6.006 Exam #1

Asymptotics: Tips - take logs to bring exponent down

$\log(ab) = \log(a) + \log(b)$

$\log(a^b) = b \log a$

$f(n) = \Omega(g(n))$ means $f(n)$ grows FASTER

$= O(g(n))$ means $f(n)$ grows SLOWER

$= \Theta(g(n))$ means same speed.

Ex $2^{\sqrt{\log n}}$ vs $n \log_{15} n$

$\log(2^{\sqrt{\log n}})$ vs $\log(n \log n)$

$\sqrt{\log n} \log(2)$ vs $\log(n \log n)$

$\log n + \log\log n$.

$2^{\sqrt{\log n}} = O(n \log_{15} n)$ ☐

Stirlings

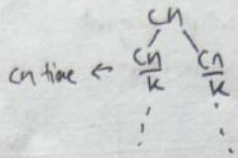
$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$\binom{n}{k} \Rightarrow \Theta\left(\frac{n^k}{k^k}\right)$

Note: $x^{k^n} = k^{(\log_k x)k^n}$

Recurrences

① Tree Method



cn time ← $h = \log_k n$

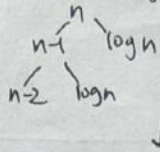
Total time $= n \log_k n$

② Master Thm

$T(n) = aT(n/b) + f(n)$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$. $\Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$, $\varepsilon > 0$ $\Rightarrow T(n) = \Theta(f(n))$

Ex $T(n) = T(n-1) + \Theta(\log n)$



$h = n$

Cost per level: $\log n$.

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

Lists: Append $O(1)$

Remove/search $O(n)$

Dicts: $O(1)$

pop(0) $= O(n)$

pop(end) $= O(1)$

Merge Sort

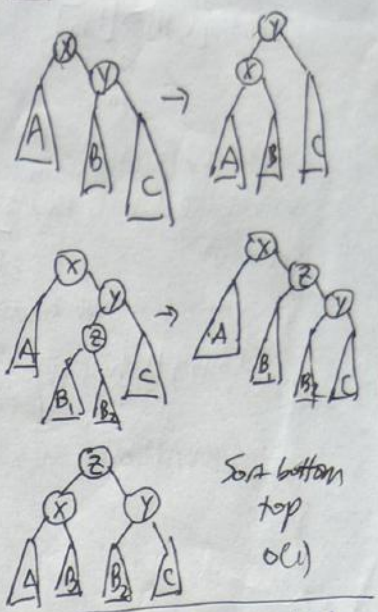
$n \log n$

elem — Layers, merge groups

Blocking $= \Theta(n)$

$\sum_{n=1}^{\infty} a(r)^n \rightarrow S = \frac{a}{1-r}$

AVL



Sort bottom top $O(1)$

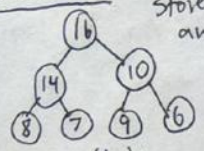
Decision Trees prove sorting must be $\Omega(\log n \cdot n)$.

- runtime lower bound = height of tree
- height = log (leaves)

possible solutions

HEAPSORT

Stored as an array



parent(i) = (i-1)//2

left(i) = 2i+1

right(i) = 2i+2

Check if is a heap: Check if each node is > children & < parent. $O(n)$.

Find(k): $O(n)$

Delete: $O(n)$

max(H): return H[0] — $O(1)$

max_heapify(H, i): — $O(\log n)$ — Moving top down

```
if H[left] > H[i] and H[left] > H[right]
    j = left
else:
    swap(H[i], H[j])
    max_heapify(H, j)
```

Insert(k): — Time $= O(\log n) + \Theta(1)$ — Bottom up

```
A[A.length] = k
i = A.length - 1
while i > 0 and A[parent(i)] < A[i]:
    swap A[i] w/ parent
```

Extract_Max: — $O(\log n)$

Swap(H[0], H[end])

max_heapify(H, 0)

swaps first elem w/ last, heap down

Build Max Heap: — $O(n)$

for i not leaf:

max_heapify(H, i)

- unsorted → Partially sorted array

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

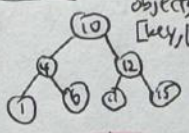
Heapsort: — $O(n \log n)$

Build Max_Heap

extract all maxes

BST/AVL

Stored as node objects w/ pointers [key, left, right, parent]



Insert(key): — Top Down — $\Theta(h)$

```
node = self
while T
    if key < node.key:
        node = node.left
    else:
        if node.right is None:
            node.right = Node
        node = node.right
```

find_min(self) — Top Down — $\Theta(h)$

```
node = self
while node.left
    node = node.left
```

delete(self): — $\Theta(h)$ bc find_min & self.right

```
node = self
if self.left & self.right:
    node = self.right.find_min
    self.key = node.key
```

prev/next(k): — $O(h)$

```
if right[k] ≠ None:
    return find_min(right[k])
else:
    if parent = None:
        return None
    elif x is left child
        parent = successor
    else:
        left_child = successor
```

Find: $\Theta(h)$

Delete: $\Theta(h)$

Augmentations

- Store info like tree height
- subtree # items
- subtree max/min

*REMEMBER HEAPS ARE BALANCED.

