minimum Spanning Tree (MST)
- start with an empty set
- add the minimum edge that does not create a cycle
- stop when there is no such edge.

Connectivity: Weak = connected w/ directions removed; Strong = normal

Graph representation:



#nodes = n
#edges = m

Adjacency matrix:

$$A = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 0 & 1 & 0 \end{array}$$

List Neighbor: O(V)
isNeighbor(u,v): Θ(1)
Space: $\Theta(n^2)$
A^k_{ij} = # of paths of length k from i ↔ j

Adjacency List (using Hash Set)
1: {2, 3}
2: {1, 4}
3: {1, 4}
4: {2, 3, 5}
5: {4}
* Amortized expected ↓
isNeighbor(u,v): Θ(1)
Space: Θ(n+m)
List Neighbors: O(neighbors)

Hashing

Universal Hashing: $h(k) = [ak + b \bmod p] \bmod m$ (p: large prime; a, b: random #'s 0 ... p-1; m: size of table)

$\Pr[h(k_1) = h(k_2)] = 1/m \quad \forall k_1 \neq k_2$

Family H of hash functions: choose h randomly from H to minimize collisions

Expected Search Time:
- $O(1 + \#\text{keys that collide w/ k})$
- $O(1 + \frac{(n-1)}{m}) \Rightarrow \alpha = n/m$ (load Factor)
  All keys but k ↳ 1/m chance collision
- $O(1 + \alpha) = O(1)$ if $m = \Theta(n)$
  Need to resize table to make sure not rarely do this so O(1) amortized t:

$p = 1 - \alpha$ probability that the first probe will work.
$\Rightarrow$ Expected # attempts $1/1-\alpha$

Collision Resolution

Chaining: linked list on each table entry

Open Addressing: Find another location

Uniform Hashing Assumption:
- Probe sequence of any key is equally likeley to be any of the m! permutations.

Linear Probing: simple but not very good
If collision, try next location until empty spot.
→ Creates clusters, slow things down

Double Hashing: Probe #
$h_m(k) = (h_1(k) + i \cdot h_2(k)) \bmod m$
So next probe will probably not be adjacent.
If $h_2(k) = 1 \ \forall k \Rightarrow$ Linear probing!

Rolling Hash: Remember DNA Pset problem
Calculate a new hash based on previous hash w/o recomputing everything
substring-match

DFS and BFS can be modified to find cycles.
BFS and DFS have the same time complexity O(V+E)
For Open Adressing, failed searches and insertion take the same # of comparisons
Bidirectional BFS performs asymptotically better than BFS in a highly branched graph. Ex, Balanced BST, Rubix Cube, Star graph
Bellman-Ford can still correctly calculate the shortest path from s to t when there are negative weight cycles if all paths from s to t contain no cycles.
DFS on an undirected graph has no forward or cross edges
BFS: O(V+E)
Use on unweighted graphs
Very Efficient on weighted graph with bounded positive integer weights.
Think about replacing an edge of weight x with x dummy nodes
Can work on a weighted directed tree graph (since there is only one unique path!)
Can test bipartiteness: Give alternating labels to vertecies. If at any step a vertex has (visited) neighbors with the same label as itself, return False.
No forward or back edges on an undirected graph
Bellman-Ford: O(VE)
Use on weighted graphs with negative edges/cycles
Invariant: After k iterations, d[v] will store the min weight cost to reach v in at most k hops
If there are no negative cycles. If there is a shortest path from s to to consisting of k edges, after the kth iteration, BF estimate of d[t] will be correct
Dijkstra's Algorithm: O((V+E)logV)
Use on directed graphs with no negative edges *
Very efficient on complete (all nodes connected) undirected graph with positive weights
Can still work with negative edges if all of them come from the source and no edges enter the source
Dijkstra = A* without heutistic
Dijkstra w/ AVL as priority queue has same running time.
Decrease-key and Extract-Min both O(logV) time.
Topological Sort and Relaxation: O(V+E)
Use on positive/arbitrarily weighted DAGs
Does not work on undirected graphs

special Case: No Neg weight edges ssse: DFS with Priority Queue expand node with least weight.
Special Case: Highly branched graphs. -> Bidirectional search.

Problem strategies:

- Shortest path with a detour v.
  Find shortest path from $s \to v$ and $v \to t$
- Add two edges to minimize shortest path.
  copy graph in 3 layers, connected by proposed edges.
  Run Dijkstra From s_1 to t_1, t_2, t_3 and compare.
  Keep track of traversed proposed edges.
- Multi Source Path problem. Find shortest path from a set of starting ptr.
  Solution 1: create node s^* that has a directed edge to every startpoint. Run dijkstra.
  Solution 2: reverse all edges and run dijkstra from t until some s is expanded.
- Dijkstra Modifications:
  Many hard problems can be solved with simple mods.
  Graph: Vertices edges and weight can have diff meaning
  Length: Different meaning Can be changed
  Priority Queue: Different comparison relation (min/max)
  Relaxation: Modified process
  Distance Initialization: different starting distances

Flight Agenda Problem:

Problem 13. The Longest Quiz Path in a Tree [15 points]

Let $T = (V, E)$ be an undirected tree (a connected graph with no cycles) with non-negative edge weights. Design and analyze an efficient algorithm to determine the length of the *longest path* in T, i.e., the value of

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ denotes the length of the shortest path from u to v. For full marks, an $O(V)$ algorithm is required.

Hint: Start by selecting an arbitrary vertex $r \in V$ to be the *root* of T.

Solution: First, select an arbitrary vertex $r \in V$ to be the *root* of T, and run a DFS or BFS to direct all edges outwards, so that we now have a rooted tree. For any given path P, there is a unique "highest" node, call it r, in this tree. (The path starts somewhere, goes up the tree, and then goes down.) In this case, we say that P is rooted at r.

Now, for each vertex v, define $P[v]$ to be the subproblem of finding the longest path from v to some leaf in the subtree rooted at v. We then have the recurrence:

$$P[v] = \max_{u} (P[u] + w_{u,v})$$

This is a DP which with $O(V)$ subproblems, where each subproblem takes $O(V)$ time to evaluate. However, notice that there is only one evaluation for each edge, and so the computation takes $O(E)$ time total. Since this is a tree, $|E| = |V| - 1$, and so $O(E) = O(V)$.

Next, we want to compute $L[v]$, the length of the longest path rooted at v. First, we claim that the longest path must go from a leaf to another leaf. If not, we could make a longer path by including another edge adjacent to the non-leaf.

Thus, to do this, for each child u, consider the quantity $P[u] + w_{u,v}$. Let u_1 and u_2 be the two children with the largest values of this. Then, we have

$$L[v] = P[u_1] + w_{u_1,v} + P[u_2] + w_{u_2,v}$$

Computing all the $L[v]$ values takes $O(V)$, by the same argument as for $P[v]$.

Finally, we have that the length of the longest path is simply

$$\max_{v} L[v]$$

Problem 7. Power to the Network [10 points] There are microwave towers along the 2451 miles of Route 66, allowing information to be transmitted from one end to another. New towers are added from time to time; when there are n towers we denote their positions by numerical values $x_1, x_2, \ldots, x_n$, where $x_i < x_{i+1}$ for $i = 1, 2, \ldots, n-1$. The power used to transmit a signal from one end of Route 66 to the other is proportional to

$$P = \sum_{i=1}^{n-1} (x_{i+1} - x_i)^2 .$$

In this problem, you must design a *Signal Tower Location* data structure that maintains the location of the towers and supports the following operations:

- `STL.insert(x)`: for a number x, add a new tower at position x into the Signal Tower Location structure STL.
- `STL.getpower()`: return the value of P, as calculated above, for all tower locations currently stored in STL.

Describe a data structure that supports `getpower()` in $O(1)$ time, and `insert(x)` in $O(\log(n))$ time, where n is the number of towers currently inserted into the Signal Tower Location structure. You may assume that no two identical values of x will ever be inserted.

Solution: Use an AVL tree, and augment it with a single integer value for P (in other words, a single number is added to the data structure; not one number per node.) Upon insdering a new tower at position x, recalculate the value of P and update it. Upon inserting x_i, we can update the value of P in $O(\log n)$ time by looking at the predecessor and successor of x_{i-1} and x_{i+1} of x_i in the AVL tree, and then adding $(x_i - x_{i-1})^2 + (x_{i+1} - x_i)^2$ and subtracting $(x_{i+1} - x_{i-1})^2$.

Problem 4. Changing Colors [15 points]

Consider a directed graph G where each edge $(u, v) \in E$ has both a weight $w(u, v)$ (not necessarily positive) as well as a color $color(u, v) \in \{\text{red}, \text{blue}\}$.

The weight of a path $p = v_1 \to v_2 \to \ldots \to v_k$ is equal to the sum of the weights of the edges, *plus* 5 for each pair of adjacent edges that are not the same color. That is, when traversing a path, it costs an *additional* 5 units to switch from one edge to another of a different color.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t, and analyze its running time. (You may assume that there exists such a path.) For full credit, your algorithm should have a running time of $O(VE)$, but partial credit will be awarded for slower solutions.

Solution: Construct a new graph G' as follows: for each vertex $v \in V$, create *two* vertices, a red vertex v_R and a blue vertex v_B, in G'. For each edge $(u, v) \in E$, if $color(u, v) =$ red, create (u_R, v_R) in G'. Otherwise, create (u_B, v_B). In either case, the new edge should have the same weight as the original. Finally, for each vertex $v \in V$, create the edges (v_R, v_B) and (v_B, v_R) in G', each with weight 5.

This works because all the red edges run between red vertices in G', and all the blue edges run between blue vertices. Switching colors requires traversing an edge in G' between a red and a blue vertex, incurring the extra cost of 5 units.

Now, run Bellman-Ford twice: once starting from s_R and once starting from s_B. Examine the paths ending at t_R and t_B. Determine which of the four paths ($s_R \rightsquigarrow t_R$, $s_R \rightsquigarrow t_B$, $s_B \rightsquigarrow t_R$, $s_B \rightsquigarrow t_B$) is shortest. Strip the subscripts to recover the desired path in G.

Constructing G' takes $O(V + E)$ time. G' has $2V$ vertices and $V + E$ edges. Running Bellman-Ford on G' takes $O(V'E') = O((2V)(V + E)) = O(VE)$ time. Thus the total running time is $O(VE)$.

Problem 9. Star Power! [20 points]

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which (along with making you temporarily invincible) lets you traverse the edge of your choice for free. In other words, you may change the weight of any one edge to zero.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t, given that you may set one edge weight to zero. Analyze your algorithm's running time. For full credit, your algorithm should have a running time of $O(E + V \lg V)$, but partial credit will be awarded for slower solutions.

Solution 1: Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t. Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \to v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \to v \rightsquigarrow t$ is the desired path.

The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra: $\Theta(E + V \lg V)$.

Ex: Graph with directed and undirected edges. assign each undirected edge a direction and create no cycles.
A: Topo sort directed edges For each undirected draw edge that goes in direction of lower sort to higher sort.

Sample h_1: 0 0 1 Is $H = \{h_1, h_2, h_3\}$ a universal family?
h_2: 1 0 0
1 2 3

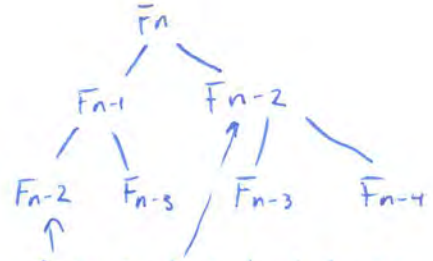
A: No because $h_1(1) = h_1(2)$ regardless of the choice of i => elements 1 and 2 will collide with prob $> \frac{1}{3} = \frac{1}{2}$
Q: Suppose we modify h_1 so $h_1(2) = 1$ Is $H = \{h_1, h_2, h_3\}$ a universal family?
A: Yes, now elements 1 and 2 collide with $p_r = 1/2$, 2 and 3 w $p_r = 1/2$ and 1 and 3 $p_r = 0$

Dynamic Programming

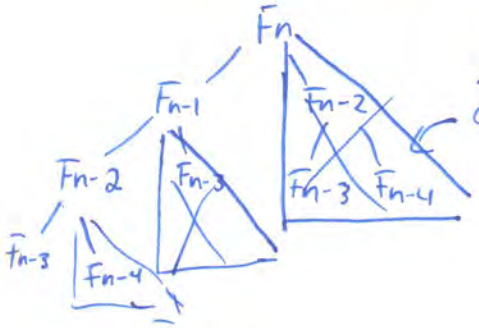
Fibbonacci numbers

```
F(n):
  if n > 2  F = 1
  else F = Fib(n-1) + Fib(n-2)
  return F
```

Run time = $\Omega(c^n)$ for some constant $c > 1$ — Bad!



Solution: Dynamic Programming Reuse, reuse, reuse



Already calculated
can look it up in the table (memo)
$\Theta(1)$ time

Memorized DP Algorithm

```
memo = {}
Fib(n):
  if n in memo  return memo[n]      // "memorized calls" - Θ(1) time
  if n ≤ 2  f = 1                   // "non-memorized calls"
  else F = Fib(n-1) + Fib(n-2)      //   = # sub problems = n
  memo[n] = F                       // time / non-memorized
  return F                          //   = time / sub problem = Θ(1)
```

Total time = #subproblems · time/subproblem

= $O(n)$

DP Notions

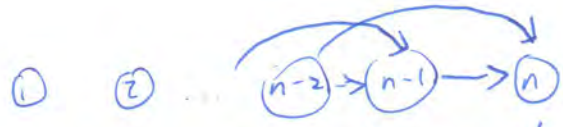
1) subproblems — {fib(1), fib(n)}
   // polynomially many of them

2) memoize = reuse solutions to subproblems

3) Sub problem dependency DAG

Another perspective : bottom up

```
memo = {}
for k in [1 ... n]
    if k ≤ 2 f = 1
    else f = memo[k-1] + memo[k-2]
  fib[k] = F
return fib[n]
```



sub problem dependency graph.

