1)

Peak Finding

One Dimension

array

a	b	c	d	e
1	2	3	4	5

a-i are #'s.

Position α is a peak if $\alpha \geq \alpha-1$ and $\alpha \geq \alpha+1$

Basic algo goes through 1-by-1 $\Theta(n)$

Better: middle -> compare either side -> go to middle of side that's higher

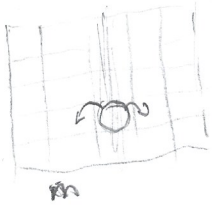
↳ keep splitting array in 2. you can do this (worst case) $\log_2(n)$ times

↳ each of these $\log_2(n)$ are of length 1 i.e $\Theta(1)$

$\Rightarrow \Theta(\log_2(n))$

2 Dimensional

could go through each one ... slow $\Theta(mn)$



finding global max

Better: pick middle column $j = m/2$, find global max on column j, say at (i,j)

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

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

else (i,j) is a peak.

↳ if not peak you now have same problem w/ half as many rows.

↳ get down to base case of one column, find global max, done.

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

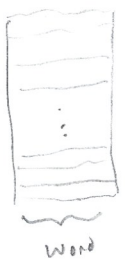
$T(n,1) = \Theta(n)$

$\Rightarrow T(n,m) = \underbrace{\Theta(n) + \dots + \Theta(n)}_{\log m} \simeq \Theta(n \log_2(m))$

2)

① Random Access Machine

- Rand. Access Memory (RAM)
  ↳ modelled by big array
- in $\Theta(1)$ time can:
  - load $O(1)$ words
  - do $O(1)$ computation
  - store $O(1)$ words
- $O(1)$ registers.
- word is w bits (should be $\log$(size of memory) ↳ rep'd w/ w bits.



② Pointer Machine

- dynamically allocated objects
- object has $O(1)$ fields
- Field = word (ex int) of pointer to object or None.

③ Python Model

① "list" = array $L[i] = L[j] + 5$ $O(1)$ time

② object with $O(1)$ attributes, $x = x.next$ $\rightarrow O(1)$ time

Ex L.append(x) -> table doubling (lecture 9)

↳ $O(1)$ time

(think about how to implement

$L = L_1 + L_2$ $O(1)$ $L = []$

$O(1+|L_1|+|L_2|)$ $O(|L_1|)$ { for x in L_1: L.append(x)

$O(|L_2|)$ { for x in L_2: L.append(x)

len(L) $O(1)$, length is stored w/ each list so just look it up

L.sort() -> $O(|L| \lg |L|)$ time to compare (lecture 3)

dict: D[key] = val -> $O(1)$ more in later lects.

long: $x+y$ $O(|x|+|y|)$, $x*y$ $O((|x|+|y|)^{\lg 3})$

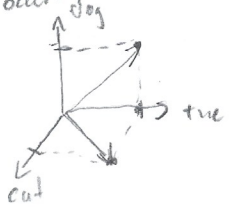
Document Distance Problem, document = sequence of words

word = string of alphanumeric chars

think of documents as a vector, D[w] = # occurances of word w

D_1 = "the cat"

D_2 = "the dog"



how similar are these documents

↳ inner product

high dot means similar

$d(D_1, D_2) = D_1 \cdot D_2 = \sum_w D_1[w] \cdot D_2[w]$

↳ scale invariant, 100 words identical ∠ 100000 words off by a.b.t

Better would be angle between $\Rightarrow d(D_1, D_2) = \arccos\left(\frac{D_1 \cdot D_2}{|D_1||D_2|}\right)$

Now we need to do 3 things

1) split docs into words
2) compute word frequencies
3) dot product.

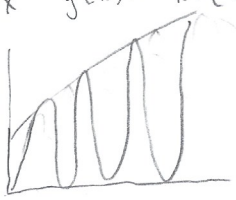
} how do you do these $\Rightarrow$

① scan through alpha numerics $O(n)$.

② for word in doc: count[word] += 1 $O(\sum |\text{words}|)$

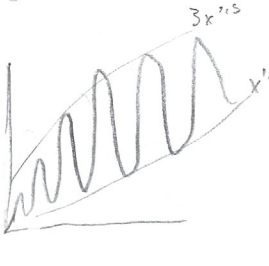
1) Asymptotic complexity Θ means you have upper and lower bound,
↳ can only use Θ if upper and lower bound differ by a constant factor (or less than)

ex $g(x) = x \cdot (1 + \sin(x))$



upper $f(x) = 2x$
lower $f(x) = 0$
can't use Θ

$g(x) = (1+\sin(x))\,x^{1.5} + x^{1.4}$



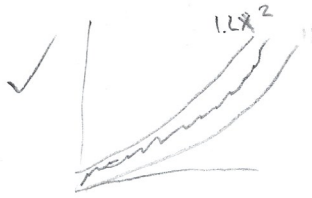
Θ ?
O $3x^{1.5}$ } differ by mult. than const factor
Ω $x^{1.4}$

ex $\log(n^{100}) \sim \Theta(\log n)$
↳ $100\log(n)$

$\log_5(n) \sim \Theta(\log n)$
↳ $\frac{\log(n)}{\log(5)}$

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

$g(x) = 1.1x^2 + (10 + \sin(x+15))\,x^{1.5} + 600 = \Theta(x^2)$



✓

$O(1.2x^2)$ } get rid of const factor for this form
$\Omega(x^2)$ }

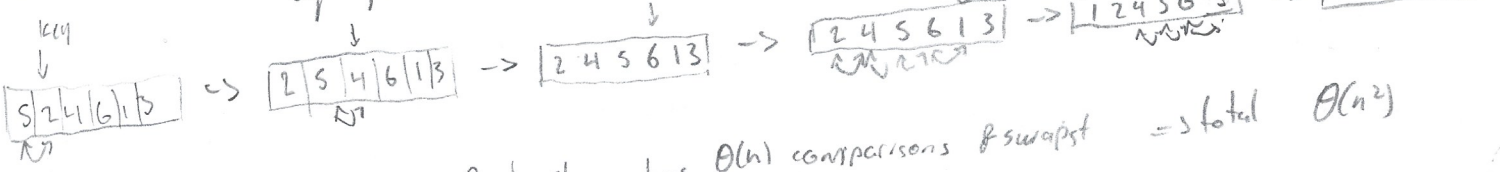
$O(x^2), \Omega(x^2)$ $\Theta(x^2)$

Worse ↓
$O(1)$
$O(\log n)$
$O(n)$
$O(n \log n)$
$O(n^2)$

2) Doc distance need to get rid of punctuation and capitals to prevent error

Recitation ↗

3) Insertion sort: For $i = 1, 2, 3, \ldots, n$ insert $A[i]$ into sorted array $[0; i-1]$
by pairwise swaps down to correct position,



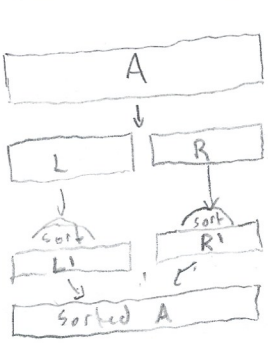
$\Theta(n)$ steps (keystrokes) Each step has $\Theta(n)$ comparisons & swaps! ⇒ total $\Theta(n^2)$

Binary Insertion: ↳ If you only care about # compares → compare >> swaps, binary search $A[0; i-1]$ already sorted
↳ search now $\Theta(\lg n)$ & $\Theta(n)$ steps → $\Theta(n \log n)$ compares (still $\Theta(n^2)$ swaps)

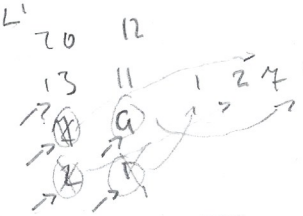
Merge Sort Divide and conquer.