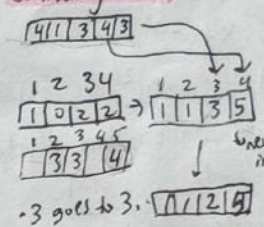
*AVL insertions take $O(\log n)$

*AVL height $= 1.44 \log(n)$

*Check if ~~an array~~ tree is sorted:

output as array and see if it's sorted $(O(n))$.

Counting Sort:



- 3 goes to 3.
- 4 goes to 5
- 3 goes to 2

(if k = O(n), $\Theta(n)$)

range

*make sure all elements ≤ len(A)

Radix Sort

Count sort digit by digit

d = length #s

n #s

k = # possible digits

$\Theta(d(n+k))$

Ex Radix sort → Make all numbers in a range [1, ...]

Employ counting sort - stable

d = 4

k = 10

↳ $\Theta(n)$, Count sort is also $O(n)$.

Merge Sort (n)

```
result = []
if len(n) == 1:
    return n
else:
    leng = int(len(n)/2)
    a = merge_sort(n[:leng])
    b = merge_sort(n[leng:])
    while len(a) > 0 or len(b) > 0:
        if both > 0:
            if a[0] > b[0]:
                result.append(b[0])
                b.pop(0)
            else:
                append a[0]
                pop a[0]
        else:
            return result
```

HASHING

$h(k): [1,N] \to [1,m]$ hashfunc = Division/Universal hashing

$\alpha = \frac{N}{m}$

*Make sure you check for key before placing into secondary structure.

SUHA: $Pr[h(k)=e] \le \frac{1}{m}$ – equally distributed

Universal: Probability that 2 keys hash to same place $\frac{1}{m}$

Collisions? $Exp[\#\text{collisions}] = \#\text{pairs} \cdot Pr[\text{collision}]$

Chaining: if hash function is good, probability $= \frac{N}{m}$

Rolling Hash: $h(k) = \sum_{i=0}^{m-1} key[i] \cdot \sigma^i \bmod p$ (length of key)

Open Addressing: $h(k,i) = (h'(k) + i) \bmod m$ (original key + i) % m Lin

Quad $(\text{original key} + i^2) \% m$

Double $h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$

Cuckoo Hashing: 2 Hash Tables.
Once key collides in both, start kicking out and moving keys

	Find	min max	next prev (from key)	insert(k)	delete(k) (find separate)	pop
DA Array	1	n	n	1	1	n
Dynamic Array	n	n	n	1	n	1
Balanced BST	lg(n)	lg n	lg n	lg n	lg n	lg n
Heaps	n	1	n	lg n	n	1
Sorted Array	lg(n)	1	lg(n)	n	n	n
Linked List	n	n	n	1	n	1
Hash Table	1 (avg)	n	n	1 (avg)	1 (avg)	1 (avg)

A.append(1) is O(1)

Ex Circular points from $\{-n^2, n^2\}$

Let $r(x,y)^2 = x^2 + y^2$.

Map points to radius^2. O(n)

Go through all radius values & return list w/ largest # of points O(n).

Ex – Max Area Triangle

Find max → For the given X, make an AVL Tree for y-coords and find_max, find_min (log n) → O(log(n))

Data Struct → Hash map maps point to AVL Tree w/ this X-coord
another table maps point to AVL Tree w/ this y-coord

Ex Proving a lower bound for a find/sort

*Use a decision tree

Find = log n

1) Determine # of leaves – what are we returning? A set? $\binom{n}{k}$ A number n
2) Determine height of tree from this → log(#leaves)
Lower bound

Sort = n log n

1) Determine # of possible permutations
2) Height = log(#leaves)
Lower bound

comparisons = log(#leaves) = log(perms)

usually permutations → n!

$n! \le \ell \le 2^h \to h \ge \log(n!)$
↳ leaves

So clearly $h = \Omega(n \log n)$ from Stirling's (front pg)

Ex Decision Tree

A[3] < A[4]?
NO: A[2] < A[3]?
YES: A[5] < A[6]?
YES: A[3]
YES

Ex Prob spot being empty $= \left(\frac{m-1}{m}\right)^n$

$Exp[\text{nonempty spots}] = \sum_{m}^{m} Pr[\text{nonempty}]$

$= m\left(1 - \left(\frac{m-1}{m}\right)^n\right)$

Ex Given array A of n integers and k, is there an A[i] equal to a previous element. O(n)

Hash table of size k – contain values of k prev entries.

