An algorithm is a way of defining computation / informal
- for solving some problem
- Generate output from an input
- It is the most analog of a computer program

We write them in pseudocode
in a way that can be mathematical

It runs on model of computation
that says what your computer
is allowed to do → specifies what operations can be done and the cost (time) of them

Python model
offer ram and other machine
arrays (references)

1 "Listy" = array → O(1)
2 objects with O(1) operations (with reasonable number of arguments)

If we do L.append(x), instead
of copying all element in
a new array, python will
table doubling with O(1)

L.sort → O(n log n)

sort [array] → O(1)

Document distance problem

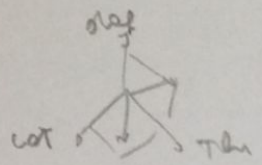
d (D1, D2)
↑ ↑ ↑
distance documents

how a document is similar to another and how much

document = sequence of words,
each word is a string
of alphanum chars.

Idea: shared words, to define distance
A document can be a vector: D[w]
↑
number of word w in D

d: (D1 · D2) = D1 · D2



find the angle between two vectors

Algorithm analysis - how to work (way of every input)

- Input size, varies based on the problem; could an array, two integers, or graphs. In each case we want differently
- Running time, given an input, # the number of steps needed. We want to define steps as machine independent as possible, so we define each line as a step
  The notation is good from our RAM and pseudocode viewpoint.
  But it is still messy and we don't need all of that info

Worst-case

We always want to consider the worst case, b/c
- that it gives us an upper bound, we know that it will never take any longer
- The worst case happen pretty often, for example ~~especially~~ when we search an element not present in a list.
- The ~~average~~ average case is ~~usually~~ roughly as bad as worst-case

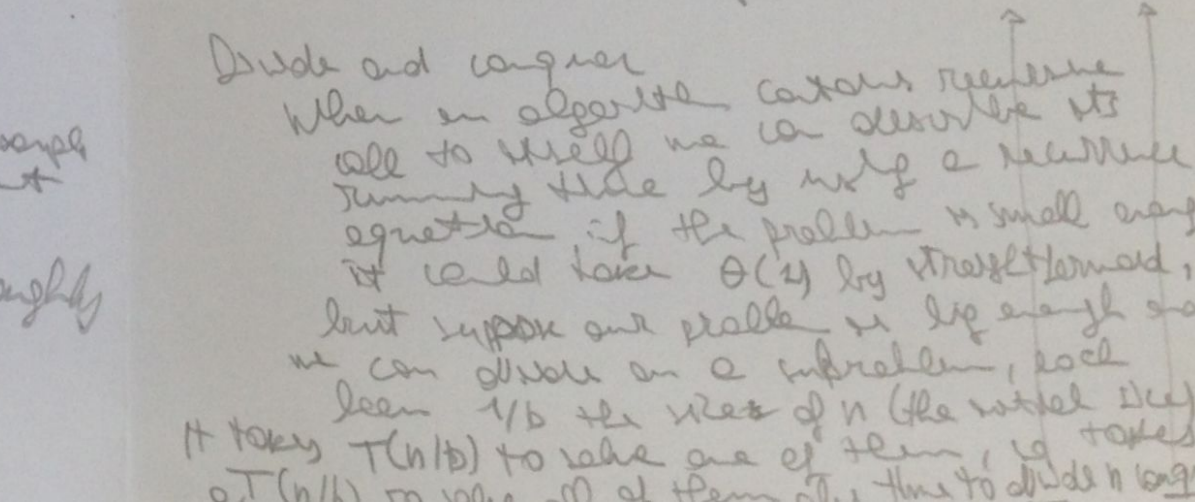
Order of growth

We now consider that leading terms since the lower order terms are insignificant as n grows (the running time). We also ignore the constant coefficient on leading term

An algorithm is more efficient if its worst case running time has lower order of growth

Due to constant factors and lower order terms, an algorithm with higher order of growth can take less time than an algorithm with lower on small inputs.