$A[0:n] \rightarrow L = A[0:\frac{n}{2}-1]$
$\rightarrow R = \ldots$

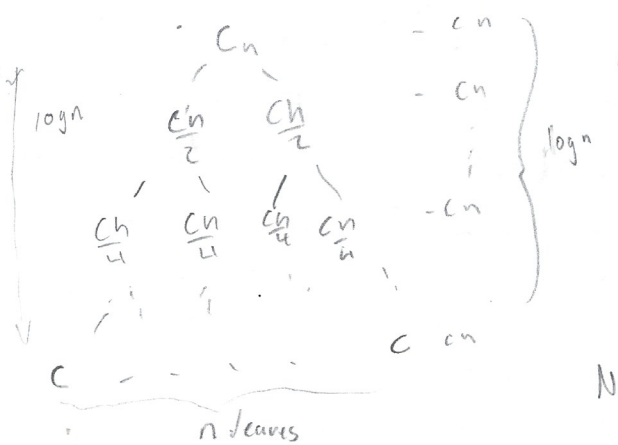
Complexity $T(n) = C_1 + 2T(n/2) + cn$
divide, recursion, merge



Merge: 2 sorted arrays as.h
two finger algo



Complexity of merging
$\Theta(n)$



cn comparisons on each level,
log n levels.

Time complexity: cn per merge, log n merges
$O(n \log_2 n)$

Space: had to create new elements (copies of arrays)
↳ $\Theta(n)$ auxillary space

Note: In place sorting ⇒ $\Theta(1)$ auxillary space
↳ For insertion sort, $\Theta(1)$ is temporary variable created when you make a swap.

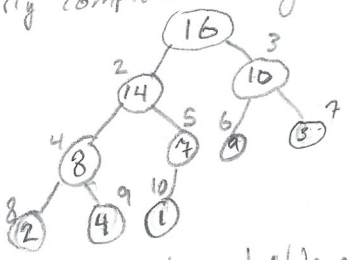
4) Priority Queue: Implements a set S of elements, each element associated w a key.

Insert (S, x) : insert element x into set S
max (S) : return element of S w largest key
extract_max(S) " " " " " " " and remove from S
increase_key (S, x, k) : increase value of x's key to new value k

Heap : An array visualized as a nearly complete binary tree

16	14	10	8	7	9	3	2	4	1

↳ nearly complete because it only has 10 elements



Heap as tree

root of tree: first element (i=1)
parent(i) = $i/2$
left(i) = $2i$ right(i) = $2i+1$

Max heap property: The key of a node ≥ the key of its children (there is also a min heap property)
↳ the value
↳ this is a property you want to maintain as you modify the heap

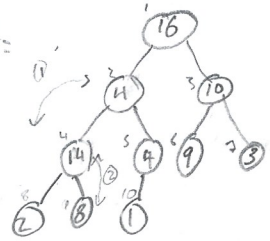
How do we maintain this property?

Heap operations:

build_max_heap : produces a max heap from an unordered array.

max_heapify : correct a single violation of the heap property in a subtree's root. Key assu...
↳ assume that trees rooted at left(i) and right(i) are max heaps.

max-heapify (A, 2) :
heap_size (A) = 10
① Exchange A[2], A[4]
Call Max_Heapify (A, 4)
② Exchange (A[4], A[9])



Exchanges called node w the largest of its ch.dre...
Simple, could run all the way down to the leaves.
→ O(log n)

Convert A[1 ... n] into a max heap:

Build_max-heap(A):
for i = n/2 downto 1
do max_heapify (A, i)

n/2+1 → n are all leaves.
↳ build your way up the tree so that you know everything below it satisfies max heap property.
kind of dragging biggest elements up

Complexity?

Observe: Max-heapify takes O(1) for node that are one level above leaves.
and in general, order ℓ time for nodes ℓ levels above leaves.
look like (n log n) But...
↳ n/4 nodes w level 1, n/8 w level 2; 1 w log n level ($\frac{n}{2}$ + $\frac{n}{4}$ + $\frac{n}{8}$ ...) = n nodes in tree
levels

Total work: $n/4(1c) + n/8(2c) + \frac{n}{16}(3c) + \cdots + 1(\log n\, c)$

$= c2^k\left(\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \cdots \frac{(k+1)}{2^k}\right)$ ← convergent series - bounded by a constant.

Set $\frac{n}{4} = 2^k$

$= 2^k \to (n/4)$ → O(n) ←

constants go away for asymptotics

Algorithm for sorting → O(n log n)

1) Build_max_heap from unordered array
2) Find max element A[1]
3) Swap elements A[1] and A[n], now max is at end of array.
4) Discard node n from heap, decrementing heap size
5) New root may violate max heap property, but children are max heap so run max_heapify (shift down log n rows)

O(n) ← build up max heap
O(1) ← move biggest element to end and discard
O(1) ← smallest elements at now but subtrees a... all max heaps so
O(1) ← run max heap (log n)
O(log n) ← repeat 2-5 n times total n log n

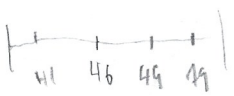
5) Scheduling and Binary search trees: Want: Fast insertion into a sorted array
↳ no data structure we have gone over can do this.

(another kind of tree: heap)

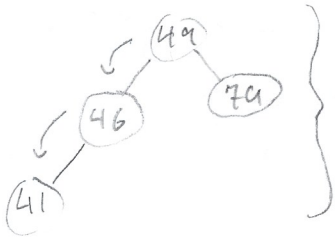
Binary Search Tree: Invariant: ∀ node x, if y is in the left subtree of x ⇒ key(y) ≤ key(x),
if y is in the right subtree of x ⇒ key(y) ≥ key(x)

node: key(x)
Pointers parent(x), left child(x), right child(x)
(unlike a heap)

Application: Airport runway reservation system:
- Airport has 1 runway
- Reservations for future landings
- Reserve request specifies landing time t.
- Add t to set R if no other landings scheduled within 4 minutes (use 4=3)



can 42 go in?
|42-41|<3
⇒ can't go in

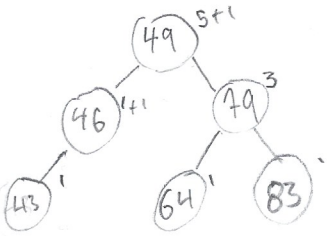


O(h)

Find_min() : go left until you hit a leaf O(h)
next_largest : O(h) time.