1) check if H has elem
2) Remove A[i-k] O(1)
Add element A[i] O(1)

Ex Finding k smallest elements in a min heap →

→ Use min heap (2 of them)
→ original is min heap built on O(n)
→ add root of original min_heap to new heap.
→ Delete that root, add children of old root.
→ Go k times, k nodes in your new min heap
→ $k \log k + n$.

Ex # close pairs:
Hash objects into scattered buckets or buckets signifying items near a checkpoint (like the restaurant problem)

City of Cambridge → intersections are vertices, roads, undirected edges with weights w. For each intersection, find min distance to closest fire station

GPH

Use a source node, connect to each fire station, run Dijk from s.

Ex DP Largest value of contiguous subarray A[i:j] in O(n) time

$p(i)$ = largest value of any subarray ending in A[i]

$p(i) = A[i] + \max(0, p(i-1))$

Solve for $\max p(i)$, $i = 1:n$

GPH Given $G=(V,E)$, unweighted directed acyclic graph, add one edge to maximize size of largest strongly connected component in $O(V^3)$

- Assume initial acyclicness, there are no strongly connected components. Create a cycle from longest path in G.
- Use DFS to calculate reach(a,b) for all vertices a. → $V+E$ ✓
- Iterate over $s \to t$ through $k \to$ count # paths from $s \to t$. ✓ (V^3)
- Find $\max_{all\ t \to s}$.

Ex Text DP Works O:n

$p(i)$ = min badness sum for putting word i to n.

$p(i) = \min_{j=[i+1,n]} (\text{badness}(i,j) + p[j])$

...in exactly k lines

$p(i,k)$ = min badness for i to n, using k lines

$p(i,k) = \min_{j=i+1:n} [\text{badness}(i,j) + p(j,k-1)]$

Ex DP: Knapsack: Knapsack w/ volume S → fill w/ items S_i size + value V_i

$p(i,j)$ – largest value using first i items packable into volume j.

$p(i,j) = \max(p(i-1,j), V_i + p(i-1, j-S_i))$

Find $p(n,S)$

DP Longest Common Subsequence: A, B

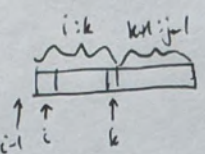
$p(i,j)$ – length of longest subsequence of prefixes A[:i], B[:j]

$$p(i,j) = \begin{cases} p(i-1,j-1)+1 & A[i]=B[j] \\ \max(p(i-1,j), p(i,j-1)) & A[i] \neq B[j] \end{cases}$$

DP $S = 21199112$

Break minimal strategy to 2119, 9112

$p(i,j)$ = maxscore attainable when it is adversary turn



$S = 112$ / 11 / 1 / $9+1+1=11$

$$p(i,j) = \min_{i-1 \le k \le j-1} \begin{cases} \max[A[i]+p(i+1,j), A[j]+p(i,j-1)] & \text{if } i-1=k \text{ (ie doesn't break it)} \\ \max[A[i]+p(i+1,k), A[k]+p(i,k-1), p(k+2,j)+A[k+1], p(k+1,j-1)+A[j]] & \text{ow} \end{cases}$$

Heap # nodes

Level contains k^i nodes

$$\text{nodes} = \sum_{i=0}^{h} k^i = k^h \sum_{i=0}^{h} \frac{1}{k^i} = \Theta(k^h)$$

so $\log_k$ [nodes] = height

Python complexity

append → O(1)

pop last → O(1)

pop other → O(k)

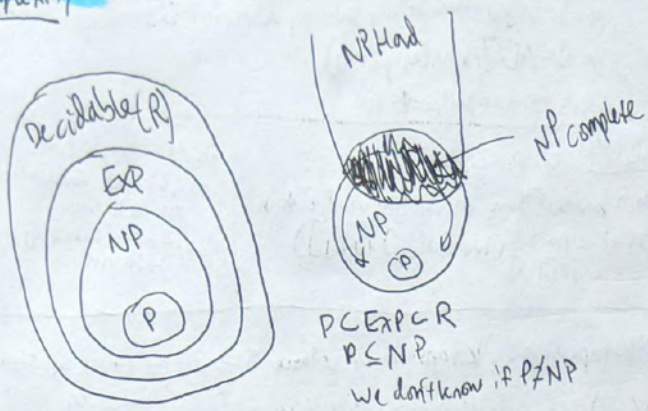
delete → O(n)

Ex Minimum Finding

- Divide and conquer: $k = \frac{n}{2}$
- Case 1: $A[k-1] < A[k]$ ← this half must contain local min (recurse here)
- Case 2: $A[k+1] < A[k]$ ← recurse on right side
- Case 3: $A[k-1] > A[k]$ and $A[k+1] > A[k]$: then $A[k]$ is the min

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

Complexity



$P \subset EXP \subset R$

$P \subseteq NP$

we don't know if $P \neq NP$

NP Hard: problems that are as hard as any problem in NP.
if $\exists$ NP complete Y s.t. $X \to Y$, $X \in NP\text{-H}$

NP complete: problems $\in$ NP, NP Hard. All x in which NP problem $Y \to X$.