On large inputs the worst worst case of the ~~fastest~~ will run quicker than the higher order of growth



Divide and conquer

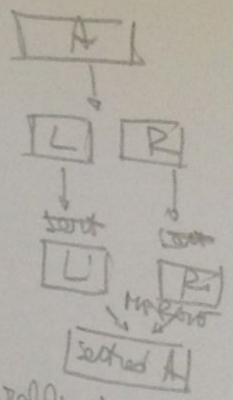
When an algorithm contains recursive call to itself we can describe its running time by using a recurrence equation. If the problem is small enough it could take $\Theta(1)$ by straightforward, but suppose our problem is big enough so we can divide on a subproblem, each been $1/b$ the size of n (the initial size). It takes $T(n/b)$ to solve one of them, and takes $aT(n/b)$ to solve all of them. The time to divide n (ignore

Divide and conquer

Divide and conquer is typically a recursive approach.

It has three steps

- Divide into subproblems that are instances of the same problem
- Conquer the subproblems recursively, in the case the subproblem is small enough, resolve it straightforwardly
- Combine the solutions for the subproblems into the solution for the original problem



Merge sort is the most common example.

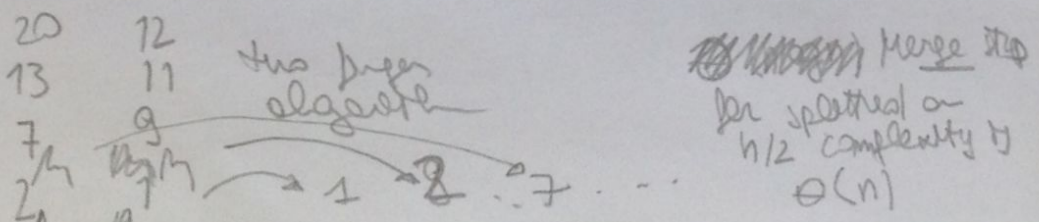
Here it goes; Divide an into two subsequences of size $n/2$, where n is the length. Conquer the two subsequences using merge sort. Merge the sorted sequences. Ends with n equals to 1

Merge (A, p, q, r) take a subarray in A going from index p to q, and another from q to r, assuming that they are sorted, and it merges forms the subarray $p-q$.

It takes $O(n)$ where $n = r - p + 1$.

that is the invariant (L and R are sorted)

The comparison in merging is really effective since you only compare one element from each list

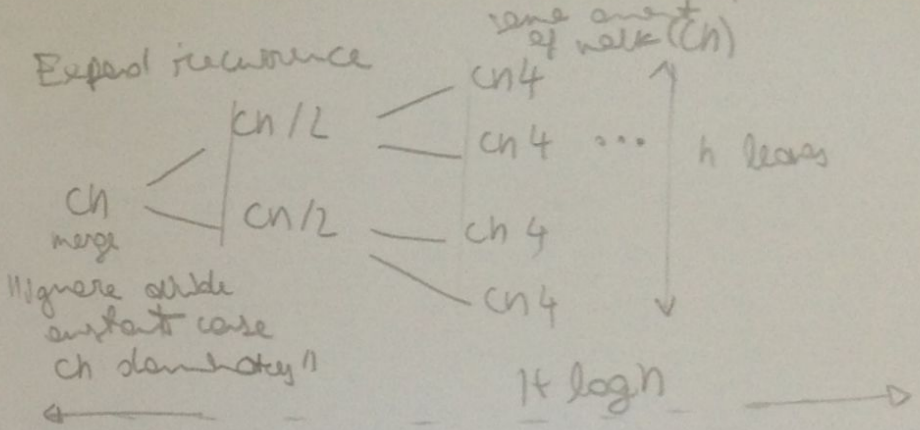


Merge is for splitted an $n/2$ complexity is $\Theta(n)$

Complexity $T(n) = \underbrace{c}_{\text{divide is constant time}} + \underbrace{2T(n/2)}_{\text{recursion over divided problems}} + \underbrace{c \cdot n}_{\text{merge}}$

work done on n items

Expand recurrence



$\Rightarrow n$ for each level and $\log n$ levels. $\Rightarrow n\Theta(n \log n)$