* New requirement: How many planes are scheduled to land at times ≤ t

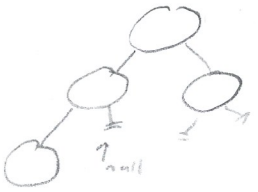
Augment BST structure.
now have subtree size associated w/ each node, note, insert/delete will modify size.



insert 43, add one to 'size' of every node you touch along the way, and assign 1 to 43.
delete more complicated

* How many ≤ t.
1) Walk down to find desired time
2) Add in nodes that are smaller
3) Add subtree's size to the left. → whenever you go right, all nodes on subtree to the left are smaller

6) AVL trees on the topic of balance
height of BST: length of longest path down to leaf,
height of node: length of longest path from it down to leaf
↳ = max(height of left child, height of right child) + 1
↳ children of leafs, null, defined to have depth -1



AVL trees require heights of left & right children of every node to differ by at most ±1

$|h_\ell - h_r| \le 1$ AVL trees are balanced: worst case is when right subtree has height 1 more than left ∀ node.

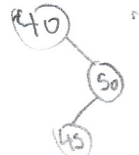
N_h = min # nodes in AVL tree w/ height h.

$N_h = 1 + N_{h-1} + N_{h-2}$, top node, then right subtree is AVL of height h-1, and left is AVL w/ h=

↳ looks like Fibbonaci, $N_h > F_h = \varphi^h/\sqrt{5}$ ⇒ $h < 1.44 \log n$

could take worst case n rotations to balance a tree.

when rotating what is below a n matters for height, so calcula h of x befor hof y



rotate

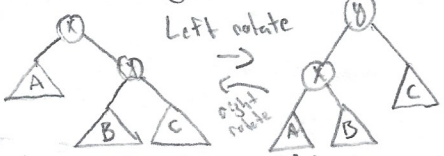
do shift
fix it
↳ rotate 50 fo
then rotate

AVL Insert
1) simple BST insert
2) Fix AVL property from the changed node up

* could unbalance every node above insertion. Rotations O(1) but could have to potentially go to O(log n)

Assume right heavy: 2 cases
if x's right child is right heavy only 1 rot
if x's right child is left heavy need 2 rot's.

Rotation



O(1) time

AVL sort: 1) Insert n items O(n log n)
O(n) 2) in-order traversal O(n)

Good for: insert/delet, min/max, successor/predecessor.

L7 | Comparison Model

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

Decision Tree: any comparison algorithm can be viewed as a tree of all possible comparisons and their outcomes and resulting answers. For any particular n.

Decision tree	algorithm
internal node	binary decision
leaf	found answer
root-to-leaf path length	algorithm execution
height of tree	running time.
	worst case running t

Searching Lower Bound: n preprocessed items (sort ↙ ↘ AVL tree), finding a given item among them in comparison model requires $\Omega(\lg n)$ in worst case.

Why? decision tree is binary & must have $\geq n$ leaves, one for each answer. => height $\geq \lg n$
↳ could have multiple paths leading to the same answ

Sorting Lower Bound

Decision trees where answers (leafs) are possible sorted orders, for n items possible orders is $n!$

leaves $\geq n!$ => height $\geq \log(n!)$ now using stirling's or sum of ln()

we get: $\Omega(n \lg n)$

These analyses were both done by thinking about them as decision trees

RAM **Linear Time Sorting (integer sorting)**

- Assume n keys sorting are integers in range $\{0, 1, \ldots, k-1\}$ (and each fits in a word)
- Can do a lot more than comparisons
- for k (not too big) can sort in $O(n)$ time

Counting Sort

```
L = array of k empty lists>          O(k)
for j in range(n):
    L[key(A[j])].append(A[j])        O(1) -> O(n)

output = []
For i in range(k):                   O(|L_i|+1) -> O(n+k)
    output.extend(L[i]).             (|L_i| = empty list)
```

Radix Sort

- imagine each integer in base b (break it into a bunch o columns)
  => # of digits = $d = \log_b k + 1$
- sort ints by least sig digit
- ⋮
- sort by most significant

} example: sort in excel by the least important column first, then... finally the most important last
↳ sorted by all in order of significance.

↳ each sort use counting sort by digit $\rightarrow O(n+b)$

Total time: $O((n+b) \cdot d)$

$= O((n+b)\log_b k)$

minimized when $b = \Theta(n)$ ↳ $O(n \log_n k)$

if $k \leq n^c$ then $O(nc)$

Hashing! In python, Dictionary: Abstract data type, maintain set of items, each w a key

- insert(item), - delete(item), - search(key) ↳ return item w given key or report doesn't exist.
  ↳ overwrites any existing key

$O(\log n)$ via AVL, but we can do better → $O(1)$

Python: dict: D[key] ~ search, D[key] = val ~ insert, del D[key] ~ delete

→ item = (key, value).

Motivations: docdist, databases, compilers & interpreters, network routers, cryptography
- Substring search, string commonalities, file/directory synchronization

Simple Approach: Direct access table, store items in an array indexed by key

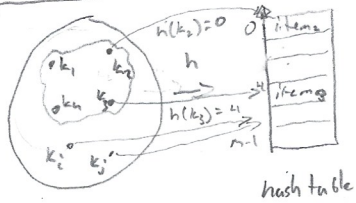
Problems: (1) Key may not be non-neg integers
(2) Huge memory hog

Solution to (1): prehash, maps keys to non-neg integers
- in theory, keys are finite and discrete (string of bits)
- in python, hash(x) is the prehash of x, hash('\0\0\0') = hash('\0\0') = 64
  Ideally: hash(x) = hash(y) ⟺ x = y
- hash = if you want to tell it what to do
  ↳ prehash should not change our time.

Solution to (2): hashing: reduce $\mathcal{U}$ of all keys (integers) down to reasonable size, m, for the table

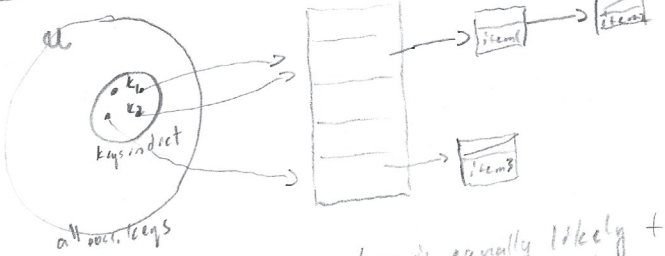
$h: \mathcal{U} \to \{0, 1, \ldots, m-1\}$

if $h(k_i) = h(k_j)$ but $k_i \neq k_j$
=> collision
↳ guaranteed to happen by pigeonhole principle.



hash table T

Chaining: linked list of colliding elements in each slot of hash table.
- worst case everything goes to 1 slot and you just end up w 1 big linked list.
  => worst case $O(n)$



Simple Uniform Hashing: each key is equally likely to be hashed to any slot of the table, independent of where other keys hashed.
↳ mutually random and independent assumptions.