Reduction $A \to B$: B is at least as hard as A

Kraft Inequality $\sum_{x_i \in X} D^{-\ell_i} \leq 1$

Huffmans: optimal code: build tree 0, 1's
- separate by probability,
- create tree branch of 2 least probable elements → can reduced bits necessary

Entropy $H(x) = -p \log_2 p - (1-p) \log_2 (1-p)$ Bernanill:

General $H_D(X) = -\sum_{x \in X} P(x) \log_D P(x)$ ⇒ D-ary alphabet

$H(X,Y) = H(X) + H(Y|X) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log_2 p(x,y)$

$H(Y|X) = -\sum_{x \in X} \sum_{y \in Y} p(x,y) \log p(y|x)$ → can reduce bits necessary

Slepian-Wolf Random Coding

Multiple dependent sources → bit rate of joint entropy (lower bit rate, the better, more compressed)

$W \to S \to X \xrightarrow{R} C \xrightarrow{\text{transmit}} W \to X$

S: information, X: codified, small msg, $(RS)^{-1}$

Cuckoo Hash: (2 tables)
- Fill in all spots keys can go in both tables
- 2 tables of size diff prime #s, insert attempt into larger first
- if both tables have collisions, kick a value in Table 1 to 2.
- if collision cycles ∃, resize.

Hashing, Probing

Linear/Quad:

Normally: key % m = slot

if conflict: linear: keep going down slots one by one

slot = (key % m + c · i) % m

quad: slot = (key % m + c · i^2) % m

if fail: resize

if $\frac{n}{m} \geq \frac{1}{2}$, resize

Double Hash

H_1 = key % table size

if conflict $H_2 = k - (\text{key} \bmod k)$ k: prime < table size

if H_2 is used, keep moving at an interval H_2 away

EXAM #2

Adjacency Lists

Matrices

	a	b	c	
a	0	1	1	dense
b	0	0	1	
c	0	1	0	

Graph Reps

Space	Adj Matrix	Linked List	Adj Array	AVL	Hash
Opt # edges →	$\Theta(n^2)$	$\Theta(n+m)$	$n+m$	$n+m$	$n+m$
Adding new edges	Dense	Sparse	Space	Sparse	Sparse
Ask edge we don't know is new	$O(1)$	$O(1)$	$O(1)$	$O(\log(\deg))$	$O(1)$
iter neighbors	$O(1)$	$O(\deg)$	$O(\deg)$	$O(\log(\deg))$	$O(1)$
iter sorted by node ing	$O(n)$	$O(\deg)$	$O(\deg)$	$O(\deg)$	$O(\deg)$
	$O(n)$	$O(\log(\log\deg))$ →		$O(\deg)$	$\deg(\log(\deg))$

DFS

$O(|V|+|E|)$

Traverse down until no children are detected on a new node, Go back and traverse next child until all children are checked.

- Tree Edge → explore new node
- Forward Edge → points to something that's already finished and visited after origin
- Back Edge → node goes back to an ancestor
- Cross → node goes back to previously node before origin

Ordering of finished nodes → top sort (DAGS)

BFS

Queue: Build up queue of unvisited nodes

$O(|V|+|E|)$

- Start at source
- Add all unvisited neighbors to queue. Mark them as visited

BFS Shortest Path

$i \leftarrow 0$

Find all neighbors of at least one vertex marked i.

If NONE: stop

mark all vertices found w/ $i+1$ if unmarked

$i \leftarrow i+1$

$\delta = i$

→ results in BFS tree, spanning tree

DAG-SP

$\Theta(n+m)$

1. top sort vertices
2. Init $\forall v.\delta = \infty$
   $v.\text{parent} = \text{None}$
   $\text{source}.\delta = 0$
3. for v in topsort:
   for $u \in G.\text{Adj}[v]$
   Relax (u,v,w)

Relax

if $\delta(s,v) > \delta(s,u) + w(u,v)$:
$\delta(s,v) = \delta(s,u) + w(u,v)$
$v.\text{parent} = u$

Johnsons

$V^2 \log V + VE$

init G' as $G \cup \{\text{new node}\}$
$G'.E = G.E \cup \{(s,v), v \in V\}$
$w(s,v) = 0$

For $v \in V$:
$h(v) = \text{value } \delta(s,v) \text{ found in BF}$

For $(u,v) \in G'.E$:
$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$

For each u in $G.V$:
Dijkstra $\hat{\delta}(u,v)$ for all v in V
For v in V:
$d_{uv} = \hat{\delta}(u,v) + h(v) - h(u)$
return D

What search should I use? for SSSP

Unweighted (w/ or w/o cycles): BFS; $O(n+m)$

weighted DAG (pos or neg): Topsort + DFS + relax; $O(n+m)$

positive weights w/ cycles: Dijkstra's $V \log V + E$

directed weighted cycles w/ negative weights: BFord $O(nm)$ — checks for neg. cycles

Undirected graph; weighted: Dijkstra's on directed graph w/ edges in both direcs.