DAG Topological Sort

Shortest Paths

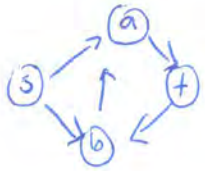
directed - weighted, no negative weight edges
single source, single target.

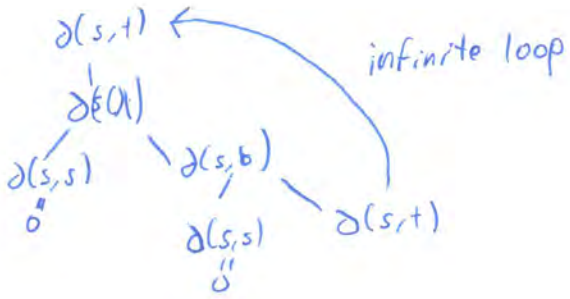
Find s(s,t) ~~grad~~ given s, t, G = (V,E)

Sup problems { δ(s,v) For all v∈V }

δ(s,v) = min distance δ(s,w) + w(w,v)

Watch out for cycles!



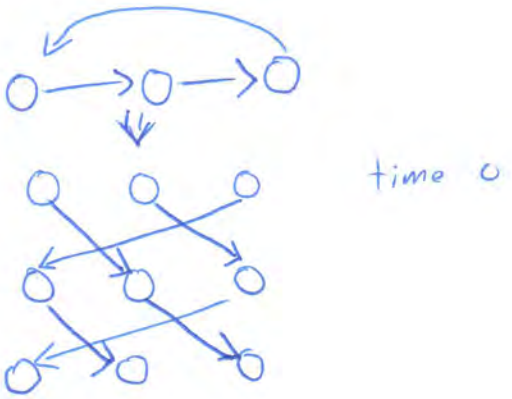


w/o cycles

$n = |V|$

$m = |E|$ $\quad O\left(\sum_{v \in V} \text{indeg}(v)\right) = O(n+m)$

Need to make graph acyclic:



time 0

sub problems

$= \{ \delta_k(s,v) =$ weight of shortest path $s \to v$ with at most k edges

11/20/14

6.006 Lecture

Fernando Trujano

Dynamic Programming II

- 4 step method for dynamic programming design
- Ex: Text Justification

- Parent pointers

4 step method

1. Define subproblems
2. Relate subproblem solutions
   - DP recursion
   - Notion: subproblem graph
   - sanity check: DAG
3. Write out DP alg
   either
   - recurse and memoize

   or
   bottom up alg (Mechanical)
4. Solve original problem
   Typically Trivial

=> # sub problems

=> ~~#~~ time/sub problems

RT: (# subproblems)(time/sup problem)

Text Justification

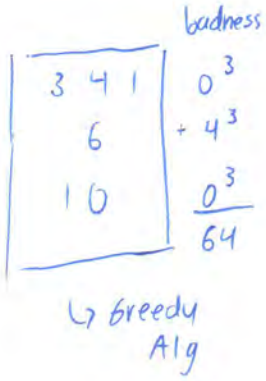
words:

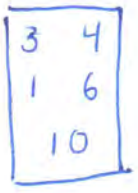
$$\text{badness}(i:j) = \begin{cases} \infty \text{ if words don't fit in one line} \\ (\#\text{extra white spaces})^3 = \left(L - \sum_{i \le k \le j} w_k - (j-i)\right)^3 \end{cases}$$

Find an alignment $w_1 \dots w_n$

$(w_1 \dots w_{i_1})(w_{i_1+1} \dots w_{i_2}) \dots (\quad)$

badness = badness$(1, i_1)$ + badness(i_1+1, i_2) + ...





Input: words[1 . . . n]

① Subproblems Guess where first line ends

A[j+1] = best way to pack words[j+1 : n] into lines

↳ {A[1] . . . A[n]}

② $A[i] = \min_j (A[j] + \text{badness}(i \ldots j-1))$

$= \theta(n)$

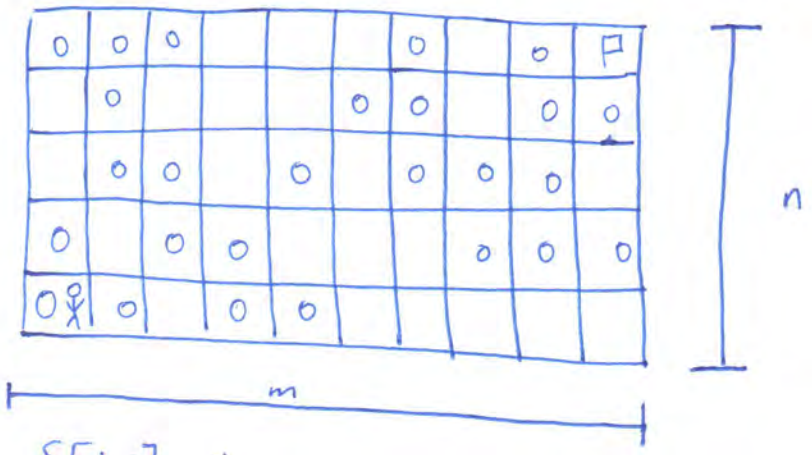
$\theta(n^2)$

11/21/14 6.006 Recitation Fernando Trujano

Dynamic Programming

Robot Problem

Pick up as many coins on way to destination



Bottom left

$S[i,j]$ = # of coins collected on optimal path from (i,j) to (n,m)

$$S[i,j] = \max(S[i+1,j], S[i,j+1]) + C_{ij}$$

$$c[i,j] = \begin{cases} 1 & \text{coin at } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

Basecase: At the Flag

$S[m,n] = C_{mn}$

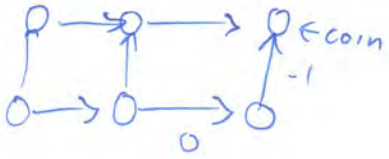
Solution: $S[0,0]$

Runtime: $\Theta(mn)$

subproblems: mn

Time / subproblem: $\Theta(1)$

Another solution.. use graphs



Run DAG shortest path.

UNO

Find longest subsequence that is valid.

7 Green, 7 Red, 4 Green, 4 Blue, Wild

* only allowed to play in order of hand left to right

A[3] = 4 Blue

S[i] = ~~i to the end of the list~~
longest possible sub seq using cards 0 ... i

S[0] = 0

$S[i] = \forall j \; (\max_{j \le i} S[j])$
A[j], A[i] valid

Look at all previous cards from ones that I have a legal connection to, pick the one with the largest subsequence.

Runtime: $\Theta(n^2)$

n sub problems
$\Theta(n)$ per subproblem

Another Solution ... use graphs

use DAG longest paths

DAG shortest paths <-> Dynamic Programming (basically the same!)

coin change

- n coin denominations
- amount C
- using as few coins as possible

Ex: 1¢, 4¢, 5¢

C = 12¢

Greedy Alg: 5¢ + 5¢ + 1¢ + 1¢ = 4 coins

4¢ + 4¢ + 4¢ = 3 coins

↳ not the best

$s[i]$ = the fewest coins needed to make i ¢

$s[C]$ = SOLUTION!

$$s[i] = \min_{\substack{j<i \\ i-j \in \text{set of coins}}} (s[j] + 1)$$

C subproblems

$\Theta(n)$ time to solve

Runtime: $\Theta(nC)$

pseudo polynomial

↳ C could be *really* big!

12/2/14

6.006 Lecture

Fernando Trujano

Dynamic Programming IV

1) Edit Distance / sequence alignment

2) Knapsack problem
(and on to NP completeness)

Edit Distance (string/sequence, alignment)

Given strings x and y, how many changes to go from x to y?
↳ insert, delete, replace char

Subproblems: Find edit distance between $x[i....n]$ and $y[j....n]$
$\#$ = $\Theta(n^2)$ $\quad$ $\swarrow$ $ED(i,j)$

DP Recurrence

$$ED(i,j) = \min \begin{cases} \text{cost(del)} + ED(i+1, j) \\ \text{cost(insert)} + ED(i, j+1) \\ \text{cost(replace)} + ED(i+1, j+1) \end{cases}$$

Knapsack

n items each value v_i weight w_i
total max weight W

Goal: Pick subset $S \subseteq [1...n]$ s.t
1) total wt $\leq W$ $\quad \sum_{i \in S} w_i \leq W$
2) total max value $\quad \max \sum_{i \in S} v_i$

Knapsack

Subproblem knap[i] = best subset of items $\{i, i+1, \ldots n\}$

Recursion

$$knap[i] = \max \begin{cases} knap[i+1] + v_i & \text{if } i \text{ goes into knapsack} \\ knap[i+1] & \text{" doesn't} \end{cases}$$

Not good enough! need to keep track of weight

subproblem knap(i, x) = best subset of items $\{i, i+1, \ldots n\}$ into a knapsack of capacity x

Recursion

$$knap(i, x) = \max \begin{cases} knap(i+1, x - w_i) + v_i \\ knap(i+1, x) \end{cases}$$

Runtime?

subproblems = nW

time/subproblem = $\theta(1)$

$\Rightarrow \underbrace{\theta(nW)}$ (bad if W is very big — not polynomial in n)

not polynomial $\Rightarrow$ pseudo poly time = polynomial in #items in input and size number of #'s

polynomial time means polynomial in the number of bits of input.

$$\sum \log v_i + \sum \log w_i \leq n \cdot \log W + n \max(\log v_i)$$

12/3/14 6.006 Recitation Fernando Trujano

Dynamic Programming

- LIS
- Balanced Partition
- Knapsack

Longest Increasing Subsequence

$x = [0, 8, 4, 6]$

LIS = 0, 4, 6 => 3 $\qquad M = [1, 2, 2, 3]$

$M(i)$ = longest increasing subsequence that ends at $x[i]$

$$= \max_{0 \le j < i} M[j] + 1 \quad s.t \quad x[j] < x[i]$$

Runtime: $O(n^2)$

But it can be Faster....

$M'(i)$ = smallest $x[k]$ s.t $x[k]$ is the last element in an increasing sequence of length i

$X = [0, 8, 4, 6]$

$M = [\infty, \infty, \infty, \infty] \rightarrow [0, 4, 6, \infty]$

(1 2 3 4)

return last index $\neq \infty$

Runtime: $O(n \log n)$

Balanced Partition

Given a set S of positive integers, determine if ∃ a partition

$S = [3, 1, 1, 2, 2, 1]$

S_1, S_2, such that $sum(S_1) = sum(S_2)$

Ex:

$S_1 = [3, 1, 1]$ or $S_1 = [3, 2]$

$S_2 = [2, 2, 1]$ $S_2 = [1, 1, 1, 2]$

Tip: Partitions should each be equal to $sum(S)/2$

$M(i, j)$ = True if there exists a subset of the first j elements in S that sum to i.

False otherwise

= True if $M(i, j-1)$ is true

or $M(i - S[j], j-1)$ is true

sum i \ index j	0	1	2	3	4	5	6
0	T	T	T	T	T	T	T
1	F	F	(T)	T	T	T	T
2	F	F	F	(T)	T	T	T
3	F	T	T	T	T	T	T
4	F	F	(T)	T	T	T	T
5	F	F	F	(T)	T	T	T

return $(M[N/2][n])$

6.006 Lecture #24

Fernando Trujano

Efficient algorithms $\Rightarrow$ { Proofs, Cryptography

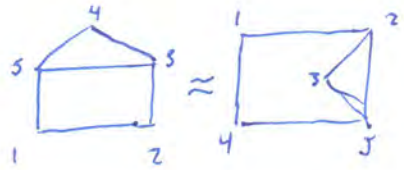
Efficient Proof?

- short string, easy to verify
  - short: Poly long (in length of statement)
  - easy: Poly time

"NP"

Ex.

$ISO = \{(G,H) : G \overset{\text{isomorphic}}{\approx} H\}$



Vertices can be mapped ↗

Proof: 4, 5, 3, 2

$NISO = \{(G,H) : G \not\approx H\}$

Proof: . . . .

not short ☹

Interactive Proofs

- See notes

12/5/14

6.006 Recitation

Fernando Trujano

IP and OP Proofs

Isomorphism

To prove: Give a correct mapping of vertices

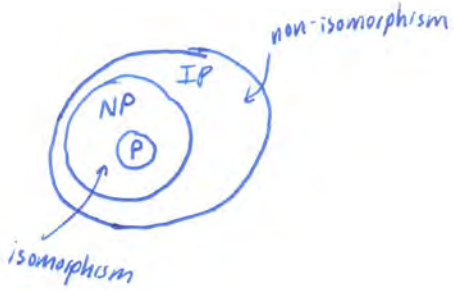
$\theta(V+E)$

To disprove

Micali strategy. Interactive proof (IP)

① Professor secretly and randomly selects a graph and randomly permutes it
② Asks students which ones he permuted
- Same as proving isomorphism
③ If students get it right multiple times, the graphs are probably not isomorphic

P: Solvable in Poly Time
NP: Poly time-checkable proof of solution



So Isomorphism check can be checked in polytime NP but cannot (yet) be solved in polynomial time P.

Cool Fact: Since mathematical proofs can be checked in poly-time so if P = NP then there should be an algorithm to solve any proof in poly time

Zero Knowledge Proof

Proof that I solved a problem without actually revealing the entire solution.

- Recall 3-coloring graph post-it note example.

12/10/14

6.006 Recitation

Fernando Trujano

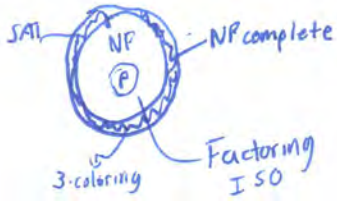
"in NP" - solutions can be confirmed in poly time - upper bound

"NP-hard" - at least as hard as everything in NP - lowerbound

"NP-complete" - in NP and NP hard

SAT: $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_1)$ ← boolean

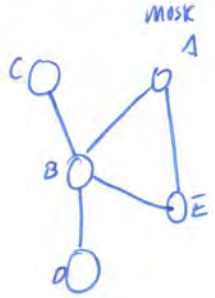
Large list of NP complete problems

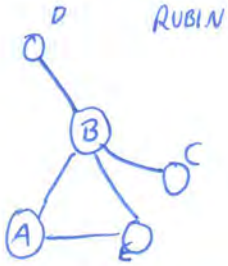


NP = Nondeterministic polynomial

Graph Isomorphism

Zero knowledge Proof





→ that the graphs are isomorphic

Prove that I have an answer without giving out the solution

Ask for one piece of info (since both would give answer away), if I'm cheating, half of the time I'll be wrong.