Analysis: expected length of a chain, n keys stored in table w m slots => $\frac{n}{m} = \alpha$ ↳ load factor

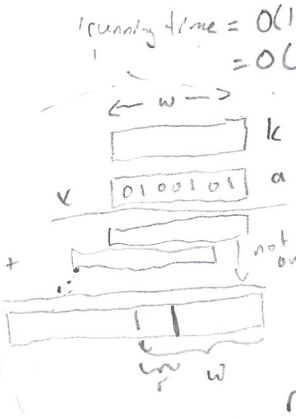
n keys. $(\frac{1}{m} + \frac{1}{m} \ldots + \frac{1}{m})$ $O(1)$ if $m = \Theta(n)$

running time = $O(1 + |\text{chain}|)$
$= O(1 + \alpha)$ ↳ compute hash

Hash Functions

1) Division method: $h(k) = k \bmod m$ (bad if k & m have common factor)
   ↳ overall bad still
2) Multiplication method: $h(k) = \lfloor (a \cdot k) \bmod 2^w \rfloor >> (w - r)$
   ↳ w bit → word size w bits,
3) Universal Hashing $h(k) = \lfloor (ak + b) \bmod p \rfloor \bmod m$ (good!)
   ↳ prime > |U| → bring # to $[0, m-1]$
   a, b random $\in \{0, 1, \ldots, p-1\}$

For worst case keys $k_1 \neq k_2$: $P_r\{h(k_1) = h(k_2)\} = 1/m$
↳ collision worst case



leftmost r bits of rightmost w bits

not random out here in the middle of stuff is colliding, becomes pretty random

$m = 2^r$

9) How to choose m? want $m = O(n)$, $\alpha = O(1)$ #scale m w/ n*

Idea: start w/ small m, e.g. m = 8, then grow/shrink as necessary.

If $n > m$, : grow table: $m \to m'$, reallocate memory; make table size m', build new hash h'
, rehash; for item in T: T'.insert(item) } $O(n+m+m')$
visit every key in old table
every item in linked lists
new table

TABLE DOUBLING

how big do we make m?

↳ double it: $m' = 2m$, only have to rebuild every $2m$

$= \Theta(1+2+4+8+\cdots+n) = \Theta(n)$ operation is not constant time?

Amortization: operation takes "T(n) amortized", => if k operations, takes $\leq k \cdot T(n)$ time

↳ thing as "T(n) on average" where average over all operations

→ Table doubling, k inserts, take $\Theta(k)$ time => $\Theta(1)$ inserts.

↳ only care about total not time of each steps for most operations.

Also: k inserts & deletes take $\Theta(k)$

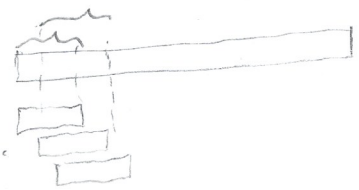
Deletion if $m < \frac{n}{4}$ -> shrink to $\frac{n}{2}$, => Amortized time $= \Theta(1)$

↳ prevents what would happen if shrink to half $m = 8\ 4\ 2\ 1$ / $m = 8\ 8\ 16\ 16\ 8$ } alternate back and forth w/ linear time for each operation.

String Matching given 2 strings s & t: does s occur as a substring of t? $\Theta(|s| \cdot (|t|-|s|)) = O(|s|\cdot|t|)$

↳ simple would be loop through t checking each string of len(s)

↳ expensive to compare strings want to view hash of these strings.



Rolling Hash: computing hash of each string wouldn't save you any time

but the next string differs by only 1 character.

r.append(c); add char c to end of x

r.skip(c); delete first char of x (assuming it's c)

r maintains a string x , r maintains a string x

Karp-Rabin Algorithm: rs() rolling hash rt() rolling hash

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

hash of s

$O(|s|)$ (check whether $s == t[i-len(s)+1 : i+1]$

yay: if equal: found match

nope else: happens w/ prob $\leq \frac{1}{|s|}$

=> O(1) expected time.

$O(|s| + |t| + \#matches \cdot |s|)$ expected time

How do we implement? which hash?

-division method: $h(k) = k \bmod m$ -> random prime $\geq |s|$

treating string x as multidigit #, u, in base a -> alphabet size

how does it change when adding a single character.

look at r.append(c):

$u \to u \cdot a + ord(c) \bmod m$

$r \to r \cdot a + ord(c)$

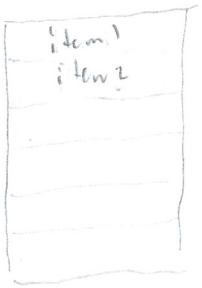
r.skip

$u \to u - c \cdot a^{|x|-1}$

Recitation 8 | Simulation Algo

10 (Hashing pt. 3) | Simplest and most important thing to remember -> open addressing

Open Addressing: no chaining



one item per slot

$m \geq n$

slots, # elements

Probing: Hash function specifies order of slots to probe for a key (for insert/search/delete)

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

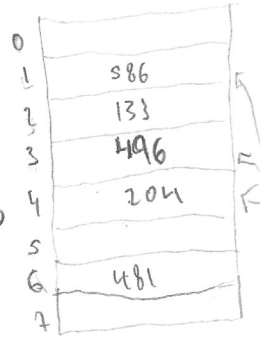
$\mathcal{U}$ = universe of keys; trial count.

try 1: $h(k,1)$, try 2: $h(k,2)$, $\ldots$, try $m-1$: $h(k,m-1)$

arbitrary key k (for a specific key!)

↳ this should be a permutation of $0, 1, \ldots, m-1$

↳ each try: cause all slots in hash table.



insert 586 $h(586,1) = 1$

insert 481 $h(481,1) = 6$.

insert 496 $h(496,1) = 4$ (fails) $\Rightarrow$ $h(496,2)$ $\ldots$ $h(496,3) = 3$

Now if I delete 586, a search for 496 would fail incorrectly. -> and replace as None

Searching? None = empty slot (Flag)

Insert: keep probing till empty slot, then insert.

Search(k): As long as slots encountered are occupied by keys $\neq k$, keep probing until you encounter k or find an empty slot.

Delete: replace deleted item w/ different flag DeleteMe

↳ different from None, prevents incorrect fa

$\Rightarrow$ Insert treats DeleteMe the same as None

$\Rightarrow$ Search keep going (different than None)

Probing Strategies

Linear Probing: $h(k,i) = (h'(k) + i) \bmod n$ (ordinary hash fcn), satisfying permutation, too much cluster

↳ Cluster: consecutive groups of occupied slots, which keep getting longer.

↳ causes you to lose coverage constant time lookup.

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

↳ if $h_2(k)$ is relatively prime to m -> permutation.

↳ $m = 2^r$, $h_2(k)$ $\forall k$ is odd, construct h_2 to only produce odd #'s.

adding a different amount with second hash helps seperate keys.

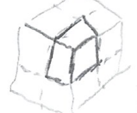
5

13 Graph Search | "explore" a graph.

Recall: graph = $G \doteq (V, E)$, V set of vertices, E: set of edges

$e = \{v, w\}$ unordered pairs, or directed graph $e = (v, w)$ ordered pairs.

Applications: • web crawling • social networking • network broadcast. • garbage collection
• model checking • checking math conjectures. • solving puzzles & games.

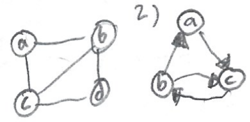


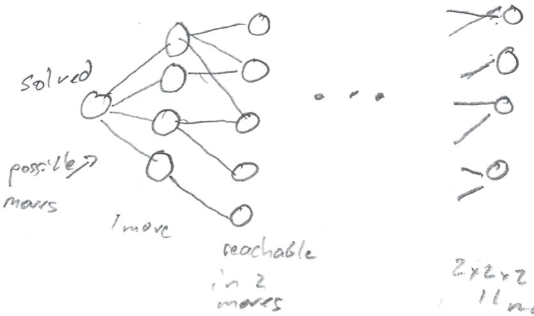
\# vertices = 8! · 3...
= 264,539,520
3 directions for each
permute the 8, 8...