ASSP

FW n^3 (Dense); Johnsons $V^2 \log V + VE$ (sparse)

positive integer weights → intermediate nodes (unweighted graph) BFS

Dijkstras

$O(T_{\text{construct } Q} + T_e \cdot n + T_d \cdot m)$

(T_e: extract min; T_d: decrease key)

Fib heap is always $O(V \log V + E)$

Init $\forall s$ as ∞
$\delta(s) = 0$
$P(v) = \text{None}$
$Q = \text{PriorityQueue.init}(\delta)$
While Q:
$u = Q.\text{extract\_min}()$
for v in $\text{adj}[u]$:
if $\delta[u] + w(u,v) < \delta[v]$
$\delta[v] = \delta[u] + w(u,v)$
$Q.\text{decrease\_key}(v, \delta[v])$
$P[v] = u$

- Q hash
  $O(T_c) = n$
  $O(T_e \cdot n) = n^2$ dense
  $O(T_d \cdot m) = m$
- Q. bin heap
  $O(T_c) = O(n)$ sparse
  $O(T_e \cdot n) = n \log n$
  $O(T_d \cdot m) = m \log n$

* Calling path can be $O(|V|)$
* To return unmarked edges, compare $w(u,v)$ w/ $\delta(u,v)$ $O(E)$
* Shortest Path → decrease one edge for each vertex, Run Dijkstras $O(V(V \log V + E))$ {return (δ, P)}

Undirected unweighted - iterate over (u,v); if $\delta(s,u)+1 = \delta(s,v)$, then not unique. ← Also how you do UNIQUENESS

Floyd Washall

$\delta(s,t) = w(s,t)\ \forall (s \to x) \in E$

for k in range(n):
for s in range(n):
for t in range(n):
$\delta[s,t] = \min\{\delta[s,t], \delta[s,k] + [k,t]\}$

Recursive: $fw(s,t,k) = $ shortest path w/ $[1 \ldots k]$

n^3 subproblems
$O(1)$ per

$$= \min \begin{cases} fw(s,t,k-1) \\ fw(s,k,k-1) + fw(k,t,k-1) \\ 0 \text{ if } s = t \\ w(s,t) \text{ if } k = 0 \text{ and } s \neq t \end{cases}$$

Dynamic Programming

Articulation Nodes?

BFS recursive search

Is x in a shortest path?

Thinking about $\delta(s,x) + \delta(x,t) = \delta(s,t)$?

From FW if

is A in your shortest path?

$d[s,a] + d[a,t] = d[s,t]$?

Bellman Ford: $O(|V||E|)$

$D(s)=0, D(v)=\infty$

For $i=1$ to $|V|-1$: (relax order can be set (or not))
  For (u,v) in E:
    relax(u,v,w)

For (u,v) in E:
  if $D(v) > D(u)+w(u,v)$: — check negative cycles
    return False

return T/D

Longest path

for general, it's NP, but...

positive DAG – turn all into negative weights and DAG-SP

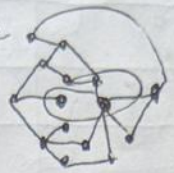
unweighted acyclic directed – set all edge weights to -1 and run DAG shortest path

~~the BFS thing~~

Example: Taking out an edge and finding max weight for it to be shortest path.

*Redefine graph as w/o edge

*Break down paths into weights w/o the edge



G', (u,v) removed

Consider $\delta'(s,u)$ and $\delta'(v,t)$

In this case $\delta'(s,t) \geq \delta'(s,u)+(u,v)+(\delta'(v,t))$

So call Dijkstra's 3 times

$O(V \log V+E)+O(V+E)$

↙ Graph construction

DYNAMIC PROGRAMMING

day $0 \to n-1$

Foo F

$c(f)=$ cal

$T(i, d_2, d_1) =$ max calories from ith day having to n^{th} day after d_2, d_1 → Time subproblem

$n|F|^2$ problems

Relation: $T(i, d_2, d_1) = \max\left[c(f)+T(i+1, d_1, f)\right]$ where $d_2, d_1, f \leq c_{max}$

Solve for $T(0, None, None)$

$|F|$ = time subproblem

nF^2 = number subproblems

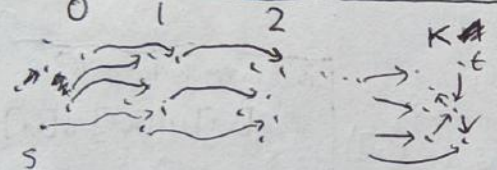
① Subproblem
② Relation
③ DAG – acyclic, Top sort – Base Cases
④ Solution ⑤ Algo – bottom up or memoize

Runtime = #problems · work/problem

1. Choose subproblem $p(\ )$ to be _____.
2. We can compute subproblems in terms of others using the following relation:
3. Subproblems depend on subproblems w/ smaller ___ so dependencies are acyclic.
4. Base Case: when ___, ___.
5. Solve (bottom up/top down) w/ (increasing/decreasing) ___.
6. There are # subproblems, each requiring ___ time.
7. The answer to the problem is $p(\ )$, optimal added time.