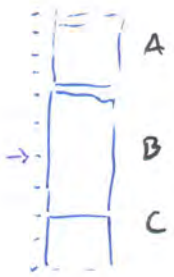
Iterate this process, I'll probably be wrong if cheating

Final

30% of grade

cutoffs (last year)

A: 40%
B: 40%
C: 20%



6.006 Review

Dynamic Programming

Box stacking

Input: n boxes (all orientations of original set of boxes), each with
height h_i
width w_i
depth d_i
$w_i \leq d_i$

Goal: Construct stack of max height

Can only stack box i on j if $w_i < w_j$ and $d_i < d_j$

base

$H(j)$: tallest stack of boxes with box j on top

$$H(j) = \max_{\substack{i<j \\ w_i > w_j \\ d_i > d_j}} \{H(i)\} + h_j$$

Runtime: $\Theta(n^2)$

Solution $\max_j \{H(j)\}$

↳ which box will end up on top?

The Integer Knapsack Problem

Input: n types of items, each with a size s_i (integer) and value v_i
Knapsack capacity C.

Goal: Pack items in knapsack, maximize value
OK to use multiple instances of item for this problem

$M(j)$: max value one can pack into a knapsack of capacity j

For $j = 1$ to C # bottom up approach
Compute $M(j)$

$$M(j) = \max\left\{ M(j-1),\ \max_{\text{items } i}\left\{ M(j-s_i) + v_i \right\} \right\}$$

- $M(j-1)$: j^{th} slot is empty
- $M(j-s_i)+v_i$: some item i occupies j^{th} slot
  - $M(j-s_i)$: Fill a knapsack of size $j-s_i$ optimally
  - $+v_i$: Add value

Runtime: C sub problems, n per sub problem $O(nC)$

Solution: store a back pointer to $M(\sim)$ that was max.
$M(C)$ will be value. Following back pointers will give items used.

Making change

Input: n denominations of coins of values

$1 = v_1 < v_2 < v_3 < \dots < v_n$

Goal: make change for amount of money C.

use as few coins as possible.

(Almost identical to Integer knapsack)

coins → items

value v_i → size s_i

value = -1

↑ so knapsack algorithm will try to use as little coins as possible. □

DP way (Making change)

$M(j)$: min # coins required to make change for amount of $ j

$$M(j) = \min_i \{ M(j - v_i) \} + 1$$

Runtime: $O(nC)$

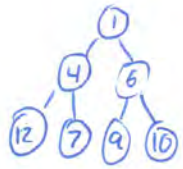
12/13/14

6.006 Final Review

Fernando Trujano

Heaps

- Implemented as array 1 4 6 12 7 9 10



Visualized as nearly complete binary tree

- Build takes $O(n)$

Min-heap property: Parent smaller than children

Priority Queue:	w/heap
MIN()	$O(1)$
EXTRACT-MIN()	$O(\log(n))$
DECREASE-KEY()	$O(\log(n))$
INSERT(n)	$O(\log(n))$

BST

$x.left \leq x < x.right$

Implemented with pointers

Operations: (height of tree)

SEARCH(n)	$O(h)$
INSERT(k)	$O(h)$
DELETE(k)	$O(h)$
PREDECESSOR(k)	$O(h)$
SUCCESSOR(k)	$O(h)$
IN-order-TRAVERSAL()	$O(n)$

AVL Tree

Balanced since many operations are $O(h)$

Augment each node with height.

Maintain property: $|x.left.height - x.right.height| \leq 1$

$h = O(\log(n))$ ← max height of childs + 1

Augmentations

- Store extra information
- Maintainable when doing inserting / deleting

Ex: RANK

Rank(x): # ≤ x

Try 1:
Traverse the tree in order until we get to x. O(n) time

Try 2:
Augment each node with their rank.
Makes INS, DEL O(n) time ← bad

Try 3:
Augment each node with the size of sub tree
INS; change augmentation during walk while augmenting: O(log n)

```
Rank(x):
    r = 0
    search for x:
    right turn at y:  r += 1 + y.left.size
    if x: r += x.left
    return r
```

Sorting

Merge Sort(l)

split list into 2 halves
merge sort each half
merge halves

1 8 10 2 5 6 9 3

↓

1 2 3 5 6 8 9 10

Runtime: $\Theta(n \log n)$

Lower Bound comparison sort

outcomes I can distinguish ≥ # possible outcomes

$2^h \geq n!$

Runtime: h

So,

$2^h \geq n!$

runtime = $h \geq \log n!$

$\geq n \log n$ → So we can't do better than $n \log n$ using comparison based sorting

Radix Sort

Assume input is array of integers bounded by n^c (c constant)

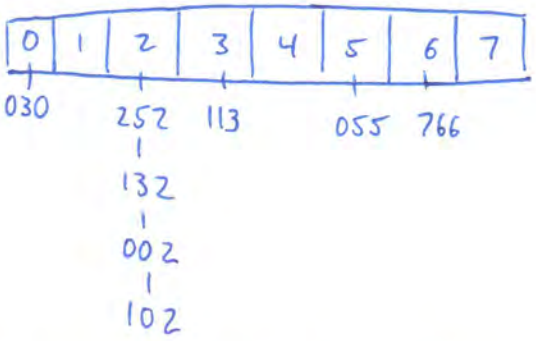
Input: 170 45 75 90 502 2 24 66

bounded by 8^3

1) Write in base 8.

252 055 113 132 766 002 030 102

2) Do counting sort keyed by least significant digit

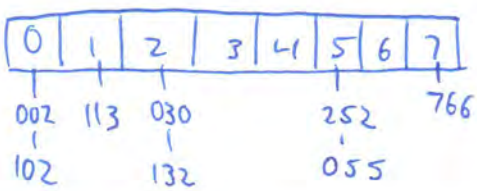


stable: tie orders preserved

3) Scan left to right, top to bottom

030 252 132 002 102 113 055 766

4) Do counting sort keyed by 2nd least significant digit



5) Repeat 3), 4), 5) → take output back to base 10 → sorted!

Numerics

Eucledian Algorithm: $a, b \to \gcd(a,b)$ in polynomial time

$$\gcd(a,b) = \gcd(a \bmod b, b)$$

$O(\log a)$ steps

Extended EA (Pulverizer)

$a = 240 \quad b = 46$

i	Quotient q_i	remainer r_i	s_i	t_i
0		240	1	0
1		46	0	1
2	5	10	1	-5
3	4	6	-4	21
4	1	4	5	-26
5	2	2 ← GCD	-9	47
		0		

Bezout's Equation

$$xa + yb = 1$$

So,

$\gcd(a,b) = d \to xa + yb = d$

invariant: $r_k = a \cdot s_k + b \cdot t_k$

$\hookrightarrow -9 \cdot 240 + 47(46) = 2$

Chinese Remainder Theorem

$n_1, \ldots, n_k$ pairwise relatively prime

$a_1, \ldots, a_k \ni \gcd(a_i, n_i) = 1$

$$CRT: \exists x : \begin{cases} x \equiv a_1 \pmod{n_1} \\ \vdots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

See Notes

...

Fermat's Litte Theorem

$$a^{p-1} \equiv 1 \pmod{p}$$

... See Notes

Hashing

a data structure with

insert (x)
delete (x)
search (x)
} O(1)

- hash table - list/array of size m
- hash function - assigns to x an index in hash table
- collision strategy - what if x and y hash to same value

Hash Functions

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

ex: $h(k) = k \bmod m$ => bad since we want collisions that are hard to predict

ex: Universal Hashing

$h(k) = [ak + b \bmod p] \bmod m$

a, b chosen randomly from $\{1, \ldots, p-1\}$

for each pair of keys k_1, k_2 (collision)

$Pr(h(k_1) = h(k_2)) = 1/m$

Universal Hashing Assumption

Collision Strategies

Chaining - keep linked list at each slot on table

Open Addressing - only one item per slot in hash table

- modified hash function:

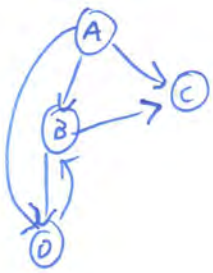
$h: U \times \underbrace{\{0, 1, 2, \ldots, n-1\}}_{\text{attempts}} \to \{0, 1, \ldots, n-1\}$

must: $n < m$

Double Hashing . . .

Graphs

Graph representation: Adjacency List , Adjacency Matrix



A: [B, C, D]
B: [C, D]
C: []
D: [B]

	A	B	C	D
A	0	1	1	1
B	0	0	1	1
C	0	0	0	0
D	0	1	0	0

	Adjacency List	Adjacency Matrix
Finding if edge exists	O(neighbors)	O(1)
visit all neighbors	O(neighbors)	O(V)

BFS / DFS

Trees are dependent on start vertex (root) and order of visits

Tree edge - edge visits new vertex
Back - leads to ancestor
Forward - leads to decendent
cross - links two branches

	BFS		DFS	
	undirected	directed	undirected	directed
Tree	✓	✓	✓	✓
Back	X	✓	✓	✓
Forward	X	X	X	✓
Cross	✓	✓	X	✓

Cycle Detection

DFS - easy to print out vertex stack when back edge encountered.

Topological Sort

Gives ordering of vertices for directed graphs s.t. For all edges (u,v) u comes before v in the ordering.

only exists for DAGs

Shortest Paths Algorithms

condition	algorithm	runtime
Unweighted	BFS	$O(V+E)$
DAG	Topo sort DFS relax in order	$O(V+E)$
Non-negative weights	Dijkstra	$O(V)$ extract-min $O(E)$ decrease-key $O(V \lg V + E)$
No neg-weight cycles	Bellman-Ford	$O(VE)$

Dynamic Programming

Technique to solve problems by breaking down into smaller sub problems

Optimal substructure — dependence on optimal answer to smaller problems

DAG dependence of subproblems.

Recurse with memoization

- Don't need to figure out order to solve problems.
- Easy to implement and understand recursion

vs. Bottom up

- No recursion overhead or max recursion depth
- Easy to analize runtime

7/8

Parent Pointers

want to recover elements of solution and not just best value.
Also memoize the problem that optimal value came from.
At end- retrace from best answer to get sequence that makes up the answer.

Practice Problems

- See previous notes

Dynamic Programming

1) Define Sub problems
2) Relate subproblem solutions w/ recurrence + base case
3) Recurse and memoize or bottom up DP table
4) Solve original problem
Analize runtime

Longest Increasing Subsequence

Input: $A_1 \dots A_n$
Goal: Find longest strictly increasing sequence (not necessarily contiguous)

$L(j)$ = LIS ending at position j

$L(j) = \max_{i<j} \{ L(i) \} + 1$
$A[i] < A[j]$

Solution = $\max_j \{ L(j) \}$

Runtime: $O(n^2)$

Edit Distance

Input: Strings $A[1..n]$ $B[1..m]$
Costs: C_i, C_d, C_R ← replacement, deletion, insertion
Goal: Min cost of transforming A → B

$T(i,j)$ = min cost to transform $A[1..i]$ into $B[1..j]$

$T(i,j) = \min \begin{cases} C_d + T(i-1, j) & \text{delete } a_i \\ T(i, j-1) + C_i & \text{transform } A \\ T(i-1, j-1) & : A[i] = B[j] \\ T(i-1, j-1) + C_R & : A[i] \neq B[j] \end{cases}$

$O(mn)$ #subproblems

Solution: $T(n, m)$

Balanced Partition ("Fairly dividing up gifts")

Input: n integers S
Goal: Partition integers into S_1 and S_2
Minimize $|\text{Sum}(S_1) - \text{Sum}(S_2)|$

$P(i,j) = \begin{cases} 1 & \text{if some subset of } \{A_1 \dots A_i\} \\ & \text{has sum } j \\ 0 & \text{otherwise} \end{cases}$ $\quad i = 0 \dots n$, $j = 0 \dots nk$

$P(i,j) = 1$ if $P(i-1, j) = 1$ or $P(i-1, j-A_i) = 1$
$= \max\{ P(i-1, j), P(i-1, j-A_i) \}$

Coin Change

Input: n denominations of coins. $1 = V_1 < V_2 < \dots < V_n$
Goal: make change for amt of money C. use as few coins.
$M(j)$: min # coins to make change for j

$M(j) = \min \{ M(j - V_i) \} + 1$

Runtime: $O(nC)$

Building Bridges

Goal: Build as many (non-crossing) bridges as possible.
$X(i)$: index of corresponding city on Northern bank.
(compute $X(i)$'s in $O(n \log n)$ time.

Find longest subsequence of $X(1) \dots X(n)$ $O(n^2)$

Integer Knapsack Problem

Input: n types of items, each with size S_i (int) and value V_i, knapsack capacity C
Goal: Pack items in knapsack maximize value. (repeated items ok!)

$M(j)$ = max val pack in knapsack $c=j$

For $j = 1$ to C # bottom up approach
compute $(M(j))$

$M(j) = \max \{ M(j - S_i) + V_i \}$ some item i in j slot
j^{th} slot empty or

Runtime: C subproblems, Fill knapsack of size $j - S_i$ optimally
n per subproblem $O(nC)$

Solution: Store a back pointer to $M(-)$ that was max.

$M(C)$ will be result. Following back pointers will give items used.

Box Stacking

Input: all orientations of original set of boxes each with depth d_i; h_i = height, $w_i \leq d_i$

Goal: Construct stack of max height can only stack box i on j if $w_i < w_j$ and $d_i < d_j$

$S = (\sum A_i)/2$
Find min val of S_i such that $P(n,i) = 1$

$\Rightarrow \min \{ S_i : P(n-i) = 1 \}$

$\text{sum}(S_1) = i$
$\text{sum}(S_2) = 2S_i$

Solution: $\min \{ S_i : P(n-1) = 1 \}$
$\Omega \min_{i \leq S} \{ S_i : \text{since } |\text{sum}(S_1) - \text{sum}(S_2)| = 2S - 2i \}$

$H(j)$ = tallest stack of boxes with box j on top.