Pocket Cube 2x2x2 configuration graph
• vertex for each possible state of the cube
• edge for each possible move

o—o undirected cause you can go back

2x2x2 : 11
3x3x3 : 20
nxnxn : $\Theta\left(\frac{n^2}{\lg n}\right)$





2x2x2 11 mo...

diameter

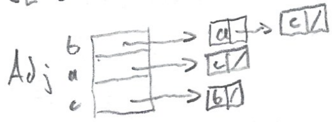
no matter the starting config can be solved in 11 moves

Graph Representation:

Adjacency list:
- Array Adj of $|V|$ linked lists
- $\forall$ vertex, $u \in V$, Adj[u] stores its neighbors
  - ↳ set $\{v \in V \mid (u,v) \in E\}$

ex graph 2) Adj[b] = {a, c}, Adj[a] = {c}, Adj[c] = {b}

implemented as:



← better if discussing multiple graphs

$\Theta(|V| + |E|)$
- ↳ V space to store V in array
- ↳ directed graph would have total of $|E|$ in linked lists
- ↳ note undirected would have $2|E|$

or

Object oriented fashion:
v.neighbors = Adj[V]

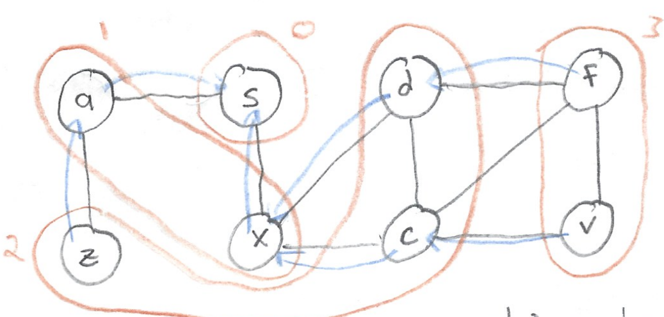
Implicit representation
- Adj(u) is a function
- v.neighbors() is a method

} not stored just called when you want to compute } good for rubix cube
↳ less space

Breadth-First Search

- visit all nodes reachable from given $s \in V$ (vertex)
  - ↳ connected component
- achieve $O(V+E)$ time
- look at nodes reachable in 0 moves {s}, 1 move Adj[s], 2 moves, ...
- careful to avoid duplicates:

```
BFS(s, Adj):          # given vertex and its adjacen...
    level = {s: Ø}    # {x: i} on each x in i moves   (keeps track of already visited)
    parent = {s: None}
    i = 1
    frontier = [s]    # what we just reached in i-...
    while frontier:
        next = []     save for next iter
        for u in frontier    current iter.
            for v in Adj[u]:
                if v not in level:
                    level[v] = i
                    parent[v] = u
                    next.append(v)
        frontier = next
        i += 1        # track current level
```



parents always point all the way back to s.
forms a tree, w root s.
shortest path back to s. (next page)

at 3, next = Ø
- ↳ at the end frontier = next
- ↳ frontier becomes Ø
- ↳ while ends.

Shortest paths

- v ← parent[v]
  ← parent[parent[v]]
  ← . . . .
  ← s

(read backwards) → is a shortest path from to v of length level[v]
↳ may not be unique.
↳ this is interesting, level is shortes # steps to get to root
↳ to get there just follow parents level[v] times.

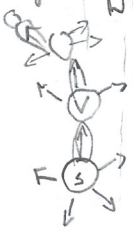
Depth first search

good example Recitation 14:20:00.

- recursively explore the graph, backtracking as necessary
- careful not to repeat vertices
  ↳ are you in parent dictionary already?
  ↳ if you visit something you've already seen, just skip it.
- will only visit vertices reachable from s.
  ↳ not necessarily the whole graph.

1) parent = {s: none}
DFS-Visit(V, Adj, s):
  for v in Adj[s]:
    if v not in parent:
      parent[v] = s
      DFS-Visit(V, Adj, v)

iterate over outgoing edges

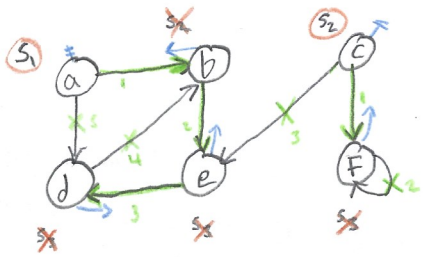


2) check each vertex, visit everything reachable from it
  ↳ check next vertex visit evrthg reachable that hasn't already been visited.

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

iterate over choices of
if we don't know where
start is not strongly
or disconnected.

ensures you search entire graph, because go ~ check each starting point
↳ finds all strongly connected components of graph.



Analysis $\Theta(V+E)$, linear time
- visit each vertex + once in DFS alone $O(V)$
- DFS-Visit(G, v) called at most once per vertex v
  ↳ pay |Adj[v]| $\Rightarrow O(\sum_{v \in V} |Adj[v]|) = O(E)$ → directed. 2E undirected.

Edge Classification

- tree edge (parent pointer): visit new vertex via edge, green edges in pic
- forward edges: node to decendent in tree ex (a)->(d)
- backward edges: node to ancestor in tree ex (d)->(b)
- cross edges: between two non-ancestor related subtrees.

In an undirected graph, can only have tree edges, and backward edges.

Cycle detection: G has a cycle $\Leftrightarrow$ DFS has a back edge.
  ↳ find cycle; start at backedge and follow tree edges.

- can't have forward edge cause it would have been found as a back edge ex (a)<->(d)
- can't have cross edge, edge would have been explored when at that node initially.

generally need to track nodes discovered, but still need to visit, Q
  ↳ use a queue

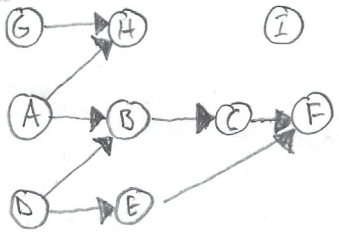
track and query if node has been visited already
  ↳ hash table

Topological Sort

Job scheduling example: given dir. acyclic graph (DAG), order vertices s.t. all edges point low → high.

Use: Topological Sort → run DFS and return reverse of finishing times for vertices

Example Intuition



Rec 14 49:00

Correctness? → for any edge $e=(u,v)$ v finishes before u finishes

Case 1: u starts before v → visit v before u finishes, v will finish before u ✓

Case 2: v starts before u

worry that you can get to u from v → u would finish before v

However, that would be a backedge, backedge ⇒ cycle.

→ can't have cycle in a dir. acyclic graph (DAG) ✓

(contradiction arrived)

15 Shortest Path

$G(E,V,w)$ — edges, vertices, w: weights, $w: E \to \mathbb{R}$

path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$(v_i, v_{i+1}) \in E$ for $0 \le i < k$

$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$, find p w/ min weight

Exponential # of paths, algos need to be smart...

Weighted Graphs

$v_0 \xrightarrow{p} v_k$, $\langle v_0 \rangle$ is a path from $v_0 \to v_0$ w/ weight 0.

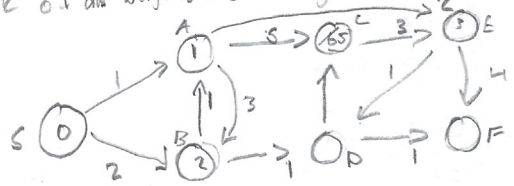
path v_0 to v_k

Shortest path weight from u to v as:

$$\delta(u,v) = \begin{cases} \min\{w(p): u \xrightarrow{p} v\}, \exists \text{ any such path.} \\ \infty, \text{ o.w.} \end{cases}$$

↳ min weight if path exists, ∞ o.w.