EXAMPLE: Graph Duplication, other Graph Problems

Ex. Shortest path b/w s,t that uses at least K edges:



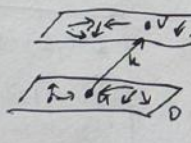
Find path from $(s,0) \to (t,k)$

$K|V|$ and $K|E|$ edges

$O(KV \log V + KE)$

*Create graphs for diff states

wb shortest path using edge k?



$s_0 \to t_1$

Duplicate graphs

wb at most K edges?

- BF K iterations, using only values from previous iteration $O(k \cdot E)$
- Same $k+1$ graphs and k copies of edges. Run Dijkstra's or DAG-SP and return $\min(\delta(s,t_0), \delta(s,t_1)\ldots)$

Second shortest paths

if weighted DAG – keep track of $\delta(u,v)$ and $\alpha(u,v)$ using top. DFS.

$\alpha(u,v) =$ 2nd min of $\left[\delta(s,u)+w(u,v),\ \alpha(s,u)+w(u,v)\right]$ over neighbors $\to v$

– otherwise... brute force w/ Dijkstras, remove each edge on shortest path, run Dijk again.

DP Problems:

Largest subsection: Take each i,j subarray, run DP n^2

if $O(n)$ needed: $p(i)$ – largest subsection that ends on $A[i]$ or starts on $A[i]$

Taking turns: if other person has optimal strategy:

max score achieved on your turn: $p(i,j) = \max(q(i+1,j)+A[i],\ q(i,j-1)+A[j])$

max score achieved not on your turn: $q(i,j) = \min(p(i+1,j)+A[i],\ p(i,j-1)+A[j])$

if not:

$$p(i,j) = \max \begin{cases} p(i+2,j)+A[i] & \text{if } A[i+1] > A[j] \\ p(i+1,j-1)+A[i] & \text{if } A[j] > A[i+1] \\ p(i+1,j-1)+A[j] & \text{if } A[i] > A[j-1] \\ p(i,j-2)+A[j] & \text{if } A[j-1] > A[i] \end{cases}$$

↑ Take into acct the in next move

Making a certain sum out of set ints?

$P(i,w)$ = True if $\exists$ some combo

ints 1 to I ↑ sum w Yes

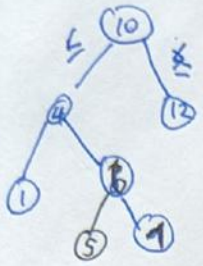
~~if p(i-1,w)~~ if $p(i-1,w)$ is T, $p(i,w)$ is T.

$p(i,w) = p(i+1,w) \vee p(i-1, w-w_i)$

↙ Sum – integer i.

BST

Stored as node objects w/ pointer



≤ right ≥ parent

left ≤ parent

→ Defined as node objects [key, left, right, parent]

OPERATIONS

h = height

→ ~~insert(self, key)~~ root

insert(self, key): $\Theta(h)$

top down

insert a key into the tree. start by comparing w/ root

```
if self.key=None
    return
node=self
while T:
    if key<node.key
        if node.left is None:
            node.left=Node[key, node, None, None]; return
        node=node.left
    else:
        if node.right is None:
            node.right=Node[key, node, None, None]
            return
        node=node.right
```

→ find_min (self)

$\Theta(h)$ - Think max case

find the min elem in tree

```
if self.key=None
    return None
node=self
while node.left
    node=node.left
return node
```

→ def delete (self) $\Theta(h)$ delete a given node

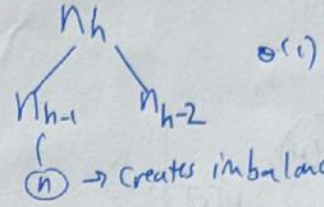
- Depending on pointers, reconfig pointers to maintain BST invariant

```
node=self
if self.left and self.right:
    node=self.right.find_min
    self.key=node.key
```

→ prev/next (k) $\Theta(h)$ be find_min or iterating up

```
If right[k]≠none:
    return find_min(right[x])
else: if x has no parent: no successor
    else if x is a left child
        parent=successor
if x is right child of a left child, then
        left child=sucessor
```