$H(j) = \max_{i<j, w_i > w_j} \{ H(i) \} + h_j$

Solution: $\max_j \{ H(j) \}$, j = which box will end on top.

Runtime = $\Theta(n^2)$

Reduction Examples

"unweighted shortest path" → "shortest weighted path with no neg edges"
↳ make all edges weight 1

"hamiltonian path" → "hamiltonian cycle"
↳ add node to graph and connect it to every other node.
cycle if path

Reductions

- Convert problem in into a problem we already know how to solve
- $A \to B$ "a reduces to B"
- So B is at least as hard as A

Interactive Proofs

Isomorphism is easy to prove but hard to disprove.
① Professor secretly and randomly selects a graph and permutes it
② Asks student which one he permuted. (Same as having iso)
Dif student get it right multiple times graphs are probably not iso morphic.

Zero-knowledge Proofs

- Proof that I solved a problem without actually revealing entire solution.

Recall 3-coloring post-it note recitation example.

"in NP" - solutions can be confirmed in poly time
"NP-hard" at least as hard as everything in NP.
"NP complete" - NP and in NP

True/False

T $F(n) + o(F(n)) = \Theta(F(n))$
T $F(n) = O(g(n)) \Rightarrow g(n) = \Omega(F(n))$
T $F(n) = O(g(n)) \wedge F(n) = \Omega(h(n)) \Rightarrow g(n) + h(n) = \Omega(F(n))$
T Under SUHA. Prob that 3 elts collide = $1/m^2$. (First can go anywhere)
T Radix sort can work using insertion sort vs. counting sort bc ↑ is stable
F Optimal knapsack problem will always contain objects with greatest value to cost ratio v_i/v_c. Fails b/c greedy doesn't always work.
T Every problem in NP can be solved in exponential time.
Hash table size m with open addressing. Pr that 3rd key needs exactly 3 probes $= 2/m(m-1)$
T There is a poly time alg for knapsack if all items have size $\{0,1\}$
T Dijkstra runs incorrectly with negative weight edges
F Any Functions f,g must be either $f = O(g)$ or $g = O(f)$
False since f,g could oscillate
T Dijkstra runs in $O(V^3)$ time
↳ True since its ↗ not Θ
F Can build BST from unsorted list in $O(n)$ time.
F since violates $\Omega(n \log n)$ lower bound on comparison sorts
T Can build heap in $O(n)$ time
T Dijkstra relaxes each edge at most once
T In worst case Merge Sort runs in $O(n^2)$ time. True sins $O(n \log n)$ is $O(n^2)$
T Merge-sort can be made stable - break ties with 'left'
F If load factor less than 1, there are no collisions.
T IF SAT $\leq_p A$, then A is NP-hard
T longest common subsequence can be solved using an algorithm for finding longest path in weighted DAG
T Restoring AVL invariant may take $\Theta(\lg n)$ rotations
T It is possible for a DFS on a directed graph with positive # edges to produce no tree edges.
T Max heap can support increase key and decrease key in $\Theta(\lg n)$
T DFS in an undirected graph never produces cross edges
T Radix sort works in linear time if elmts. are in range $\{1, \dots n^d\}$
T Can test if G is a tree in $O(V+E)$ time using DFS or BFS
T Bellman-Ford detects negative weight directed cycles in input graph.
T Topological Sort of dag can be computed in linear time. DFS
T Cannot detect neg-weight cycles in $O(V+E)$ time. Best BF $O(VE)$
T cannot build BST in $o(n \lg n)$ time

Problem 2. Storing Partial Maxima [30 points] (1 part)

6.006 student, Mike Velli, wants to build a website where the user can input a time interval in history, and the website will return the most exciting sports event that occurred during this interval. Formally, suppose that Mike has a chronologically sorted list of n sports events with associated integer "excitement factors" $e_1, \ldots, e_n$. You can assume for simplicity that n is a power of 2. A user's query will consist of a pair (i, j) with $1 \le i < j \le n$, and the site is supposed to return $\max(e_i, e_{i+1}, \ldots, e_j)$.

Mike wishes to minimize the amount of computation per query, since there will be a lot of traffic to the website. If he precomputes and stores $\max(e_i, \ldots, e_j)$ for every possible input (i, j), he can respond to user queries quickly, but he needs storage $\Omega(n^2)$ which is too much.

In order to reduce storage requirements, Mike is willing to allow a small amount of computation per query. He wants to store a cleverer selection of precomputed values than just $\max(e_i, \ldots, e_j)$ for every (i, j), so that for any user query, the server can retrieve two precomputed values and take the maximum of the two to return the final answer. Show that now only $O(n \log n)$ values need to be precomputed.

Solution: We are given the list $e_1, \ldots, e_n$. For each $1 \le i \le n/2$, store $\max(e_i, e_{i+1}, \ldots, e_{n/2})$, and for each $n/2 < j \le n$, store $\max(e_{n/2+1}, \ldots, e_j)$. Recurse separately on the two lists $e_1, \ldots, e_{n/2}$ and $e_{n/2+1}, \ldots, e_n$. Stop the recursion when the list size becomes 1.

If the user's query is (i, j) with $i \le n/2$ and $j > n/2$, then we can return $\max(\max(e_i, e_{i+1}, \ldots, e_{n/2}), \max(e_{n/2+1}, \ldots, e_j))$. If both $i, j \le n/2$ or $i, j > n/2$, then the answer is found recursively.

Let $S(n)$ be the number of values stored for a list of length n. By construction, $S(n) = O(n) + 2S(n/2)$, and therefore, $S(n) = O(n \log n)$.

Problem 6. Does this path make me look fat? [10 points]

Consider a connected weighted directed graph $G = (V, E, w)$. Define the ***fatness*** of a path P to be the maximum weight of any edge in P. Give an efficient algorithm that, given such a graph and two vertices $u, v \in V$, finds the minimum possible fatness of a path from u to v in G.

Solution: There are two good solutions to this problem.

We can see that that fatness must be the weight of one of the edges, so we sort all edge weights and perform a binary search. To test whether or not a path with a fatness no more than x exists, we perform a breadth-first search that only walks edges with weight less than or equal to x. If we reach v, such a path exists.

If such a path exists, we recurse on the lower part of the range we are searching, to see if a tighter fatness bound also applies. If such a path does not exist, we recurse on the upper part of the range we are searching, to relax that bound. When we find two neighboring values, one of which works and one of which doesn't, we have our answer. This takes $O((V + E) \lg E)$ time.

Another good solution is to modify Dijkstra's algorithm. We use "fatness" instead of the sum of edge weights to score paths, and the only change to Dijkstra itself that is necessary is to change the relaxation operation so that it compares the destination node's existing min-fatness with the max of the weight of the incoming edge (i, j) and the min-fatness of any path to i (the source of the incoming edge).

The correctness argument is almost precisely the same as that for Dijkstra's algorithm. A correct solution also had to note that negative-weight edges, which could be present here and normally break Dijkstra, don't do that here; adding negative numbers produces ever-more-negative path weights, but taking their max doesn't. This solution has the same time complexity as Dijkstra's algorithm.

Problem 8. I Am Locutus of Borg, You Will Respond To My Questions [24 points]

Upon arrival at the planet Vertex T, you and Ensign Treaps are captured by the Borg. They promptly throw the ensign out of the airlock. You had better solve their problem, lest you share his fate.

The Vertex T parking lot is an $n \times n$ matrix A. There are already a number of spaceships parked in the lot. For $0 \le i, j < n$, let $A[i][j] = 0$ if there is a ship occupying position (i, j), and 1 otherwise.

	0	1	2	3	4
0	0	1	1	1	0
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	0	0	0
4	1	1	1	0	1

The Borg want to find the ***largest square parking space*** in which to park the Borg Cube. That is, find the largest k such that there exists a $k \times k$ square in A containing all ones and no zeros. In the example figure, the solution is 3, as illustrated by the 3×3 highlighted box.

Describe an efficient algorithm that finds the size of the largest square parking space in A. Analyze the running time of your algorithm.

Hint: Call $A[0][0]$ be the *top-left* of the parking lot, and call $A[n-1][n-1]$ the *bottom-right*. Use dynamic programming, with the subproblem $S[i, j]$ being the side length of the largest square parking space whose bottom-right corner is at (i, j).

Solution: The base cases are the positions along the top and left sides of the parking lot. In each of these positions, you can fit a 1×1 square parking space if and only if the space is unoccupied. Thus, for the base cases ($i = 0$ or $j = 0$), we have $S[i, j] = A[i][j]$.

Now for the general case. Again, if $A[i][j] = 0$, then $S[i, j] = 0$, so we'll only consider the case where $A[i][j] = 1$. There is a parking space of size x with bottom-right corner at (i, j) if and only if there are three (overlapping) spaces of size $x - 1$ at each of the locations $(i - 1, j)$, $(i, j - 1)$, and $(i - 1, j - 1)$. Thus, the largest possible parking space is

$$S[i, j] = 1 + \min(\, S[i-1, j] \,,\, S[i, j-1] \,,\, S[i-1, j-1] \,)$$

The desired answer is $\max_{i,j} S[i, j]$, which requires $O(n^2)$ to compute.

There are n^2 subproblems, and each takes $O(1)$ time to solve, for a time of $O(n^2)$. This, added to the $O(n^2)$ to extract the desired answer, gives a total $O(n^2)$ running time, which is clearly optimal because all the data must be examined.

Problem 10. Guess Who? [10 points]

Woody the woodcutter will cut a given log of wood, at any place you choose, for a price equal to the length of the given log. Suppose you have a log of length L, marked to be cut in n different locations labeled $1, 2, \ldots, n$. For simplicity, let indices 0 and $n + 1$ denote the left and right endpoints of the original log of length L. Let d_i denote the distance of mark i from the left end of the log, and assume that $0 = d_0 < d_1 < d_2 < \ldots < d_n < d_{n+1} = L$. The ***wood-cutting problem*** is the problem of determining the sequence of cuts to the log that will cut the log at all the marked places and minimize your total payment. Give an efficient algorithm to solve this problem.

Solution: Dynamic programming. $c(i, j) = \min_{i<k<j}\{c(i, k) + c(k, j) + (d_j - d_i)\}$ where $c(i, j)$ is the min cost of cutting a log with left endpoint i and right endpoint j at all its marked locations. Start with $c(i, i+1)$ (consecutive cuts) and move outwards to $c(i, i+2)$, $c(i, i+3)$ until the maximal distance $c(1, n)$, which gives the optimal score for the whole wood. Remember pointers to k that gave max score at each step, and trace back pointers to construct optimal solution. Each iteration takes $O(n)$ (linear search between i and j) and there are $O(n^2)$ entries to fill (a triangle really, not a square). Greedy solutions that pick the maximum cut each time do not work. Similarly, heuristics like picking the point closest to the center do not work.

Problem 7. Indiana Jones and the Temple of Algorithms [10 points]

While exploring an ancient temple, Prof. Jones comes across a locked door. In front of this door are two pedestals, and n blocks each labeled with its positive integer weight. The sum of the weights of the blocks is W. In order to open the door, Prof. Jones needs to put every block on one of the two pedestals. However, if the difference in the sum of the weights of the blocks on each pedestal is too large, the door will not open.

Devise an algorithm that Prof. Jones can use to divide the blocks into two piles whose total weights are as close as possible. To avoid a boulder rolling quickly toward him, Prof. Jones needs your algorithm to run in pseudopolynomial time.

Solution: Just consider the smaller of the two piles. The goal is to make the weight of this pile as close to $W/2$ as possible, while not exceeding that weight. Our guess for each block is whether the block is included in this smaller pile or not.

Our subproblems are $P(i, w)$, which has the value "true" if it is possible to make a pile of weight $w \le W/2$ with some subset of blocks 1 through i, and the value "false" otherwise. Initially, let $P(0, 0)$ be true, and $P(0, w)$ be false for all values of $w > 0$.

Let the weight of block i be w_i. Use the following recurrence:

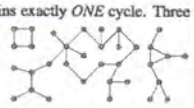
$$P(i, w) = P(i-1, w) \vee P(i-1, w - w_i)$$

(where $P(i-1, w-w_i)$ is considered to be false if $w_i > w$). If $P(i, w)$ is true, record in a separate table $B(i, w)$ which of $P(i-1, w)$ or $P(i-1, w-w_i)$ was true (choose arbitrarily if they are both true).

Find the largest value of w for which $P(n, w)$ is true. Then backtrack using the table B to determine which blocks were included in this pile to achieve this weight. This algorithm runs in $O(nW)$ time, because there are nW subproblems, each of which takes constant time.

Problem 9. Pseudotrees [20 points]

A *pseudotree* is defined as a connected, undirected graph that contains exactly *ONE* cycle. Three sample pseudotrees are shown below:



In the following questions, suppose $T = (V, E)$ is a pseudotree.

(a) [5 points]

Suppose you want to represent T in a way that minimizes the amount of space used. Is it more efficient to use an adjacency matrix, or adjacency lists? Explain.

Solution: Adjacency lists are more efficient. T has $|V|$ edges exactly (one edge per vertex). Consequently, while an adjacency matrix uses $O(|V|^2)$ bits to represent T, an adjacency list uses only $O(|V| \log |V|)$ bits to represent T, which is a substantial savings.

(b) [5 points]

Define C to be the list of all vertices that lie in the cycle in T. Give an $O(V)$ algorithm that returns a list $[u_1, \ldots, u_k]$ containing precisely the vertices in C. Prove that your algorithm is correct and has the desired running time.

Solution: Perform a DFS on T. Upon reaching a vertex that has already been visited, simply pop back up the stack and print all of the vertices while doing so. This prints precisely the vertices along the cycle in T, because those vertices are exactly those vertices encountered when relaxing a chain of edges leading to a vertex that has been visited before. The running time guarantee follows from the running time of DFS, and the fact that $O(V + E)$ is $O(V)$ in pseudotrees.