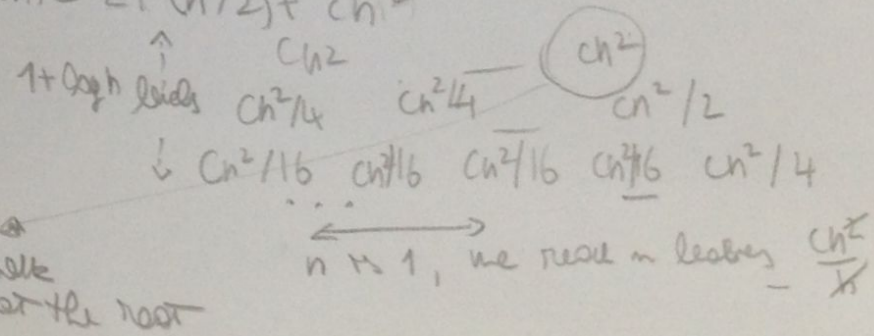
Insertion sort does in-place sorting while merge sort you need extra space

merge-sort $\Rightarrow O(n)$ auxiliary space

in-place sort $\Rightarrow O(1)$, the temp variable needed for swapping

Recurrence solving

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



Asymptotic notation

used to describe running time.

It applies to functions for example suppose that it is an^2+bn+c, we abstracted some details of this function to have $O(n^2)$.

Asym. can be used to characterize other aspects of an algorithm, even with running time however, sometimes we're interested in worst case running time, others we want the running time no matter the input

Domain of natural #s

for instance $\Theta(g(n)) =$ $T(n)$ = running time
$\{f(n)$ there exists positive n = input size
constants c_1, c_2 and n_0 such
that $0 \le c_1 g \le f(n) \le c_2 g$ for
all $n \ge n_0\}$

A function $f(n)$ belongs to the set $\Theta(g(n))$ if it exist constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for ~~sufficiently~~ sufficiently large n

by $f(n) = \Theta(g(n))$ we mean $f(n) \in \Theta(g(n))$

$f(n) \in \Theta(g(n))$ members must be asymptotically positive. this must hold for every asymptotic notation

For asymptotic notation we throw away ~~exact~~ leading coefficient and lower order terms To justify it suppose we have

$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$, we divide by n^2

$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$ ← right hand holds for $c_2 \ge \frac{1}{2}$, left hand holds for

the lhs, result holds so → $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

lower order terms of an asymptotically positive function can be ignored because they are insignificant for large n. Some for a ~~highest~~ highest order. coefficient

O notation

gives an asymptotic upper bound. Given $g(n)$, we define $O(g(n))$ as

$O(g(n)) = \{f(n)$: there exist positive c and n_0 such that $0 \le f(n) \le cg(n)$ for $n \ge n_0$

Note that $f(n) = \Theta(g(n))$ implies that $f(n) = O(g(n))$

When $c > 0$, any given function with the form $an + b$ is $O(n^2)$.

Usually O notation can cover every input being an upper bound, although for Θ that depends on the type of input

Ω ~~gives~~ notation

It provides an asymptotically lower bound for a given $g(n)$, $\Omega(g(n))$ is

$\Omega(g(n)) = \{f(n)$: there exist c and n_0 such that $0 \le cg(n) \le f(n)$ for $n \ge n_0$

So for any $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $\Omega(g(n))$

So a running time of $\Omega(g(n))$ means that no matter the input for n there, the running time is at least $g(n)$

Heap.

Priority queue

Implements a set S of elements, each element is associated with a key

operations

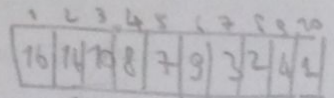
INSERT(S, x): insert element x into set S

MAX(S): return the element with largest key

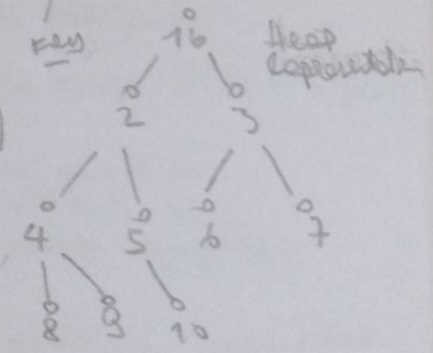
extract_max: // and removes it from x

increase_key(S, x, k): increase key x's key to k

Heap is an implementation of the priority queue, in an array structure visualized as a nearly complete bin. Tree (should be a multiple of two to be complete, like 4 8 16 2 or)



key



Heap representation

Heap as a tree

root: first element ($i = 1$)

parent(i) = $i/2$

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

Max-Heap property: the key of a node is $\geq$ the keys of its children

Min-Heap ... $\leq$ to

Heap operations

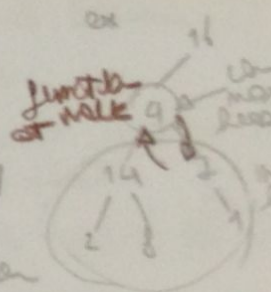
build_max_heap: produces a max heap from an unordered array (A, i)

max_heapify: corrects a single violation of the heap property in a subtree's root

heap_size(A)