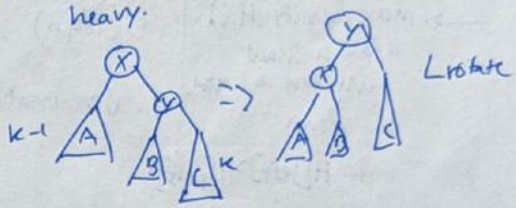
AVL Trees



$\Theta(1)$

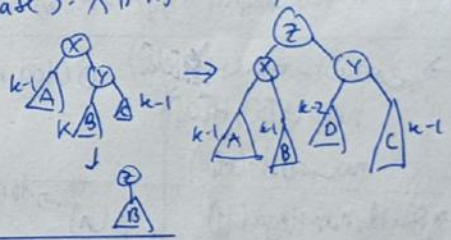
→ Creates imbalance

Case 1: X is right heavy, Y is right heavy.



Case 2: similar

Case 3: X is right and y is left heavy



Heapsort - stored as an array

- Ideas B to insert elements into a heap tree to sort

parent(i) = $\lfloor \frac{i-1}{2} \rfloor$

left(i) = 2i+1

right(i) = 2i+2

} 0 index

OPERATIONS

→ max(H):
- return H[0]

O(1) returns the max

→ max-heapify(H,i)
take a element anywhere and move down
if H[left] > H[right]
j = left
else: j = right
if H[j] > H[i]:
swap(H[i], H[j])
max_heapify(H, j)

O(log n)

i = current pt

→ Take larger child

→ if child > parent, swap

takes an arbitrary element and moves it down the heap to correct place

* top down

→ extract_max(H)
swap(H[0], H[end])
max_heapify(H, 0)

O(log n)

extracts the first element and swaps w/ last element, bring down

---

→ Build maxHeap(H)
for i from $\lfloor \frac{n}{2} \rfloor$ to 0:
max_heapify(H, i)

O(n) → for loop

- organize unsorted into sorted array

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

↓ Split into 2 subtrees

↓ cost per tree

→ Heapsort(H):
Build_Max_Heap(H)
fo r i = n-1 to 1:
extractmax(H, i)

O(n log n)

~~Build~~ Build heap, but then return 100% sorted array

→ Insert (k):
A[A.length] = k
i = A.length - 1
while i > 1 and A[parent(i)] < A[i]:
swap A[i] w/ parent