Think of all weights as being ∞ since you haven't found any paths yet, want to reduce them down.



$d(v)$: current weight (# inside node)

$\pi[v]$: predecessor on current best path to v

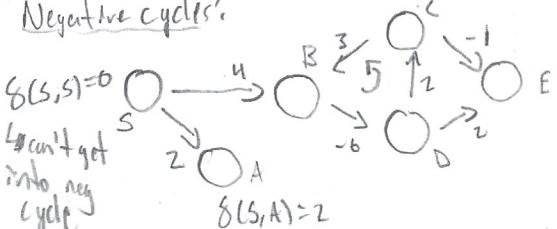
↳ $\pi[s] = \text{nil}$

when $d(v) = \delta(s,v)$ ⇒ done ✓

Negative Weights

why? reverse tolls, social networks etc...

Negative cycles:



for B, C, D, E, can keep running through B → D → C and keep getting more negative

↳ need a termination condition that accounts for

General Structure: shortest path algos. (no negative cycles)

Initialize, for $u \in V$: $d[u] \leftarrow \infty$, $\pi[u] = \text{NIL}$, $d[s] \leftarrow 0$

Repeat: select some edge (u,v) [somehow]

"Relax" edge (u,v): if $d[v] > d[u] + w(u,v)$: (found a better path)

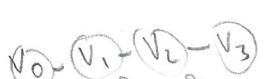
$d[v] = d[u] + w(u,v)$

$\pi[v] \leftarrow u$

until all edges have $d[v] \le d[u] + w(u,v)$

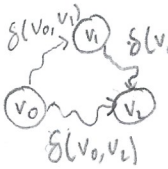
Optimal Substructure

1) Subpaths of a shortest path are shortest paths.



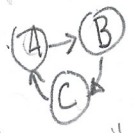
Pos shortest path $p_{12} + p_{23}$ shortest from 1→3

2) $\delta(v_0, v_2) \le \delta(v_0, v_1) + \delta$



Why acyclic?

class pre-req



to take A, need to take C, to take B, to take C

no solution

can't start w/ any

so no ordering exists if there is a cycle

acyclic means no back

16 Dijkstra

Relaxation is safe: maintains invariant $d[v] \geq \delta(s,v)$

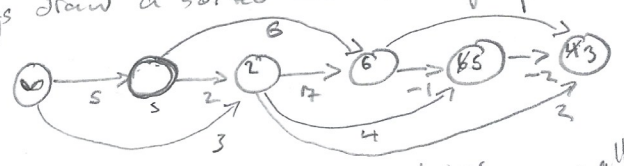
```
Relax(u,v,w)
  if d[v] > d[u] + w(u,v)
    d[v] = d[u] + w(u,v)
    π[v] = u
```

Lemma: The relaxation operation maintains the invariant that $d[v] \geq \delta(s,v)$ $\forall v \in V$

For **DAG** (no neg cycles) use topological sort (whenever DAG, try topol. sort is a great option)

1) Topological sort the DAG. Path from u to v implies u is before v in the ordering
2) One pass over vertices in topologically sorted order, relax each edge that leaves each vertex

$O(V+E)$ Can always draw a sorted DAG (nearly), everything to left of source w/ ∞



go node-by-node, only moving to next once all vertices have been rela

Dijkstra (G, w, s) → start — no neg cycles

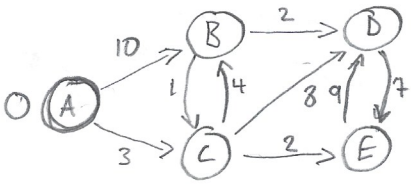
Initialize(G,s) — d[s]=0 — make the start

$S \leftarrow \emptyset$ → set that has been processed

$Q \leftarrow V[G]$ → all vertices set to be processed → priority queue ↳ priorities are d values ↳ first is s cause d(s)=0

```
While Q ≠ ∅:
    u ← Extract-min(Q)        % delete u from Q
    S ← S ∪ {u}
    for each vertex v ∈ Adj[u]   → every vertex you can reach from u
        Relax(u,v,w)
```

good to read this proof



$S = \{\}$ $\quad Q = \{A, B, C, D, E\}$; $(A)=0$, others ∞

$S = \{A\}$ $\quad Q = \{B, C, D, E\}$; $10, (3), \infty, \infty$

$S = \{A, C\}$ $\quad Q = \{B, D, E\}$; $7, 11, (5)$

$S = \{A, C, E\}$ $\quad Q = \{B, D\}$; $(7), 11$

⋮

Complexity

$\Theta(V)$ inserts into priority Queue

$\Theta(V)$ Extract-min ops.

$\Theta(E)$ Decrease key ops ↳ Relax()

Implement queues as:

Arrays

$\Theta(V)$ For extract min
$\Theta(1)$ For decrease key
$\Rightarrow \Theta(V \cdot V + E \cdot 1) \approx \Theta(V^2)$

Binary Min-Heap

$\Theta(\lg V)$ For ext. min
$\Theta(\lg V)$ For decrs. key
$\Rightarrow \Theta((V+E)\lg V) \approx \Theta(V \lg V + E)$

↑ w/ Fibonacci heaps (6.046)

☺

17 | Bellman-Ford

(used if neg cycles are a possibility)

Generic S.P. Algo:

Initialize for $v \in V$ $d[v] \leftarrow \infty$, $\pi[v] \leftarrow nil$

$d[s] \leftarrow 0$

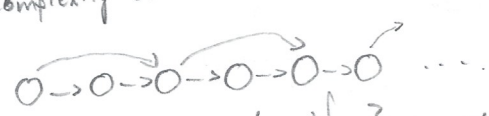
Main. repeat select edge [somehow]

Relax(u, v, w)

until you can't relax anymore

Problems in this general algo

1) Complexity could be exponential time.



2) Will not terminate if ∃ a -ve cycle reachable from the source.

```
Bellman_Ford (G, w, s)        -> d[s]=0, others ∞
    Initialize ()             -> then just relax every edge |V|-1 times
    for i=1 to |V|-1          <- this loop O(VE) ->
        for each edge (u,v) ∈ E
            Relax(u,v,w)
    for each edge (u,v) ∈ E :  <- would be right here if no -ve cycles -> check for -ve cycles.
        if d[v] > d[u] + w(u,v)
            then report -ve cycle exists.
```

```
Relax(u,v,w):
    if d[v] > d[u] + w(u,v):
        d[v] = d[u] + w(u,v)
        π[v] = u
```

You relax every edge, in every pass.
You make |V|-1 passes.

Theorem: If $G(V,E)$ contains no -ve weight cycles, then:
after B-F executes, $d[v] = \delta(s,v)$ $\forall v \in V$.

Corolary: If a value $d[v]$ fails to converge after $|V|-1$ passes, then,
$\exists$ a -ve weight cycle reachable from s.