Most of time you're forming a node up, because inevitably you add up, must be MAX_HEAPIFY and by working from bottom so no less work

Assume that the trees rooted at left(i) and right(i) are max heaps



"it recursively to MAX_HEAPIFY down to the leaves when needed"

$O(\log n)$

Heapsort

We take an unordered array A[1..n] TO a max heap

for $i = n/2$ to 1
  max_heapify(A, i)

leafs are A[n/2+1..n]

You are calling max_heapify multiple times, but every time, you verify the precondition, to start from n/2 to jump the leaves and then ~~keep~~ you ~~can~~ ~~top~~ work your way up

Max_heapify takes constant $O(1)$ time for nodes one level above the leaves and $O(L)$ for nodes L levels above and we have $n/4$ at level 1, $n/8$ at 2 ... 1 at $\log n$ level

$n \log n$

$\Theta(n)$

A binary search tree is the tree representation of binary search

Runway reservation problem

airport with a single runway

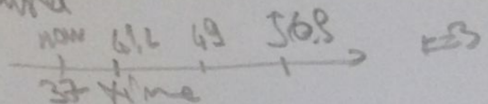
*Time is part of the system.

- reservations for future landings
- Reserve requests with landing time T $|R| = n$
- Add T to the set R, if no other landings are scheduled in k minutes $O(\lg n)$

insert operation with constraint to be checked in order to be able to insert

remove time from R after landing

Example



req 53 OK
req 44 not allowed (too close to 41.4)
req 20 not al. (past)

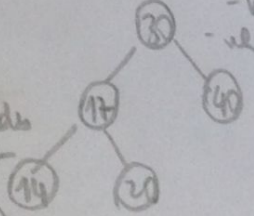
Unsorted array, list:
Does not meet requirement as scheduling requests would take O(n), lin

Sorted array
Does not matter, search is melt, insertion takes O(n) cause you have to shift

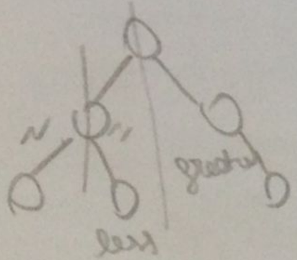
Sorted list solves the insertion problem but search takes too much

fast insertion on sorted array → BST
they have strong invariant

example



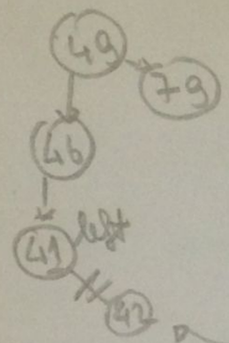
- node x key(x)
- no pointers unlike a heap parent(x) leftchild(x) right(x)



Invariant for all nodes x, if y is the left subtree of x, key(y) ≤ key(x)
y is right... key(y) ≥ key(x)

runway problem

Inserting
Insert 49
Insert 79
Insert 46
Insert 41



Insert 42 (k=3)
You proceed like in binary search
42 with 49 ✓
42 with 46 ✓
42 with 41 ✗ (not k=3)

I don't reserve

If h is height, insertion with check is done in O(h)
(h because the tree isn't necessarily balanced)

find_min()
go to the left-most leaf O(h)

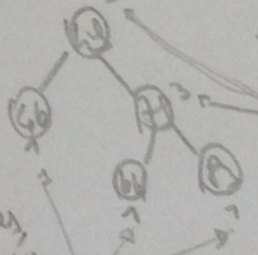
next_larger(x)
(next larger of a given value) O(h)

Augmented BSTs

For example we want to be able to compute rank(t):
How many planes are scheduled to land at time ≤ t

Don't change the complexity!

Example



Insert and delete modify the numbers that correspond to subtree sizes, to compute rank(t)

computing the count and the subtrees

subtree sizes are 1 at leafs

Rank stuff ...