Time = $\theta(1) + O(\log n)$

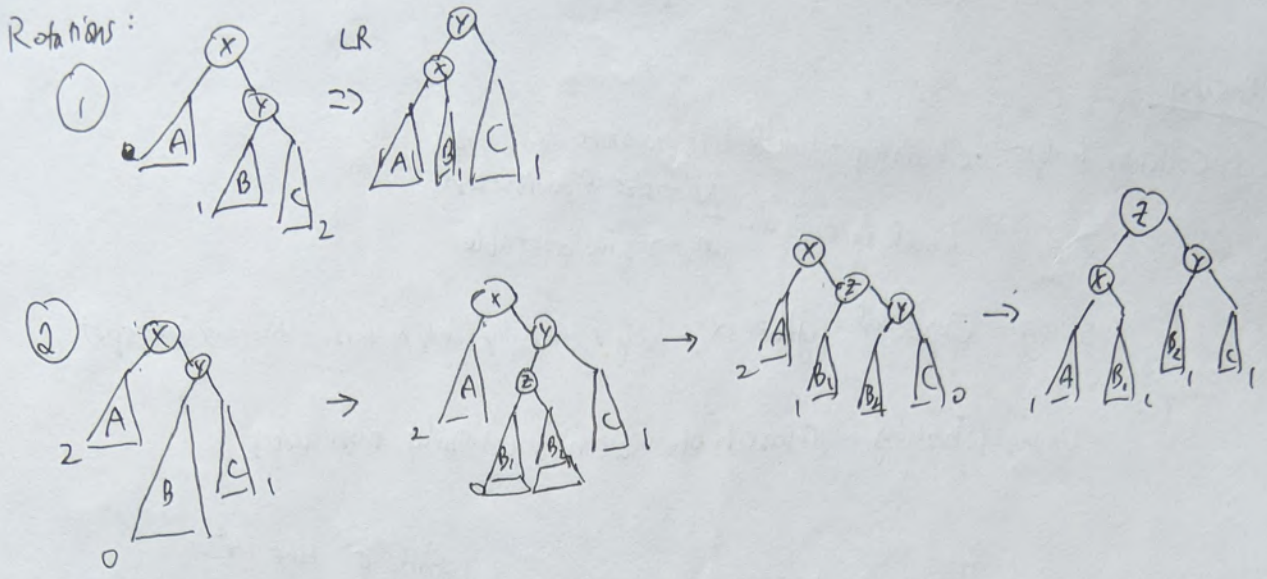
↓ Adding elem

- Reverse max-heapify
- Bottom up

AVL Trees - Balanced BST trees

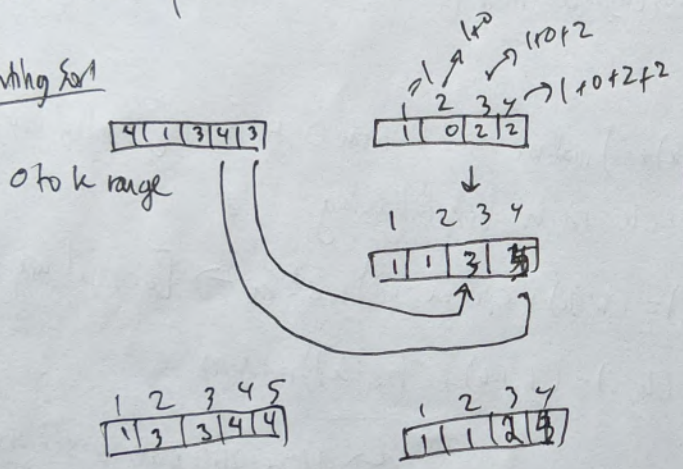
Invariant: for every node x, height of left/right child differ by at most 1

Rotations:



Decision Trees prove sorting must be $\Omega(\log n \cdot n)$

Counting Sort



if $k = O(n)$, $\Theta(n)$

Stable sorting

- Counting sort
- Insertion sort
- Merge sort

What is not stable?

HEAPSORT

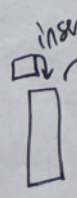
Radix sort: Sort digit by digit (Counting Sort)

~~[illegible]~~, d = length numbers

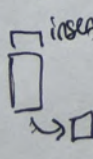
$\Theta(d(n+k))$ n numbers

k = # ~~possible~~ possible digits

Data Structs

Stack = 

Linked list = (key, pointer to next, prev)

Queue = 

Problem - insert and array allocation

Fix - dynamic array - Allocate more than you need

Hashing

Collision help - • chaining - linked list to store collisions

Load factor $\alpha = \frac{N}{m}$ → size of possible keys / → size of table

SUHA - Expected length = α, key is equally likely to hash to any one spot

Universal hashing - choose hash function randomly from keys

Rolling Hash

$$h(key) = \sum_{i=0}^{m-1} key[i] \cdot \sigma^i \bmod p$$

↗ literally σ^i like 4^2

Open Addressing:

Linear probe $h(k,i) = (h'(k)+i) \bmod m$, for i from 0 to m-1 [original key + i] %. m

Check each box following #

Quadratic Probing $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ → [original key + i^2] %. m

Double hashing: $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

↳ Move this many positions from the original collision location

Asymptotic Behavior:

$(c)^n$ faster than n^c

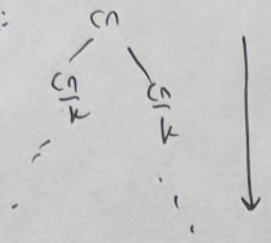
n^c faster than log

Note $X^{k^n} = k^{(\log_k X)k^n}$

- $f(n) = O(g(n))$ means $g(n)$ is faster than $f(n)$
- $f(n) = \Omega(g(n))$ means $g(n)$ is slow than $f(n)$

Recurrences

① Tree method:



$h = \log_k n$ → cn time

→ cn time

We say total time = $n \log_k n$

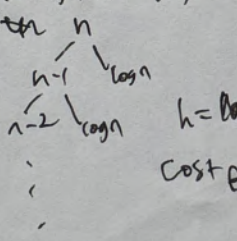
② Master Thm: $T(n) = aT(n/b) + f(n)$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ $\varepsilon > 0 \Rightarrow T(n) = \Theta(f(n))$

Ex. $T(n) = T(n-1) + \Theta(\log n)$



$h = n$

cost per level = $\log n$.

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

Python costs:

Lists: Append $O(1)$

Remove/search for elem = $O(n)$

Dicts all $O(1)$

Linear Probing / Quadratic

- Hash by key %. m = slot
  - if conflict →

$$slot = (key \,\%. m + c \cdot i) \,\%. m \quad i \in 0, 1, 2 \ldots m$$

or

$$slot = (key \,\%. m + c \cdot i^2) \,\%. m \quad i \ldots$$

  - if this fails — resize

- resize if $\frac{n}{m} \geq \frac{1}{2}$

Double Hashing

H_1 = key %. table size

if collision → H_2 = $k - (key \bmod k)$ k = prime < table size

if H_2 is used, then move H_2 down from H_1

Cuckoo Hashing

- Fill in all spots keys can go — in both tables
  - 2 tables of size diff prime #s — insert attempt into larger first
  - if both tables have collisions, kick a value in Table 1 into Table 2.
  - if collision cycle occurs — resize