Proofs in notes

you have topol. sorted vertices

guarantee to have $\delta(v_0, v_1)$ after first loop

then to have $\delta(v_0, v_2)$ after second loop

⋮

$\delta(v_0, v_k)$ after k passes.

if k+1 vertices, get k after $|V-1|$ iterations

essentially expanding the frontier by 1 every time and will have found shortest simple path

other condition is to check that there hasn't been a negative cycle updating a bunch.

you will catch it because no paths should be updating after V-1 passes

order edges are relaxed each loop doesn't matter.

- No known algorithm for finding shortest simple path if -ve cycle exists,
- so is longest path

Lecture 18 | Speeding Up Dijkstra

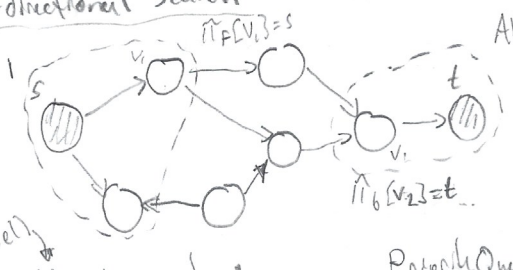
doesn't change worst case, improves "average" case.

Dijkstra: Initialize() $d[s]=0$ $d[s \neq s]=\infty$
while $Q \neq \emptyset$
do $u \leftarrow$ Extract_Min(Q)
for each vertex $v \in$ Adj[u]
do RELAX(u,v,w)

$Q \leftarrow V[G]$ $s \to t$
stop if $u = t$

DS needs edges able to be traversed both ways.

Bi-directional Search



Alternate: • forwards search from s
• backwards search from t (following edges backwards)

π_f: forward
π_b: backward.

$d_f[u]$: distances for forward search
$d_b[u]$: distances for backward search.

Priority Queues: Q_f: forward
Q_b: backwards.

Initialized

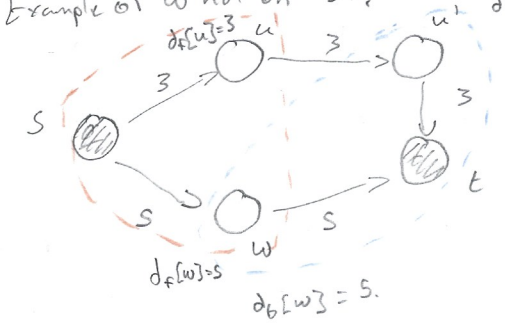
2 different priorities:
$d_f[s]=0$ $d_b[t]=0$

fronters meeting

Termination Condition: Some vertex u has been processed both in the forward and backwards sea
and has been deleted from Q_f and Q_b.

Shortest Path: If w was processed first from both Q_f, Q_b } Not quite: w may not be on S.P.
find S.P using π_f from s to w
find S.P. using π_b from t to w (backwards.

Example of w not on SP.



	$\delta(s,x)$	fwd $d_f[x]$	Q_f	$\delta(t,x)$	bwd $d_b[x]$	Q_b
1)	{s}	{u u' w t}		2) {t}	{u u' w s}	
	0	3 ∞ 5 ∞		0	∞ 3 5 ∞	
3)	{s u}	{u' w t}		4) {t u'}	{u w s}	
	0 3	6 5 ∞		0 3	6 5 ∞	
5)	{s u w}	{u' t}		6) {t u' w}	{u s}	
	0 3 5	0 10		0 3 5	6 10	

Terminate! w processed for both forwards and backwards.
↳ But incorrect length of 10 (should be 9)

Do a little more work...

→ Find an x (possibly different from w) w minimum value of $d_f[x] + d_b[x]$ // look in Q_f and Q_b

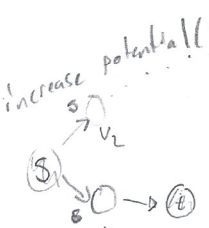
find an x (possibly different from w) w minimum value of $d_f[x] + d_b[x]$ // look in Q_f and Q_b
$d_f[u] + d_b[u] = 9$ ✓
$d_f[u'] + d_b[u'] = 9$ ✓

Goal-Directed Search

Modify edge weights with potential functions:

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

Correctness: $\bar{w}(p) = w(p) - \lambda_t(s) + \lambda_t(t)$ ↳ any path from s→t shifted by same amount. shortest path before will still be shortest path



how to get it to guess v_1 instead v_2.

landmark $l \in V$ } $\lambda_t(u) = \delta(u,l) - \delta(t,l)$
precompute $\delta(u,l)$

this increases speed, have a point you know you need to pass through.

9/

Dynamic Programming (DP) | general powerful algo. design technique.

* DP ≈ careful brute force, subproblems and reuse.

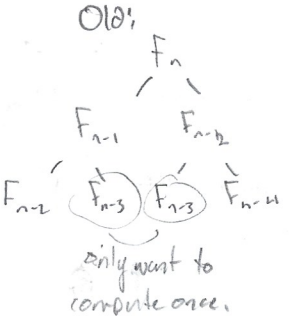
Fibonacci Numbers: $F_1 = F_2 = 1$ $\quad F_n = F_{n-1} + F_{n-2}$ goal: compute F_n

Naive recursive algo: Fib(n):
```
if n ≤ 2: f = 1
else: f = fib(n-1) + fib(n-2)
return f
```

Exponential: $T(n) = T(n-1) + T(n-2) + \Theta(1)$
$\geq F_n \approx \varphi^n$
$T(n) \geq 2 \cdot T(n-2)$
$= \Theta(2^{n/2})$

Memoized DP algorithm: whenever you compute a fibonacci number, store it in a dictionary.

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    if n ≤ 2: f = 1                      # same as above
    else: f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```



only want to compute once.

fib(k) only recurses the first time it's called, $\forall k$
- memoized calls cost $\Theta(1)$
- number of non-memoized calls (first time you call fib(a) $\forall a$) is n: fib(1), fib(2), ..., fib(n)
- non recursive work per call is constant. $\Rightarrow$ time $= \Theta(n)$ → n calls, each costing $\Theta(1)$

* DP ≈ recursion + memoization.
- memoize (remember, or write down rough work), and reuse solutions to subproblems that help solve the problem

$\Rightarrow$ time = #subproblems · time/subproblem
($\Theta(1)$ → don't count memoized recurrences)

Bottom-Up DP algorithm:

```
fib = {}
For k in range(1, n+1):
    if k ≤ 2: f = 1
    else: f = fib[k-1] + fib[k-2]
    fib[k] = f
return fib[n]
```

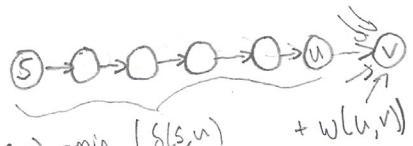
in general →

- bottom up does exactly same computation as memoized
- topological sort of subproblem dependency DAG

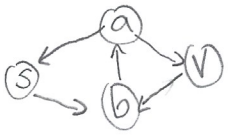
- can often save space (only need to store last 2 values)

In all dynamic programs you can always go bottom-up like here in fibonacci

Shortest paths: $\delta(s, v)$ $\forall v$



$\delta(s,v) = \min_{(u,v)} (\delta(s,u) + w(u,v))$



$\delta(s,v)$
$\delta(s,a)$
$\delta(s,b)$
$\delta(s,s)$ $\delta(s,v)$

Guessing: don't know the answer? guess, try all guesses, take the best one.

DP = guessing + recursion + memoization

Infinite time on graphs w/ cycles.