Insert
follow the path down and increment by 1 and the new node is 1 ⇝ In general just run the regular insert, if it fails then down, update

Lec 3 — on BSTs

Algorithm type
- queries — max, min; next level; search
- updates — insert

If rep. invariant does not hold queries give wrong answers

If you want to speed up your code you have to make sure the invariant still holds

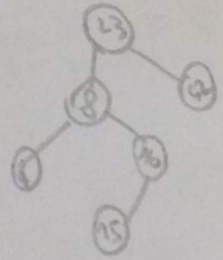
A good way to keep it to to create check_ri(), then delete it
- check_ri → rep invariant
- delete it → to avoid slowdown while code

DATA TYPES ↑

BST example

nodes (py objects)
- left
- right
- parent
- key

each node has it

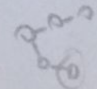


Invariant
all Nodes left smaller, all nodes right bigger of a given element

- Fast search!

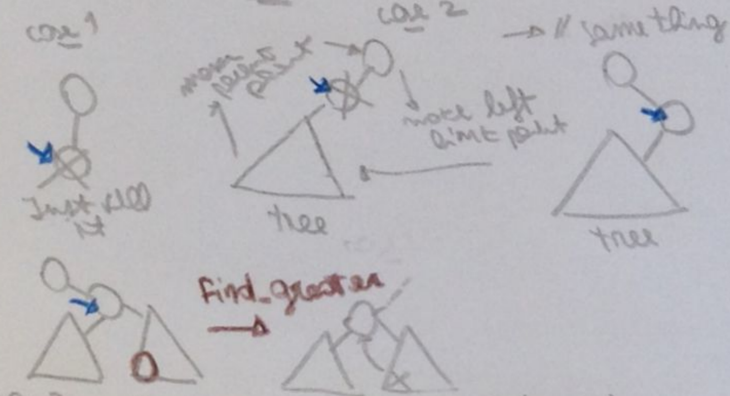
python code
find()
is binary search
O(h), tree is unbalanced

find_next_larger()
case 1 I have a node, you go right and then reach the leftmost leaf.
case 2 if it's a leaf or has only left, you go up and then do the thing
case 3



go up while able to go left

BST deletions



delete next greater and move it to the node you're trying to delete (x), delete by replacing

O(h):
- search the key is O(h)
- case 3 (worst) takes O(h) to find successor

1 can augment a tree by make a node fatter by storing more data

BSTs are trees, some nodes have only left/right, others two childs

Invariant is that $x \geq$ left and $x \leq$ right.

You can have the sorted order by doing in-order traversal.

operations are usually in $O(h)$

The height of a tree is given by the height of the largest path from root to leaf

A tree is balanced if the height is $O(\lg n)$

We use AVL trees to maintain balance

The height of a node is the longest path from node to the leaf. The root has height that is max height among left and right, + 1.

I can use data structure augm. to store height value

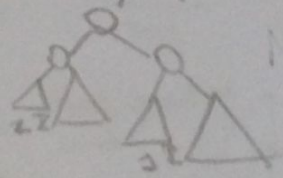
"will store null pointers after leaves as -1 to make it work"

AVL Trees

Requires heights of left and right children of every node to differ by at most 1.

AVL trees are balanced

In the worst case is when the right subtree has height one more than left for every node



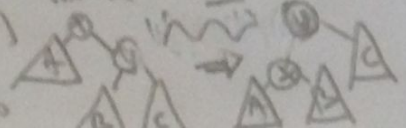
N_h = minimal # of nodes possible in an AVL tree of height h

$N_h = 1 + N_{h-1} + N_{h-2}$
right left

$h < 2 \lg n$

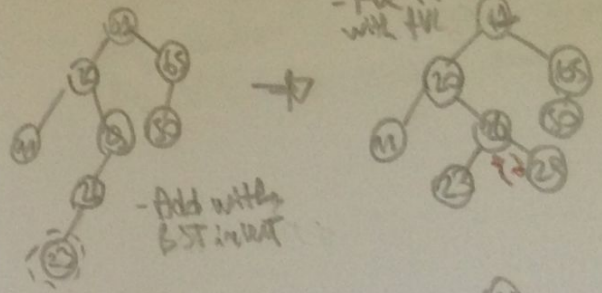
If we can achieve this the tree is balanced!