(c) [10 points]

Now, suppose that each edge in T is given a positive edge weight, and suppose you are given a source vertex $s \in V$ that is *NOT* on the cycle C. Give an $O(V)$ algorithm that computes the shortest path length $\delta(s, u)$ from s to all vertices u that lie on the cycle C. Prove that your algorithm is correct and has the desired running time. You may use part (b) as a subroutine.

Solution: Run the algorithm from part (b) and mark each vertex in the cycle. Then do a BFS or DFS from s until a vertex x on C is reached. Record the value of $\delta(s, x)$. Then do two searches around the edges in C, one proceeding in each direction around the cycle. In doing so, record the distance from x to each vertex $y \in C$ if one goes from x to y walking each of the two directions of the cycle. Then assign $\delta(x, y)$ to be the minimum of the two distances. Finally, compute $\delta(s, y) = \delta(s, x) + \delta(x, y)$ for each $y \in C$. This is correct since x must lie along a shortest path from s to any $y \in C$. The running time is clearly $O(V)$ as we run $O(1)$ graph searches in a graph with $O(V)$ edges.

Problem 13. The Longest Quiz Path in a Tree [15 points]

Let $T = (V, E)$ be an undirected tree (a connected graph with no cycles) with non-negative edge weights. Design and analyze an efficient algorithm to determine the length of the *longest path* in T, i.e., the value of

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ denotes the length of the shortest path from u to v. For full marks, an $O(V)$ algorithm is required.

Hint: Start by selecting an arbitrary vertex $r \in V$ to be the *root* of T.

Solution: First, select an arbitrary vertex $r \in V$ to be the *root* of T, and run a DFS or BFS to direct all edges outwards, so that we now have a rooted tree. For any given path P, there is a unique "highest" node, call it r, in this tree. (The path starts somewhere, goes up the tree, and then goes down.) In this case, we say that P is rooted at r.

Now, for each vertex v, define $P[v]$ to be the subproblem of finding the longest path from v to some leaf in the subtree rooted at v. We then have the recurrence:

$$P[v] = \max_{u} (P[u] + w_{u,v})$$

This is a DP which with $O(V)$ subproblems, where each subproblem takes $O(V)$ time to evaluate. However, notice that there is only one evaluation for each edge, and so the computation takes $O(E)$ time total. Since this is a tree, $|E| = |V| - 1$, and so $O(E) = O(V)$.

Next, we want to compute $L[v]$, the length of the longest path rooted at v. First, we claim that the longest path must go from a leaf to another leaf. If not, we could make a longer path by including another edge adjacent to the non-leaf.

Thus, to do this, for each child u, consider the quantity $P[u] + w_{u,v}$. Let u_1 and u_2 be the two children with the largest values of this. Then, we have

$$L[v] = P[u_1] + w_{u_1,v} + P[u_2] + w_{u_2,v}$$

Computing all the $L[v]$ values takes $O(V)$, by the same argument as for $P[v]$.

Finally, we have that the length of the longest path is simply

$$\max_{v \in V} L[v]$$

Optimizing Sorts

HC[0,j,k] = HC[i,j,0] = 0

HC[i,j,k] = max { HC[i-1,j,k], HC[i,j-1,k], HC[i-1,j-1,k-1] + hij } ∀ i,j,k

subproblems: O(n²q)
time per subproblem: O(1)
Runtime (HC): O(n²q)

Barbarian!

M[T,k] = min { pc + ME[RLA(c)] } ∀ c ∈ children of T

4) Do AVL search for right most node whose key is less than or equal to k-th root node of C.

Asymptotic Relationships

Small

$1/n$
$1/5$
$\log(\log(n))$
$\log(n)$
$n^{1/100}$
$10^{100}n$
$n\log n, \log(n!)$
$n^{100}, \binom{n}{100}$
$3^{\sqrt{n}}$
$2^n, 2^{n+1}$
$n2^n$
3^n
$4^n, 2^{2n}$
$n!$

Big

Explanations for the most common errors:

- $\log n = o(n^x)\ \forall x > 0$. Using L'Hôpital's Rule:

$$\lim_{n\to\infty} \frac{n^x}{\log n} = \lim_{n\to\infty} \frac{\frac{d}{dn}n^x}{\frac{d}{dn}\log n} = \lim_{n\to\infty} \frac{xn^{x-1}}{1/n} = \lim_{n\to\infty} xn^x - \infty$$

$\log(n^2) = 2\log(n)$

- $2^n = o(3^n)$:

$$\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$$

$\log(2^n) \Rightarrow \Theta(n)$

- By Stirling's approximation, $n! = \Theta(n^n)$, so $\log n! = \Theta(\log n^n) = \Theta(n \log n)$.
- $\binom{n}{100}$, or "n choose 100", can be expressed as

$\log_2 n = k$
$\Rightarrow 2^k = n$

$$\binom{n}{100} = \frac{n!}{100! \cdot (n-100)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-99)}{100!}$$

Now, observe $(n-100)^{100} \le n \cdot (n-1) \cdot \ldots \cdot (n-99) \le n^{100}$. Because $(n-100)^{100} = \Theta(n^{100})$, it follows that $n \cdot (n-1) \cdot \ldots \cdot (n-99) = \Theta(n^{100})$ and therefore $\binom{n}{100} = \Theta(n^{100})$.

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{ne}{k}\right)^k$$

$f(n) = [g(n)]$

O, Θ, Ω or $\asymp$	$f(n)$	$g(n)$
O	n^2	n^3
Ω	$n \lg n$	n
Θ	1	$2 + \sin n$
Ω	3^n	2^n
Θ	4^{n+1}	2^{2n+2}
O	$n \lg n$	$n^{1.0001n}$
Θ	$\lg\sqrt{10n}$	$\lg n^4$
O	$n!$	$(n+1)!$

Master Theorem

The master method provides a simple method for solving recurrences of the form

$T(n) = a(n/b) + f(n)$.

a= # of subproblems
(n/b) = size of each subproblem
f(n) = how long it takes to divide into subproblems and combine the results

where $a \ge 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The master method can be broken down into three cases depending on how the function $f(n)$ compares with the function $n^{\log_b a}$. The three cases can also be interpreted as different weightings of the recursion tree associated with the recursion. ① Calculate $n^{\log_b a}$ ② Compare with $f(n)$

1. **Case 1:** If $f(n) = \Theta(n^{\log_b a - \varepsilon})$, then the amount of work per level geometrically increases as we go down the tree. The work at the leaf level dominates and $T(n) = \Theta(n^{\log_b a})$.

2. **Case 2:** If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then the amount of work per level have about the same cost (i.e. work per level does not *polynomially* increase or decrease, though work per level still may increase or decrease at some slower rate). We have to take the work of all levels into account and $T(n) = \Theta(n^{\log_b a} \log^k n \log n) = \Theta(n^{\log_b a} \log^{k+1} n)$. $T(n) = \Theta(f(n)\log n)$

3. **Case 3:** If $f(n) = \Theta(n^{\log_b a + \varepsilon})$, then the amount of work per level geometrically decreases as we go down the tree. The work at the root level dominates and $T(n) = \Theta(f(n))$. Note: the recursion must additionaly satisfy the "regularity" condition: $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n.

To use the master method, simply determine which case (if any) of the master theorem applies, by comparing $f(n)$ with $n^{\log_b a}$. If the function $n^{\log_b a}$ is the larger, consider case 1. If $f(n)$ and $n^{\log_b a}$ are the same, consider case 2. And if $f(n)$ is the larger, consider case 3.

Change of Variables (Solving Recurrences)

For some particularly tricky recurrences, it may be necessary to combine the existing methods we've looked at with a method known as **change of variables**. The idea behind change of variables is to rewrite a tricky recurrence in terms of a new variable that is somehow related to the old variable, solve the new recurrence, then change back to the original variable.

Consider, for example, the following recurrence containing a square root:

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

None of our existing methods provide an easy way to solve this recurrence. To solve this we can perform a change of variables by defining a new variable m as $n = 2^m$. Substituting this in gives:

$$\begin{aligned} T(2^m) &= 2T(\sqrt{2^m}) + \Theta(\log 2^m) \\ &= 2T(2^{(m/2)}) + \Theta(m) \end{aligned}$$

This recursion still isn't in the standard form that we expect for Master Theorem, so we define a new recurrence, S as $S(m) = T(2^m) = T(n)$. This gives us:

$$\begin{aligned} S(m) = T(2^m) &= 2T(2^{(m/2)}) + \Theta(m) \\ &= 2S(m/2) + \Theta(m) \end{aligned} \qquad \begin{matrix} a=2 \\ b=2 \end{matrix}$$

Suddenly, we have a form we can work with using any of the previous methods we've learned. Using Master Theorem, for example, gives us $S(m) = m \log m$. Now we simply have to substitute back to make this meaningful in the context of T and n:

$$\begin{aligned} S(m) &= \Theta(m \log m) \\ T(n) &= \Theta(\log n \log\log n) \end{aligned}$$

Exercise 3 – Suppose you are given $T(\log_2 n) = 2T(\log_4 n) + \Theta(\log\log n)$). Compute a closed form for $T(x)$ using change of variables.

Notation

Recurrences: Merge Sort: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
Insertion Sort: $T(n) = T(n-1) + O(n) = O(n^2)$ 2D Peak finding: $T(n) = T(n/2) + \Theta(n) = \Theta(n)$

Data structure to find how many ints in range (a,b).
in range in $4k$
A: same preprocessing as counting sort. $c_i = C'(b) - C'(A)$

Sq mod p

Is 3 a square mod 4931?

Find int x: $x^2 \equiv 3 \pmod{4931}$

$3^{\frac{p-1}{2}} = \begin{cases} +1 & \text{a square} \\ -1 & \text{not a square} \end{cases}$

$3^{2465} \bmod 4931$

$3^4 = 9$ $3^{32} = 2581$ $3^{256} = 1620$
$3^8 = 81$ $3^{64} = 2402$ $3^{512} = 1363$
$3^{16} = 1135$ $3^{128} = 334$ $3^{1024} = 3207$
$3^{32} = 4022$ $3^{256} = 3074$ *All mod 4931

Multiply powers (have one prop)

$2465 = 100110100001$

$\equiv 3207 \cdot 3074 \cdot 334 \cdot 2581 \pmod{4931}$

$3^{2464} \equiv 1 \pmod{4931}$

$3^{\frac{2464}{2}} = 1$

$\Rightarrow 3$ is a sq.

Find roots of 3?

$3^{\frac{4930}{2}} \equiv 1 \pmod{4931}$

$3^{\frac{2(1233)+2}{2}} \equiv 1 \pmod{4931}$

$3^{2(1233)+1} \equiv 1 \pmod{4931}$

$3^{2(1233)+2} \equiv 3 \pmod{4931}$

$\left(3^{\frac{2468}{4}}\right)^2 \equiv 3$

Square root!

$1233 = 10011010001$

$1368 \cdot 324 \cdot 2402 \cdot 4022 \cdot 3 = 1291$

$\Rightarrow 1291^2 \equiv 3 \pmod{4931}$

$4931 - 1291 = 3640$, $3640^2 \equiv 3 \pmod{4931}$

roots = 1291, 3640

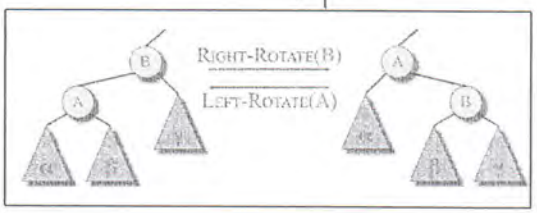
√ mod p

$a^{\frac{p-1}{2}} \equiv 1 \mod p$
$a^{\frac{3+4k-1}{2}} \equiv 1 \mod p$
$a^{1+2k} \equiv 1 \mod p$
$a^{2+2k} \equiv a \mod p$
$(a^{1+k})^2 \equiv a \mod p$
$a^{1+k} \equiv \sqrt{a} \mod p$

Verify BST

```
def verify(node):
    return helper(node, -∞, ∞)
def helper(node, lo, hi):
    if not(node.left is none or helper(node.left, lo, node.key)
        return False
    if not(node.right is None or helper(node.right, node.key, hi)
        return False
    if (node.key < lo or node.key > hi):
        return False
    else: return True.
```

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



GCD + Pulverizer

$$\begin{aligned} \gcd(259,70) &= \gcd(70,49) && \text{since } 259 \text{ rem } 70 = 49 \\ &= \gcd(49,21) && \text{since } 70 \text{ rem } 49 = 21 \\ &= \gcd(21,7) && \text{since } 49 \text{ rem } 21 = 7 \\ &= \gcd(7,0) && \text{since } 21 \text{ rem } 7 = 0 \\ &= 7. \end{aligned}$$

x	y	$(x \text{ rem } y)$	$=$	$x - q \cdot y$
259	70	49	$=$	$259 - 3 \cdot 70$
70	49	21	$=$	$70 - 1 \cdot 49$
			$=$	$70 - 1 \cdot (259 - 3 \cdot 70)$
			$=$	$-1 \cdot 259 + 4 \cdot 70$
49	21	7	$=$	$49 - 2 \cdot 21$
			$=$	$(259 - 3 \cdot 70) - 2 \cdot (-1 \cdot 259 + 4 \cdot 70)$
			$=$	$\boxed{3 \cdot 259 - 11 \cdot 70}$
21	7	0		

	Time complexity								Space complexity
	Average				Worst				Worst
	Find Min	Search	Insert	Delete	Find Min	Search	Insert	Delete	
Array	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
(Min/max) Heap	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$
BST tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

There are two kinds of binary heaps: max-heaps and min-heaps. Max-heaps have the property that $A[\texttt{Parent(i)}] \geq A[i]$, and min-heaps have the property that $A[\texttt{Parent(i)}] \leq A[i]$.