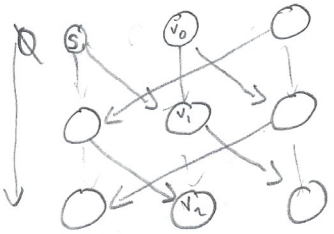
DAGs: $O(V+E)$
time for subprob $\delta(s,v)$ = indegree(v) + 1
$\Rightarrow$ total time = $\sum_{v \in V}$ indeg(v) + 1 = $\Theta(E) + V$

subproblem dependencies should be acyclic (otherwise infinite)
↳ that why we topl. sorted DAG for bottom up.

If cyclic need to make it acyclic. How?

Explode graph into multiple layers. Make every edge go down to next layer

"time"



$\delta_k(s,v)$ = weight of shortest s->v path that uses $\leq k$ edges

$\delta_k(s,v) = \min_{(u,v) \in E} (\delta_{k-1}(s,u) + w(u,v))$

subproblems = V^2

20) 5 steps for dynamic programming

① define subproblems → #subproblems
② guess (part of solution) → #choices for guess (similar)
③ relate subproblem solutions (recurrence) → time per subproblem
④ recurse and memoize (check subproblem recurrence is acyclic → has topological sort)
or build DP table bottom-up
⑤ solve original problem

eg fibonacci:
① F_k for $k = 1, \dots, n$ — n of them
② no guess
③ $F_k = F_{k-1} + F_{k-2}$ — 1

Text Justification: split text into "good" lines

text = list of words

$$badness(i,j) = \begin{cases} \infty & \text{if they don't fit} \\ (\text{page width} - \text{total width})^3 \end{cases}$$

use words[i:j] as line.

① subproblems: suffixes words[i:]
- #subproblems: n

② guess: where to start 2nd line.
- #choices ≤ n − i = O(n)

③ recurrence: DP(i)
$= \min(DP(j) + badness(i,j) \text{ for } j \text{ in range}(i+1, n+1))$

time per subprob
O(n)

DP(n) = ∅ → no words after n.

④ topo. order: i = n, n−1, …, ∅
total time $\Theta(n^2)$

⑤ original problem
DP(∅)

Parent pointers: remember which guess is best.

parent[i] = argmin(…) = j value.

If you ignore the first i words, what's the best outcome?

Best outcome if you chose to leave i+1 or i+2 or …

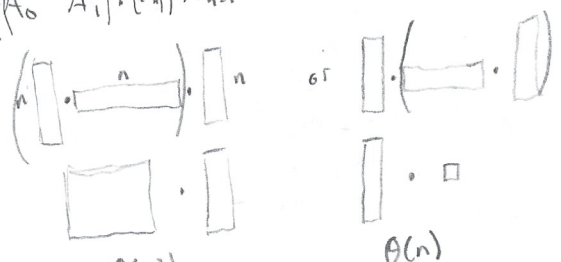
Each DP you solve for the best case of the subproblems, then take the min of current badness + best case. Keep working your way back up to right answer

21) Subproblems for strings/sequences.
- suffixes x[i:] ∀i
- prefixes x[:i] ∀i
- substrings x[i:j]

} O(n) (never use both ↔ instead substring)

Parenthesization: optimal evaluation of associative expression.

$(A_0 \; A_1) \cdot (\dots) \; A_{n-1}$ reduce cost of multiplication.



② guess the outermost/last multiplication

$(A_i \cdots A_{k-1}) \cdot (A_k \cdots A_{j-1})$

$(A_0 \cdots A_{k'}) \cdot (A_{k'+1} \cdots A_{k-1})$ — not prefix/suffix ↔ need to use substring

$O(j-i+1)$
$O(n)$

① subproblem = optimal evaluation $A_i \cdots A_{j-1}$ (i:j)

③ recurrence $DP(i,j) = \min($
$DP(i,k) + DP(k,j) + \text{cost of } (A_{i:k}) \cdot (A_{k:j})$
for k in range(i+1, j)

time/subproblem = O(n)
④ time = $\Theta(n^3)$
topological order: increasing substring size

Edit distance: given 2 strings x and y. what's the cheapest possible sequence of character edits to turn $x \to y$.

chars edits: insert c, delete c, replace $c \to c'$ each operation and c has different cost.

longest common subsequence: HIEROGLYPHOLOGY, MICHAELANGELO => HELLO

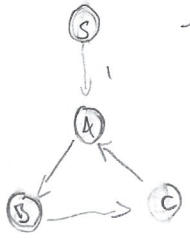
cost of insert / delete = 1

$$\text{replace} = \begin{cases} 0 & \text{if } c = c' \\ \infty & \text{else} \end{cases}$$

① subproblem = edit distance on $x[i:]$ and $y[j:]$ $\forall i, j$ $\to$ #subproblems $= \Theta(|x| \cdot |y|)$

② guess

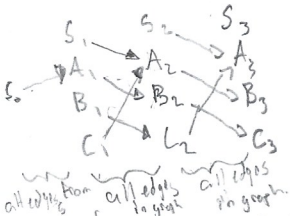
Cycles



The minimum distance calculation will call its own distance. This will lead to an infinite number of calls and never solved. Need to incorperate time.

$\delta_k(s,v) = \delta_{k-1}(u,v) + w_{uv}$

let's build this graph



- longest simple path is $V-1$, so we need 3 layers

$V' = (V-1) \cdot V + 1 = O(V^2)$

$E' = O(VE)$

$O(V' + E') = O(VE)$

This is Belman Ford and is the actual intuition behind why it w

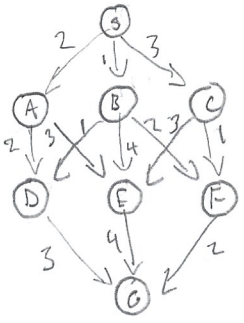
All dynamic progcammmy problems can be thought of as a graph. Solve it smartly w/o building a gra

W optimal substructure, you are solving smaller parts of the problem and building on them (shortest paths) This is shown below, find shortest path for subproblem then chose which problem is best slightly bigger problem.

→ This is precisely what turns the exponential numb of calculations into polynomial.

Just store your results after you compute them.

Example DP helped intuition from recitation.



$$\delta(S,D) = \min\begin{pmatrix} \delta(S,A) + w_{AD} \\ \delta(S,B) + w_{BD} \end{pmatrix} = 2$$

$$\delta(S,E) = \min\begin{pmatrix} \delta(S,A) + w_{AE} \\ \delta(S,B) + w_{BE} \\ \delta(S,C) + w_{CE} \end{pmatrix} = 5$$

$$\delta(S,F) = \min\begin{pmatrix} \delta(S,B) + w_{BF} \\ \delta(S,C) + w_{CF} \end{pmatrix} = 3$$

$$\delta(S,G) = \min\begin{pmatrix} \delta(S,D) + w_{DG} \\ \delta(S,E) + w_{EG} \\ \delta(S,F) + w_{FG} \end{pmatrix} = 5$$

base case is $\delta(s,s) = 0$

Key takeaway is that having $\delta(S,E)$ calculated prevents it needed to be recalculated again. Further having $\delta(S,A)$, $\delta(S,B)$, $\delta(S,F)$ calculated prevents them being recalculated for $\delta(S,D)$, $\delta(S,E)$, $\delta(S,F)$, $\delta(S,G)$.