AVL insert
- Simple BST insertion
- Fix AVL property
  - with rotations

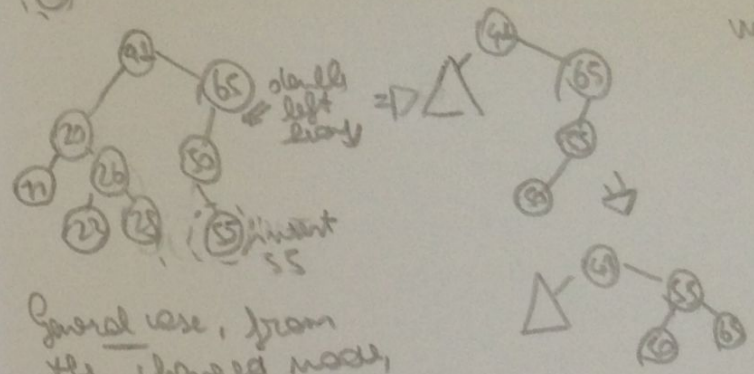


leftRotate(x)
$O(1)$
"root moves to

satisfies the ordering BST property



with a right rotation of node "29" that is double left heavy



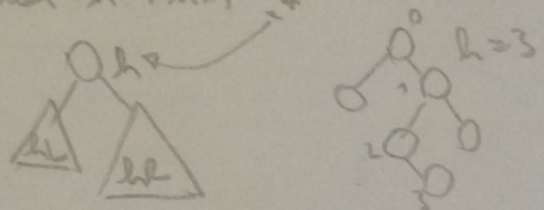
with a right rotation I get the symmetric, violent solving. I make a double rotation: the first to eliminate the pattern to ?, then another one since we are in the previous case

General case, from the changed node, up, you must fix violations.

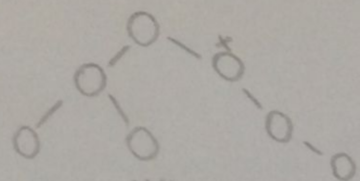
Suppose x is lowest node not AVL

AVL trees let every operation take O for an $\log h$

$h = \max(h_L, h_R) + 1$



This is how we update the height for a given node, we should do this for insertions and deletions



Not AVL because not balanced, the node x has height bigger on right than left of more than 1

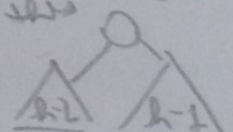
Invariant

$\forall$node, $|h_L - h_R| \leq 1$

is a BST

height is $O(\log n)$

If you want to build an AVL tree with the fewest nodes possible given a height, you'd do this



For this problem only

$N(h) = N(h-1) + N(h-2) + 1$

$N(h) \approx 1.6^h$ ← exponential is what matters

also phi (Φ)

(the formula looks like the fibonacci formula, with +1, so they are bigger)

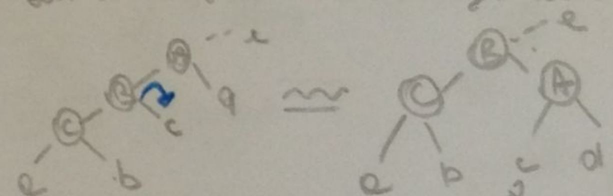
Insertion

- usual BST insertion going down path
- update the height of every node in the path

Deletion // similar

Rotation

You can left rotate or right-rotate



c got moved in such a way it is a BST, others are in same order

pseudo

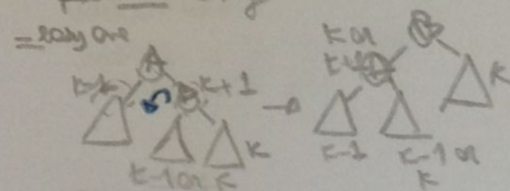
B.p = x
if x.r == A (pointer comparison)
x.r = B
else
x.l = B