Max-heaps have the following operations: Heaps

`Max-heapify(A, i)`
Description: Assuming that binary trees rooted at the left and right children of i are max-heaps (but the tree rooted at i may not be), return A such that the binary tree rooted at i is a max-heap.
Approach: Compare the value $A[i]$ with its left and right children. If it is smaller than either of its children, exchange it with the larger of its two children. Continue comparing the value $A[i]$ with its left and right children and making exchanges with the larger of its children, until the value $A[i]$ is greater or equal to both of its children.
Analysis: Since $A[i]$ may need to be checked with each value in some branch of the tree, the runtime is $O(h) = O(lgn)$.

Heaps: parent(i) = $\lfloor i/2 \rfloor$, left(i) = $\lfloor 2i \rfloor$, Right(i) = $\lfloor 2i+1 \rfloor$

`Build-max-heap(A)`
Description: Given an array $A[1..n]$, build a max-heap.
Approach: Starting from the parent of the last leaf and traversing backwards in the array (traversing left and up in the binary heap), call `Max-heapify` on each index of the array.
Analysis: Using the recurrence $T(n) = 2T(n/2) + lgn$ to model the operation, the runtime is $O(n)$.

`Heapsort(A)`
Description: Given an unordered array $A[1..n]$, sort A using a max-heap.
Approach: Build a max-heap from A. Remove the maximum element by swapping it with the last leaf and placing it at the end of our sorted array. Calling `Max-heapify` on the root will result in a max-heap with the second largest element at the root. Continue the process of exchanging the root with the last leaf, removing the current maximum,placing it in our sorted array, and calling `Max-heapify` on the root.
Analysis: We make $n-1$ calls to `Max-heapify` (one call after each element in A that is visited), which has running time $O(lgn)$, so the total runtime of `Heapsort` is $O(n \lg n)$.

`Heap-maximum(A)`
Description: Return just the value of the maximum element of the max-heap A.
Approach: The maximum element is the root of the max-heap or $A[1]$.
Analysis: $O(1)$.

`Heap-extract-max(A)`
Description: Return the value of the maximum element of the max-heap and also remove the element from the max-heap.
Approach: The maximum element is the root of the max-heap. To remove it, we exchange it with the last leaf, remove the element, and `Max-heapify` the new root.

BST Operations

Binary Search Trees

Binary search trees have the following operations that allows users to search for elements and manipulate the tree.

`search(k)`
Description: Find key k. Return the node that contains k if it exists or NIL if it is not in the tree.
Approach: Starting at the root, check to see if the node we're at contains key k. If so, return that node. If not, traverse to the left child if k is smaller or traverse to the right child if k is larger and repeat. If we reach NIL before finding k, then k is not in the tree and we return NIL.
Analysis: Searching could potentially go along the longest branch of the tree, so the runtime of `search` is $O(h)$ where h is the height of the tree.

`insert(k)`
Description: Insert key k into the tree.
Approach: Starting at the root, compare the key of the node we're at to k. Traverse to the left child if k is smaller or traverse to the right child otherwise and repeat until we reach NIL. Once we reach NIL, replace the NIL node with a node containing k, thus inserting k into the tree.
Analysis: Searching where to insert k could potentially go along the longest branch of the tree, so the runtime of `insert` is $O(h)$ where h is the height of the tree.

`find-min(x)`
Description: Find the node with the minimum key of the tree rooted at node x and return it.
Approach: Starting at x, keep traversing to the left child until we reach a node with no left child. This node contains the minimum key, so we return it.
Analysis: The left-most branch could potentially be the longest branch of the tree, so the runtime of `find-min` is $O(h)$ where h is the height of the tree.

`next-larger(x)`
Description: Find the node that contains the next larger key relative to the key at node x.
Approach: If x has a right child, return find-min(x.right). Otherwise, traverse upwards through the tree in x's ancestry line until we reach a node with a larger key than x's key and return this node.
Analysis: We could potentially go through the longest branch of the tree up x's ancestry line or down x's right subtree so the runtime of `find-min` is $O(h)$ where h is the height of the tree.

`delete(x)`
Description: Remove node x from the tree while maintaining the tree's properties
Approach: If x has no children, just remove it by replacing x with NIL. If x has one child, splice out x by linking x's parent to x's child. If x has two children, splice out x's successor and replace x with x's successor.

Finding the next larger element: successor(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);
```

Radix Sort

- imagine each integer in base b
- $\Rightarrow d = \log_b k$ digits $\in \{0, 1, ..., b-1\}$
- sort (all n items) by least significant digit $\rightarrow$ can extract in $O(1)$ time.
- ...
- sort by most significant digit $\rightarrow$ can extract in $O(1)$ time.
- use counting sort for digit sort.
  - $\Theta(n+b)$ per digit
  - $\Theta((n+b)d) = \Theta((n+b)\log_b k)$ total time
  - minimized when $b = n$
  - $\Theta(n \log_n k)$
  - $O(nc)$ if $k \leq n^c$

Numerics: $\mathbb{Z}_n$: addition mod n $\phi(n) = |\mathbb{Z}_n^*|$
$\mathbb{Z}_n^*$: Multiplication. Set of # relatively prime to n

Chinese Remainder Theorem: Relatively prime
If $x \in \mathbb{Z}_n$ $n = p_1 p_2 \cdots p_k$
Then $\Rightarrow$ $x \leftrightarrow (x_1, x_2 \ldots x_k)$ where $x = x_i \pmod{p_i}$

Multiplication $O(k^2)$. 'shift and add' (k = # digits) } Polynomial in size of input
Division $O(k^2 \lg k)$ 'Binary search'

$\binom{n}{k}$ = c^n

Have you seen my integer?
Input: Sorted array N-1 distinct int in range $\{1 \ldots N\}$. So exactly one int $a \in \{1 \ldots N\}$ is not in A.
Goal: Find a? in $O(\log n)$
Solution: Binary search for missing value as follows: guess index i to be the location of missing num. If A[i+1]-A[i-1] = 2 return i. Otherwise, if A[i] = i, missing num must be > i. If A[i] = i+1 missing num < i. Perform recursion on appropriate half of array.

Pseudo-Code

d[v] = current calculated distance from s to v.

w(u,v) = Edge weights of (u,v)

π(v) = current parent of v that lead to the shortest path.

```
Relax(u,v):
    if d[v] > d[u] + w(u,v):        // Found a shorter path
        d[v] = d[u] + w(u,v)        // Update path
        π[v] = u                    // Update parent

Initialize (V, E, s):
    for v ∈ V:
        d[v] = ∞ , π(v) = null
    d[s] = 0, π[s] = s

Dijkstra (V,E,s):                   // use on weighted w/ non-neg edges
    Initialize (V,E,s)
    Q.push (all v∈V)                // Q is a min priority queue
    while Q not empty:
        u = Q.pop                   // u has smallest d[u] value
        for each neighbor v of u:   // For v still in Q
            relax (u,v)
    return d, π
    // Runtime: O(V (extract-min) + E (decrease-key))
    // Look through each edge once and relax each edge once
    // O(V log V + E) using Fibonacci Heaps

Bellman-Ford (V,E,s) // use on weighted w/ neg edges
    Initialize (V,E,s)
    for i=1 to |V|-1:               // Relax all edges |V|-1 times
        for each edge (u,v)∈E :
            relax (u,v)

    for each edge (u,v) ∈ E :       // check edges one more time
        if d[v] > d[u] + w(u,v)     // Shorter path Found
            return False            // Report negative cycle
    return d, π
    // Runtime: O(VE)
```

Bellman-Ford calculaes the shortest path from the source to every vertex in G.

```
BFS(G,s):                           // Will not necessarily find shortest path
    Q = queue()
    V = {} // Hashset of visited nodes
    V.add(s)
    Q.add(s)
    while Q not empty:
        t = Q.dequeue
        if t == goal: return t
        for all adjacent edges in t:
            v = adjacent vertex
            if v not in V:
                V.add(v)
                Q.add(v)
    return None
// Runtime O(V+E)