B.r = A
A.p = B

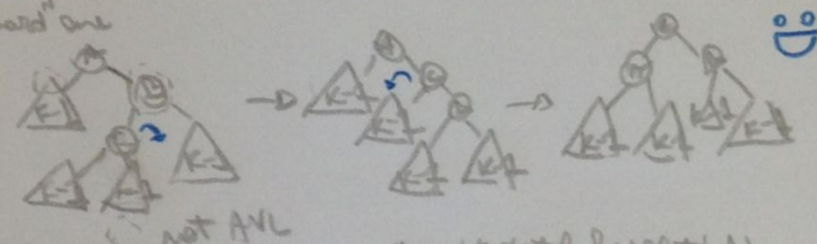
A.l = c
c.p = A

for a left rotation, swap r and l

Rebalancing



or reverse



not AVL

"restore height" update height(A) update height(B)

Lecture 7

Linear time sorting

comparison model, usually used for lower bound

- In that model, all input items are black boxes (ADT)
- The data structure supports only comparisons

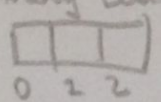
Every sorting algorithm seen so far is about comparisons

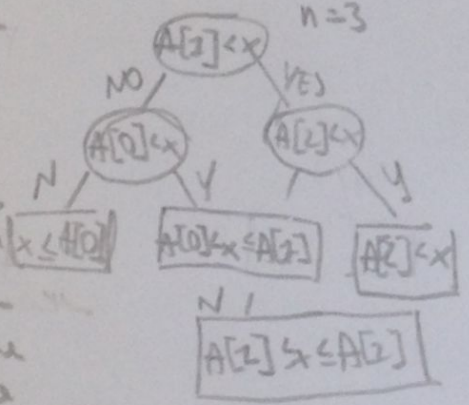
- time cost = # comparisons

Decision tree

Any comparison algorithm can be viewed as a tree of all possible comparisons and their outcomes, and resulting answer. For any particular n:

eg. binary search





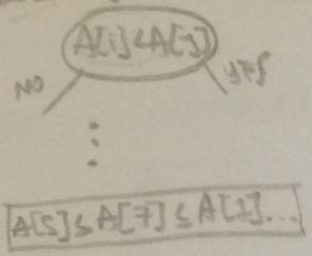
decision tree	algorithm
internal node	decision
leaf	found answer
root-leaf path length	execution
height of tree	running time
	worst case running time

search

Sorting lower bound

- proof
decision tree is binary and must have at least $n!$ leaves, one for each answer (can be more in other algorithms)
- then the height must be at least $\lg n!$, where n is the running time

sorting



proof decision tree is binary and # leaves ≥ # possible answers ($n!$)

→ height ≥ $\lg(n!)$
... most stuff
→ $n \lg n - O(n) = \Omega(n \lg n)$

leaf the number of comparisons must be $n \lg n$

RAM model, memory is an array, I can access anything in constant time

→ linear time sorting (integer sorting)

- assume the keys we're sorting are integers and they're positive. Each fits in a word (can be manipulated in constant time)
- can do more than just comparisons

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

→ $O(n+k)$

counting sort

3 5 5 7 5 3 6

3 3 5 5 5 6 7

- Allocate an array of counters for 1, run through in order and increment the key corresponding in the array element

3 5 7 5 3 6

If you want to sort these trees, we use lists instead of counters

Sorting $\Omega(n)$

insertion $O(n^2)$
merge $O(n \lg n)$
heap $O(n \lg n)$

} (comparison sorts $\Omega(n \lg n)$! (it's not the last time)

counting $O(n+k)$
radix

(not fast enough)

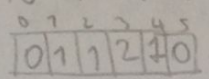
key (object):
2.x def __cmp__(self, other)
< 0
= 0
> 0

3.x def __lt__ (self, other) <
__le__ <=
__gt__ >
__ge__ >=
__eq__ ==
__ne__ !=

Counting sort

4 1 2 3 3

Makes assumption that the values goes from 0 to k-1 (k values)



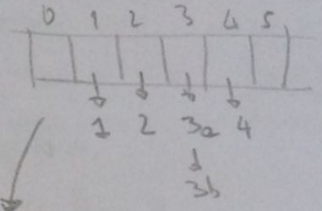
- Create array with 0s, then count numbers

1 2 3 3 4 - print out 0 output

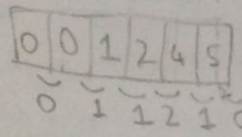
However we care about stability. If you have two keys that look equal to the algorithm (but they might be different objects), the one that shows up first in the input should also show up first in the output