DFS: (G,s)
    S = stack()
    S.push(s)
    while S not empty
        v = S.pop()  // most recently added
        if v not discovered:
            v.discovered()
            for all neighbors of v:
                S.push(neighbor)
```

BFS/DFS.
Trees are dependent on start vertex (root) and order of visits.

Types
Tree: edge visits new vertex
Back: leads to ancestor
Forwards: " " decendent
cross: Links two branches

	BFS undirected	BFS Directed	DFS undirected	DFS Directed
Tree	✓	✓	✓	✓
Back	X	✓	✓	✓
Forward	X	X	X	✓
Cross	✓	✓	X	✓

Graph Theory

Chromatic #: min val of k for coloring problem

Basic Coloring Algorithm:
1) Order nodes by largest degree first
2) Order colors c_1, c_2, c_3
3) For all nodes Assign v_i the lowest legal color
will use at most max(degree)+1 colors

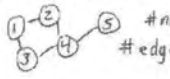
Handshake Lemma: $\sum_{v \in V} \deg(v) = 2|E|$

Definitions: Odd length cycle --> not bipartite
- Walk: Sequence of vertices in G
- Path: Walk with all different V's
- Cycle: Walk where $V_0 = V_k$ and $V_0, V_1 \ldots V_{k-1}$ are different.
- Connectivity: Vertices - if there is a path from v to v
  - Graph: Every pair of vertices is connected
  - ↳ indegree must = out degree + strongly connected
- Tree: Acyclic connected graph
- Spanning Tree: Subgraph (that is a tree) with the same set of vertices.
- A graph of n-nodes is a tree if:
  connected, no cycles and n-1 edges.
- Hamiltonian cycle: Cycle that visits each node exactly once.
- Euler Walk: Walk that traverses every edge once
- Euler Tour: Euler walks that ends at start node
  ↳ cant exists if a vertex has odd deg

Kraskals Algorithm: Find Minimum Spanning Tree (MST)
- start with an empty set
- add the minimum edge that does not create a cycle
- stop when there is no such edge.

Connectivity: Weak = connected w/ directions removed; Strong = normal

Graph representation:



#nodes = n
#edges = m

Adjacency matrix:

$$A = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 0 & 1 & 0 \end{array}$$

List Neighbor: O(V)
isNeighbor(u,v): Θ(1)
Space: $\Theta(n^2)$
A^k_{ij} = # of paths of length k from i ↔ j

Adjacency List (using Hash Set)
1: {2, 3}
2: {1, 4}
3: {1, 4}
4: {2, 3, 5}
5: {4}
* Amortized expected
isNeighbor(u,v): Θ(1)
Space: Θ(n+m)
List Neighbors: O(neighbors)

p = 1 - α probability that the first probe will work.
=> Expected # attempts 1/1-α

Collision Resolution

Chaining: linked list on each table entry

Open Addressing: Find another location

Uniform Hashing Assumption:
- Probe sequence of any key is equally likeley to be any of the m! permutations.

Linear Probing: simple but not very good
If collision, try next location until empty spot.
→ Creates clusters, slow things down

Double Hashing: Probe #
$h_m(k) = (h_1(k) + i \cdot h_2(k)) \bmod m$
So next probe will probably not be adjacent.
If $h_2(k) = 1 \ \forall k$ => Linear probing!

Rolling Hash Remember DNA Pset problem
Calculate a new hash based on previous hash w/o recomputing everything
↳ substring-match

Hashing

Universal Hashing: $h(k) = [ak + b \bmod p] \bmod m$
(p: large prime; a, b: random #'s $0 \ldots p-1$; m: size of table)

$\Pr[h(k_1) = h(k_2)] = 1/m \quad \forall k_1 \neq k_2$

Family H of hash functions: choose h randomly from H to minimize collisions

Expected Search Time:
- $O(1 + \#\text{keys that collide w/ k})$
- $O(1 + \frac{(n-1)}{m}) \Rightarrow \alpha = n/m$ (load Factor)
  All keys but k ↳ 1/m chance collision
- $O(1 + \alpha) = O(1)$ if $m = \Theta(n)$
  Need to resize table to make sure not rarely do this so O(1) amortized t:

DFS and BFS can be modified to find cycles.
BFS and DFS have the same time complexity O(V+E)
For Open Adressing, failed searches and insertion take the same # of comparisons
Bidirectional BFS performs asymptotically better than BFS in a highly branched graph. Ex, Balanced BST, Rubix Cube, Star graph
Bellman-Ford can still correctly calculate the shortest path from s to t when there are negative weight cycles if all paths from s to t contain no cycles.
DFS on an undirected graph has no forward or cross edges
BFS: O(V+E)
Use on unweighted graphs
Very Efficient on weighted graph with bounded positive integer weights.
Think about replacing an edge of weight x with x dummy nodes
Can work on a weighted directed tree graph (since there is only one unique path!)
Can test bipartiteness: Give alternating labels to vertecies. If at any step a vertex has (visited) neighbors with the same label as itself, return False.
No forward or back edges on an undirected graph
Bellman-Ford: O(VE)
Use on weighted graphs with negative edges/cycles
Invariant: After k iterations, d[v] will store the min weight cost to reach v in at most k hops
If there are no negative cycles. If there is a shortest path from s to to consisting of k edges, after the kth iteration, BF estimate of d[t] will be correct
Dijkstra's Algorithm: O((V+E)logV)
Use on directed graphs with no negative edges *
Very efficient on complete (all nodes connected) undirected graph with positive weights
Can still work with negative edges if all of them come from the source and no edges enter the source
Dijkstra = A* without heutistic
Dijkstra w/ AVL as priority queue has same running time.
Decrease-key and Extract-Min both O(logV) time.
Topological Sort and Relaxation: O(V+E)
Use on positive/arbitrarily weighted DAGs
Does not work on undirected graphs

special Case: No Neg weight edges ssse: DFS with Priority Queue expand node with least weight.
Special Case: Highly branched graphs. -> Bidirectional search.

Problem strategies:

- Shortest path with a detour v.
  Find shortest path from $s \to v$ and $v \to t$
- Add two edges to minimize shortest path.
  copy graph in 3 layers, connected by proposed edges.
  Run Dijkstra From s_1 to t_1, t_2, t_3 and compare.
  Keep track of traversed proposed edges.
- Multi Source Path problem. Find shortest path from a set of starting ptr.
  Solution 1: create node s^* that has a directed edge to every startpoint. Run dijkstra.
  Solution 2: reverse all edges and run dijkstra from t until some s is expanded.
- Dijkstra Modifications:
  Many hard problems can be solved with simple mods.
  Graph: Vertices edges and weight can have diff meaning
  Length: Different meaning Can be changed
  Priority Queue: Different comparison relation (min/max)
  Relaxation: Modified process
  Distance Initialization: different starting distances

Flight Agenda Problem:

Problem 13. The Longest Quiz Path in a Tree [15 points]

Let $T = (V, E)$ be an undirected tree (a connected graph with no cycles) with non-negative edge weights. Design and analyze an efficient algorithm to determine the length of the *longest path* in T, i.e., the value of

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ denotes the length of the shortest path from u to v. For full marks, an $O(V)$ algorithm is required.

Hint: Start by selecting an arbitrary vertex $r \in V$ to be the *root* of T.

Solution: First, select an arbitrary vertex $r \in V$ to be the *root* of T, and run a DFS or BFS to direct all edges outwards, so that we now have a rooted tree. For any given path P, there is a unique "highest" node, call it r, in this tree. (The path starts somewhere, goes up the tree, and then goes down.) In this case, we say that P is rooted at r.

Now, for each vertex v, define $P[v]$ to be the subproblem of finding the longest path from v to some leaf in the subtree rooted at v. We then have the recurrence:

$$P[v] = \max_{u} (P[u] + w_{u,v})$$

This is a DP which with $O(V)$ subproblems, where each subproblem takes $O(V)$ time to evaluate. However, notice that there is only one evaluation for each edge, and so the computation takes $O(E)$ time total. Since this is a tree, $|E| = |V| - 1$, and so $O(E) = O(V)$.

Next, we want to compute $L[v]$, the length of the longest path rooted at v. First, we claim that the longest path must go from a leaf to another leaf. If not, we could make a longer path by including another edge adjacent to the non-leaf.

Thus, to do this, for each child u, consider the quantity $P[u] + w_{u,v}$. Let u_1 and u_2 be the two children with the largest values of this. Then, we have

$$L[v] = P[u_1] + w_{u_1,v} + P[u_2] + w_{u_2,v}$$

Computing all the $L[v]$ values takes $O(V)$, by the same argument as for $P[v]$.

Finally, we have that the length of the longest path is simply

$$\max_{v} L[v]$$

Problem 7. Power to the Network [10 points] There are microwave towers along the 2451 miles of Route 66, allowing information to be transmitted from one end to another. New towers are added from time to time; when there are n towers we denote their positions by numerical values $x_1, x_2, \ldots, x_n$, where $x_i < x_{i+1}$ for $i = 1, 2, \ldots, n-1$. The power used to transmit a signal from one end of Route 66 to the other is proportional to

$$P = \sum_{i=1}^{n-1} (x_{i+1} - x_i)^2 .$$

In this problem, you must design a *Signal Tower Location* data structure that maintains the location of the towers and supports the following operations:

- `STL.insert(x)`: for a number x, add a new tower at position x into the Signal Tower Location structure STL.
- `STL.getpower()`: return the value of P, as calculated above, for all tower locations currently stored in STL.

Describe a data structure that supports `getpower()` in $O(1)$ time, and `insert(x)` in $O(\log(n))$ time, where n is the number of towers currently inserted into the Signal Tower Location structure. You may assume that no two identical values of x will ever be inserted.

Solution: Use an AVL tree, and augment it with a single integer value for P (in other words, a single number is added to the data structure; not one number per node.) Upon insdering a new tower at position x, recalculate the value of P and update it. Upon inserting x_i, we can update the value of P in $O(\log n)$ time by looking at the predecessor and successor of x_{i-1} and x_{i+1} of x_i in the AVL tree, and then adding $(x_i - x_{i-1})^2 + (x_{i+1} - x_i)^2$ and subtracting $(x_{i+1} - x_{i-1})^2$.

Problem 4. Changing Colors [15 points]

Consider a directed graph G where each edge $(u, v) \in E$ has both a weight $w(u, v)$ (not necessarily positive) as well as a color $color(u, v) \in \{\text{red}, \text{blue}\}$.

The weight of a path $p = v_1 \to v_2 \to \ldots \to v_k$ is equal to the sum of the weights of the edges, *plus* 5 for each pair of adjacent edges that are not the same color. That is, when traversing a path, it costs an *additional* 5 units to switch from one edge to another of a different color.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t, and analyze its running time. (You may assume that there exists such a path.) For full credit, your algorithm should have a running time of $O(VE)$, but partial credit will be awarded for slower solutions.

Solution: Construct a new graph G' as follows: for each vertex $v \in V$, create *two* vertices, a red vertex v_R and a blue vertex v_B, in G'. For each edge $(u, v) \in E$, if $color(u, v) =$ red, create (u_R, v_R) in G'. Otherwise, create (u_B, v_B). In either case, the new edge should have the same weight as the original. Finally, for each vertex $v \in V$, create the edges (v_R, v_B) and (v_B, v_R) in G', each with weight 5.

This works because all the red edges run between red vertices in G', and all the blue edges run between blue vertices. Switching colors requires traversing an edge in G' between a red and a blue vertex, incurring the extra cost of 5 units.

Now, run Bellman-Ford twice: once starting from s_R and once starting from s_B. Examine the paths ending at t_R and t_B. Determine which of the four paths ($s_R \rightsquigarrow t_R$, $s_R \rightsquigarrow t_B$, $s_B \rightsquigarrow t_R$, $s_B \rightsquigarrow t_B$) is shortest. Strip the subscripts to recover the desired path in G.

Constructing G' takes $O(V + E)$ time. G' has $2V$ vertices and $V + E$ edges. Running Bellman-Ford on G' takes $O(V'E') = O((2V)(V + E)) = O(VE)$ time. Thus the total running time is $O(VE)$.

Problem 9. Star Power! [20 points]

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which (along with making you temporarily invincible) lets you traverse the edge of your choice for free. In other words, you may change the weight of any one edge to zero.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t, given that you may set one edge weight to zero. Analyze your algorithm's running time. For full credit, your algorithm should have a running time of $O(E + V \lg V)$, but partial credit will be awarded for slower solutions.

Solution 1: Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t. Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \to v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \to v \rightsquigarrow t$ is the desired path.

The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra: $\Theta(E + V \lg V)$.

Ex: Graph with directed and undirected edges. assign each undirected edge a direction and create no cycles.
A: Topo sort directed edges For each undirected draw edge that goes in direction of lower sort to higher sort.

Sample h_1: 0 0 1 Is $H = \{h_1, h_2, h_3\}$ a universal family?
h_2: 1 0 0
1 2 3

A: No because $h_1(1) = h_1(2)$ regardless of the choice of i => elements 1 and 2 will collide with prob $> \frac{1}{3} = \frac{1}{2}$
Q: Suppose we modify h_1 so $h_1(2) = 1$ Is $H = \{h_1, h_2, h_3\}$ a universal family?
A: Yes, now elements 1 and 2 collide with $p_r = 1/2$, 2 and 3 w $p_r = 1/2$ and 1 and 3 $p_r = 0$

Dynamic Programming

1) Define Sub problems
2) Relate subproblem solutions w/ recurrence + base case
3) Recurse and memoize or bottom up DP table
4) Solve original problem
Analize runtime

Longest Increasing Subsequence

Input: $A_1 \dots A_n$
Goal: Find longest strictly increasing sequence (not necessarily contiguous)

$L(j)$ = LIS ending at position j

$L(j) = \max_{i<j} \{ L(i) \} + 1$
$A[i] < A[j]$

Solution = $\max_j \{ L(j) \}$

Runtime: $O(n^2)$

Edit Distance

Input: Strings $A[1..n]$ $B[1..m]$
Costs: C_i, C_d, C_R ← replacement, deletion, insertion
Goal: Min cost of transforming A → B

$T(i,j)$ = min cost to transform $A[1..i]$ into $B[1..j]$

$T(i,j) = \min \begin{cases} C_d + T(i-1, j) & \text{delete } a_i \\ T(i, j-1) + C_i & \text{transform } A \\ T(i-1, j-1) & : A[i] = B[j] \\ T(i-1, j-1) + C_R & : A[i] \neq B[j] \end{cases}$

$O(mn)$ #subproblems

Solution: $T(n, m)$

Balanced Partition ("Fairly dividing up gifts")

Input: n integers S
Goal: Partition integers into S_1 and S_2
Minimize $|\text{Sum}(S_1) - \text{Sum}(S_2)|$

$P(i,j) = \begin{cases} 1 & \text{if some subset of } \{A_1 \dots A_i\} \\ & \text{has sum } j \\ 0 & \text{otherwise} \end{cases}$ $\quad i = 0 \dots n$, $j = 0 \dots nk$

$P(i,j) = 1$ if $P(i-1, j) = 1$ or $P(i-1, j-A_i) = 1$
$= \max\{ P(i-1, j), P(i-1, j-A_i) \}$

Coin Change

Input: n denominations of coins. $1 = V_1 < V_2 < \dots < V_n$
Goal: make change for amt of money C. use as few coins.
$M(j)$: min # coins to make change for j

$M(j) = \min \{ M(j - V_i) \} + 1$

Runtime: $O(nC)$

Building Bridges

Goal: Build as many (non-crossing) bridges as possible.
$X(i)$: index of corresponding city on Northern bank.
(compute $X(i)$'s in $O(n \log n)$ time.

Find longest subsequence of $X(1) \dots X(n)$ $O(n^2)$

Integer Knapsack Problem

Input: n types of items, each with size S_i (int) and value V_i, knapsack capacity C
Goal: Pack items in knapsack maximize value. (repeated items ok!)

$M(j)$ = max val pack in knapsack $c=j$

For $j = 1$ to C # bottom up approach
compute $(M(j))$

$M(j) = \max \{ M(j - S_i) + V_i \}$ some item i in j slot
j^{th} slot empty or

Runtime: C subproblems, Fill knapsack of size $j - S_i$ optimally
n per subproblem $O(nC)$

Solution: Store a back pointer to $M(-)$ that was max.

$M(C)$ will be result. Following back pointers will give items used.

Box Stacking

Input: all orientations of original set of boxes each with depth d_i; h_i = height, $w_i \leq d_i$

Goal: Construct stack of max height can only stack box i on j if $w_i < w_j$ and $d_i < d_j$

$S = (\sum A_i)/2$
Find min val of S_i such that $P(n,i) = 1$

$\Rightarrow \min \{ S_i : P(n-i) = 1 \}$

$\text{sum}(S_1) = i$
$\text{sum}(S_2) = 2S_i$

Solution: $\min \{ S_i : P(n-1) = 1 \}$
$\Omega \min_{i \leq S} \{ S_i : \text{since } |\text{sum}(S_1) - \text{sum}(S_2)| = 2S - 2i \}$

$H(j)$ = tallest stack of boxes with box j on top.

$H(j) = \max_{i<j, w_i > w_j} \{ H(i) \} + h_j$

Solution: $\max_j \{ H(j) \}$, j = which box will end on top.

Runtime = $\Theta(n^2)$

Reduction Examples

"unweighted shortest path" → "shortest weighted path with no neg edges"
↳ make all edges weight 1

"hamiltonian path" → "hamiltonian cycle"
↳ add node to graph and connect it to every other node.
cycle if path

Reductions

- Convert problem in into a problem we already know how to solve
- $A \to B$ "a reduces to B"
- So B is at least as hard as A

Interactive Proofs

Isomorphism is easy to prove but hard to disprove.
① Professor secretly and randomly selects a graph and permutes it
② Asks student which one he permuted. (Same as having iso)
Dif student get it right multiple times graphs are probably not iso morphic.

Zero-knowledge Proofs

- Proof that I solved a problem without actually revealing entire solution.

Recall 3-coloring post-it note recitation example.

"in NP" - solutions can be confirmed in poly time
"NP-hard" at least as hard as everything in NP.
"NP complete" - NP and in NP

True/False

T $F(n) + o(F(n)) = \Theta(F(n))$
T $F(n) = O(g(n)) \Rightarrow g(n) = \Omega(F(n))$
T $F(n) = O(g(n)) \wedge F(n) = \Omega(h(n)) \Rightarrow g(n) + h(n) = \Omega(F(n))$
T Under SUHA. Prob that 3 elts collide = $1/m^2$. (First can go anywhere)
T Radix sort can work using insertion sort vs. counting sort bc ↑ is stable
F Optimal knapsack problem will always contain objects with greatest value to cost ratio v_i/v_c. Fails b/c greedy doesn't always work.
T Every problem in NP can be solved in exponential time.
Hash table size m with open addressing. Pr that 3rd key needs exactly 3 probes $= 2/m(m-1)$
T There is a poly time alg for knapsack if all items have size $\{0,1\}$
T Dijkstra runs incorrectly with negative weight edges
F Any Functions f,g must be either $f = O(g)$ or $g = O(f)$
False since f,g could oscillate
T Dijkstra runs in $O(V^3)$ time
↳ True since its ↗ not Θ
F Can build BST from unsorted list in $O(n)$ time.
F since violates $\Omega(n \log n)$ lower bound on comparison sorts
T Can build heap in $O(n)$ time
T Dijkstra relaxes each edge at most once
T In worst case Merge Sort runs in $O(n^2)$ time. True sins $O(n \log n)$ is $O(n^2)$
T Merge-sort can be made stable - break ties with 'left'
F If load factor less than 1, there are no collisions.
T IF SAT $\leq_p A$, then A is NP-hard
T longest common subsequence can be solved using an algorithm for finding longest path in weighted DAG
T Restoring AVL invariant may take $\Theta(\lg n)$ rotations
T It is possible for a DFS on a directed graph with positive # edges to produce no tree edges.
T Max heap can support increase key and decrease key in $\Theta(\lg n)$
T DFS in an undirected graph never produces cross edges
T Radix sort works in linear time if elmts. are in range $\{1, \dots n^d\}$
T Can test if G is a tree in $O(V+E)$ time using DFS or BFS
T Bellman-Ford detects negative weight directed cycles in input graph.
T Topological Sort of dag can be computed in linear time. DFS
T Cannot detect neg-weight cycles in $O(V+E)$ time. Best BF $O(VE)$
T cannot build BST in $o(n \lg n)$ time

Problem 2. Storing Partial Maxima [30 points] (1 part)

6.006 student, Mike Velli, wants to build a website where the user can input a time interval in history, and the website will return the most exciting sports event that occurred during this interval. Formally, suppose that Mike has a chronologically sorted list of n sports events with associated integer "excitement factors" $e_1, \ldots, e_n$. You can assume for simplicity that n is a power of 2. A user's query will consist of a pair (i, j) with $1 \le i < j \le n$, and the site is supposed to return $\max(e_i, e_{i+1}, \ldots, e_j)$.

Mike wishes to minimize the amount of computation per query, since there will be a lot of traffic to the website. If he precomputes and stores $\max(e_i, \ldots, e_j)$ for every possible input (i, j), he can respond to user queries quickly, but he needs storage $\Omega(n^2)$ which is too much.

In order to reduce storage requirements, Mike is willing to allow a small amount of computation per query. He wants to store a cleverer selection of precomputed values than just $\max(e_i, \ldots, e_j)$ for every (i, j), so that for any user query, the server can retrieve two precomputed values and take the maximum of the two to return the final answer. Show that now only $O(n \log n)$ values need to be precomputed.

Solution: We are given the list $e_1, \ldots, e_n$. For each $1 \le i \le n/2$, store $\max(e_i, e_{i+1}, \ldots, e_{n/2})$, and for each $n/2 < j \le n$, store $\max(e_{n/2+1}, \ldots, e_j)$. Recurse separately on the two lists $e_1, \ldots, e_{n/2}$ and $e_{n/2+1}, \ldots, e_n$. Stop the recursion when the list size becomes 1.

If the user's query is (i, j) with $i \le n/2$ and $j > n/2$, then we can return $\max(\max(e_i, e_{i+1}, \ldots, e_{n/2}), \max(e_{n/2+1}, \ldots, e_j))$. If both $i, j \le n/2$ or $i, j > n/2$, then the answer is found recursively.

Let $S(n)$ be the number of values stored for a list of length n. By construction, $S(n) = O(n) + 2S(n/2)$, and therefore, $S(n) = O(n \log n)$.

Problem 6. Does this path make me look fat? [10 points]

Consider a connected weighted directed graph $G = (V, E, w)$. Define the ***fatness*** of a path P to be the maximum weight of any edge in P. Give an efficient algorithm that, given such a graph and two vertices $u, v \in V$, finds the minimum possible fatness of a path from u to v in G.

Solution: There are two good solutions to this problem.

We can see that that fatness must be the weight of one of the edges, so we sort all edge weights and perform a binary search. To test whether or not a path with a fatness no more than x exists, we perform a breadth-first search that only walks edges with weight less than or equal to x. If we reach v, such a path exists.

If such a path exists, we recurse on the lower part of the range we are searching, to see if a tighter fatness bound also applies. If such a path does not exist, we recurse on the upper part of the range we are searching, to relax that bound. When we find two neighboring values, one of which works and one of which doesn't, we have our answer. This takes $O((V + E) \lg E)$ time.

Another good solution is to modify Dijkstra's algorithm. We use "fatness" instead of the sum of edge weights to score paths, and the only change to Dijkstra itself that is necessary is to change the relaxation operation so that it compares the destination node's existing min-fatness with the max of the weight of the incoming edge (i, j) and the min-fatness of any path to i (the source of the incoming edge).

The correctness argument is almost precisely the same as that for Dijkstra's algorithm. A correct solution also had to note that negative-weight edges, which could be present here and normally break Dijkstra, don't do that here; adding negative numbers produces ever-more-negative path weights, but taking their max doesn't. This solution has the same time complexity as Dijkstra's algorithm.

Problem 8. I Am Locutus of Borg, You Will Respond To My Questions [24 points]

Upon arrival at the planet Vertex T, you and Ensign Treaps are captured by the Borg. They promptly throw the ensign out of the airlock. You had better solve their problem, lest you share his fate.

The Vertex T parking lot is an $n \times n$ matrix A. There are already a number of spaceships parked in the lot. For $0 \le i, j < n$, let $A[i][j] = 0$ if there is a ship occupying position (i, j), and 1 otherwise.

	0	1	2	3	4
0	0	1	1	1	0
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	0	0	0
4	1	1	1	0	1

The Borg want to find the ***largest square parking space*** in which to park the Borg Cube. That is, find the largest k such that there exists a $k \times k$ square in A containing all ones and no zeros. In the example figure, the solution is 3, as illustrated by the 3×3 highlighted box.

Describe an efficient algorithm that finds the size of the largest square parking space in A. Analyze the running time of your algorithm.

Hint: Call $A[0][0]$ be the *top-left* of the parking lot, and call $A[n-1][n-1]$ the *bottom-right*. Use dynamic programming, with the subproblem $S[i, j]$ being the side length of the largest square parking space whose bottom-right corner is at (i, j).

Solution: The base cases are the positions along the top and left sides of the parking lot. In each of these positions, you can fit a 1×1 square parking space if and only if the space is unoccupied. Thus, for the base cases ($i = 0$ or $j = 0$), we have $S[i, j] = A[i][j]$.

Now for the general case. Again, if $A[i][j] = 0$, then $S[i, j] = 0$, so we'll only consider the case where $A[i][j] = 1$. There is a parking space of size x with bottom-right corner at (i, j) if and only if there are three (overlapping) spaces of size $x - 1$ at each of the locations $(i - 1, j)$, $(i, j - 1)$, and $(i - 1, j - 1)$. Thus, the largest possible parking space is

$$S[i, j] = 1 + \min(\, S[i-1, j] \,,\, S[i, j-1] \,,\, S[i-1, j-1] \,)$$

The desired answer is $\max_{i,j} S[i, j]$, which requires $O(n^2)$ to compute.

There are n^2 subproblems, and each takes $O(1)$ time to solve, for a time of $O(n^2)$. This, added to the $O(n^2)$ to extract the desired answer, gives a total $O(n^2)$ running time, which is clearly optimal because all the data must be examined.

Problem 10. Guess Who? [10 points]

Woody the woodcutter will cut a given log of wood, at any place you choose, for a price equal to the length of the given log. Suppose you have a log of length L, marked to be cut in n different locations labeled $1, 2, \ldots, n$. For simplicity, let indices 0 and $n + 1$ denote the left and right endpoints of the original log of length L. Let d_i denote the distance of mark i from the left end of the log, and assume that $0 = d_0 < d_1 < d_2 < \ldots < d_n < d_{n+1} = L$. The ***wood-cutting problem*** is the problem of determining the sequence of cuts to the log that will cut the log at all the marked places and minimize your total payment. Give an efficient algorithm to solve this problem.

Solution: Dynamic programming. $c(i, j) = \min_{i<k<j}\{c(i, k) + c(k, j) + (d_j - d_i)\}$ where $c(i, j)$ is the min cost of cutting a log with left endpoint i and right endpoint j at all its marked locations. Start with $c(i, i+1)$ (consecutive cuts) and move outwards to $c(i, i+2)$, $c(i, i+3)$ until the maximal distance $c(1, n)$, which gives the optimal score for the whole wood. Remember pointers to k that gave max score at each step, and trace back pointers to construct optimal solution. Each iteration takes $O(n)$ (linear search between i and j) and there are $O(n^2)$ entries to fill (a triangle really, not a square). Greedy solutions that pick the maximum cut each time do not work. Similarly, heuristics like picking the point closest to the center do not work.

Problem 7. Indiana Jones and the Temple of Algorithms [10 points]

While exploring an ancient temple, Prof. Jones comes across a locked door. In front of this door are two pedestals, and n blocks each labeled with its positive integer weight. The sum of the weights of the blocks is W. In order to open the door, Prof. Jones needs to put every block on one of the two pedestals. However, if the difference in the sum of the weights of the blocks on each pedestal is too large, the door will not open.

Devise an algorithm that Prof. Jones can use to divide the blocks into two piles whose total weights are as close as possible. To avoid a boulder rolling quickly toward him, Prof. Jones needs your algorithm to run in pseudopolynomial time.

Solution: Just consider the smaller of the two piles. The goal is to make the weight of this pile as close to $W/2$ as possible, while not exceeding that weight. Our guess for each block is whether the block is included in this smaller pile or not.

Our subproblems are $P(i, w)$, which has the value "true" if it is possible to make a pile of weight $w \le W/2$ with some subset of blocks 1 through i, and the value "false" otherwise. Initially, let $P(0, 0)$ be true, and $P(0, w)$ be false for all values of $w > 0$.

Let the weight of block i be w_i. Use the following recurrence:

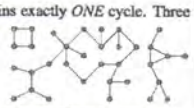
$$P(i, w) = P(i-1, w) \vee P(i-1, w - w_i)$$

(where $P(i-1, w-w_i)$ is considered to be false if $w_i > w$). If $P(i, w)$ is true, record in a separate table $B(i, w)$ which of $P(i-1, w)$ or $P(i-1, w-w_i)$ was true (choose arbitrarily if they are both true).

Find the largest value of w for which $P(n, w)$ is true. Then backtrack using the table B to determine which blocks were included in this pile to achieve this weight. This algorithm runs in $O(nW)$ time, because there are nW subproblems, each of which takes constant time.

Problem 9. Pseudotrees [20 points]

A *pseudotree* is defined as a connected, undirected graph that contains exactly *ONE* cycle. Three sample pseudotrees are shown below:



In the following questions, suppose $T = (V, E)$ is a pseudotree.

(a) [5 points]

Suppose you want to represent T in a way that minimizes the amount of space used. Is it more efficient to use an adjacency matrix, or adjacency lists? Explain.

Solution: Adjacency lists are more efficient. T has $|V|$ edges exactly (one edge per vertex). Consequently, while an adjacency matrix uses $O(|V|^2)$ bits to represent T, an adjacency list uses only $O(|V| \log |V|)$ bits to represent T, which is a substantial savings.

(b) [5 points]

Define C to be the list of all vertices that lie in the cycle in T. Give an $O(V)$ algorithm that returns a list $[u_1, \ldots, u_k]$ containing precisely the vertices in C. Prove that your algorithm is correct and has the desired running time.

Solution: Perform a DFS on T. Upon reaching a vertex that has already been visited, simply pop back up the stack and print all of the vertices while doing so. This prints precisely the vertices along the cycle in T, because those vertices are exactly those vertices encountered when relaxing a chain of edges leading to a vertex that has been visited before. The running time guarantee follows from the running time of DFS, and the fact that $O(V + E)$ is $O(V)$ in pseudotrees.

(c) [10 points]

Now, suppose that each edge in T is given a positive edge weight, and suppose you are given a source vertex $s \in V$ that is *NOT* on the cycle C. Give an $O(V)$ algorithm that computes the shortest path length $\delta(s, u)$ from s to all vertices u that lie on the cycle C. Prove that your algorithm is correct and has the desired running time. You may use part (b) as a subroutine.

Solution: Run the algorithm from part (b) and mark each vertex in the cycle. Then do a BFS or DFS from s until a vertex x on C is reached. Record the value of $\delta(s, x)$. Then do two searches around the edges in C, one proceeding in each direction around the cycle. In doing so, record the distance from x to each vertex $y \in C$ if one goes from x to y walking each of the two directions of the cycle. Then assign $\delta(x, y)$ to be the minimum of the two distances. Finally, compute $\delta(s, y) = \delta(s, x) + \delta(x, y)$ for each $y \in C$. This is correct since x must lie along a shortest path from s to any $y \in C$. The running time is clearly $O(V)$ as we run $O(1)$ graph searches in a graph with $O(V)$ edges.

Problem 13. The Longest Quiz Path in a Tree [15 points]

Let $T = (V, E)$ be an undirected tree (a connected graph with no cycles) with non-negative edge weights. Design and analyze an efficient algorithm to determine the length of the *longest path* in T, i.e., the value of

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ denotes the length of the shortest path from u to v. For full marks, an $O(V)$ algorithm is required.

Hint: Start by selecting an arbitrary vertex $r \in V$ to be the *root* of T.

Solution: First, select an arbitrary vertex $r \in V$ to be the *root* of T, and run a DFS or BFS to direct all edges outwards, so that we now have a rooted tree. For any given path P, there is a unique "highest" node, call it r, in this tree. (The path starts somewhere, goes up the tree, and then goes down.) In this case, we say that P is rooted at r.

Now, for each vertex v, define $P[v]$ to be the subproblem of finding the longest path from v to some leaf in the subtree rooted at v. We then have the recurrence:

$$P[v] = \max_{u} (P[u] + w_{u,v})$$

This is a DP which with $O(V)$ subproblems, where each subproblem takes $O(V)$ time to evaluate. However, notice that there is only one evaluation for each edge, and so the computation takes $O(E)$ time total. Since this is a tree, $|E| = |V| - 1$, and so $O(E) = O(V)$.

Next, we want to compute $L[v]$, the length of the longest path rooted at v. First, we claim that the longest path must go from a leaf to another leaf. If not, we could make a longer path by including another edge adjacent to the non-leaf.

Thus, to do this, for each child u, consider the quantity $P[u] + w_{u,v}$. Let u_1 and u_2 be the two children with the largest values of this. Then, we have

$$L[v] = P[u_1] + w_{u_1,v} + P[u_2] + w_{u_2,v}$$

Computing all the $L[v]$ values takes $O(V)$, by the same argument as for $P[v]$.

Finally, we have that the length of the longest path is simply

$$\max_{v \in V} L[v]$$

Optimizing Sorts

HC[0,j,k] = HC[i,j,0] = 0

HC[i,j,k] = max { HC[i-1,j,k], HC[i,j-1,k], HC[i-1,j-1,k-1] + hij } ∀ i,j,k

subproblems: O(n²q)
time per subproblem: O(1)
Runtime (HC): O(n²q)

Barbarian!

M[T,k] = min { pc + ME[RLA(c)] } ∀ c ∈ children of T

4) Do AVL search for right most node whose key is less than or equal to k-th root node of C.