4 1 2 3^a 3^b ← different objects

To solve stability I can implement a linked list at each array element




```
for i=n-1 to 0
    pos[i]=sum-pos[i]
    num=pos[i]
    out...
```



suppose we are doing this in C

- We set how many keys there are smaller than the searched value

Radix sort

A key is a sequence of digits

2 3 4 1
1 4 3 2
2 4 1 3
1 2 4 3
2 1 4 3
1 2 3 4 ← already sorted

d digits (per key)
Digit $\in 0 \ldots k-1$ (not base 10)
I can represent k^d numbers

- We're going to have d rounds, in each round we take all the keys and sort them according to one of the digits
- Start with least significant digit and move your way out

2 4 1 3
1 4 3 2 ← stable sorting: I care the 2nd to 3rd
2 2 3 4 ← I care the 1st to 2nd
2 3 4 1
1 2 4 3
2 1 4 3

As I go to the left I will see the sorted keys

These last three keys, they through stable sorting, they are also sorted according to last two digits

For each digit I run counting sort, $O(n+k)$,
so total is $O(dn+dk)$

$d n = O(n)$ → $O(n)$
$d = O(1)$
$k = O(n)$

HASHING (1)

Dictionary ADT

maintain a set of items, each with a key

insert(item) – (overrides any existing key)
delete(item)
search(key) – return item with given key or report doesn't exist

"In DST you can look up for next longer/smaller in databases you are not allowed to do this"

We can have as $O(1)$ with high probability

Python: dict
- D[key] - search
- D[key] = val - insert
- del D[key] - delete
- item = (key, value)

Motivation
- discord
- globalrose
- compilers and interpreters
- network routing servers
- substring search
- string commonalities
- file/dir synchronization

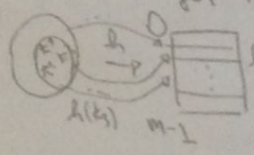
Simple approach

Direct access table
- store items in array indexed by a key
- associate with only an integer key

Bad:
- Keys may not be integers
- Gigantic memory hog

solution, pre-hash
- maps keys to a non-negative integer
- strings can be written as bits, like integers

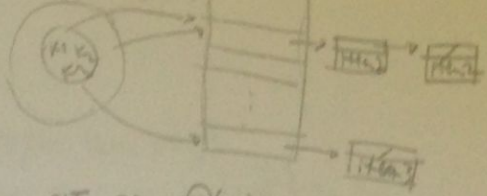
solution, hashing (hash into table and mix around)
- Reduce universe U of all possible keys (integers) to a reasonable size m for table



$m = \Theta(n)$ ← # keys in dictionary

If more keys map to slot by pigeonhole principle

Collision – if you have multiple items in the same slot, store them as a list



Unless you're really unlucky, the items will be distributed

worst case $\Theta(n)$ → All items in the same slot

proving that, with this assumption Simple Uniform Hashing, this is false, but useful for analysis

equality [for pair – each key is equally likely to be hashed to any slot of the table independent of other key hashing!] independently

Analysis:
expected length of list for n keys, m slots
$n/m = \alpha$ as the (load factor) load factor
$\Theta(1)$ if $m = \Theta(n)$
running time $= O(1+\alpha)$ (hash)

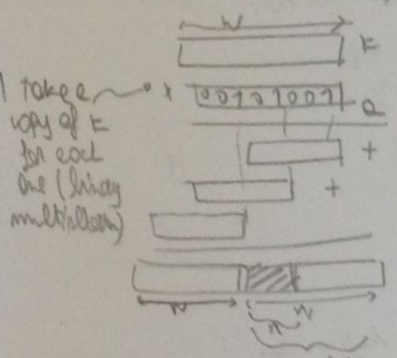
Hash functions

Division method: $h(k) = k \bmod m$
- Good only if m a prime or m not very close to a power of 2 or 10

(intuition why keeping nearby)

Multiplication method: $\lfloor h(k) = (a \cdot k) \bmod 2^w \rfloor \gg (w-r)$

w = word size



shift right

right word
(take the right word and shift right by w-r)

$m = 2^r$, r is between 0 and w-1

- a must be odd and not close to a power of two

UNIVERSAL HASHING

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

for worst case keys $k_1 \neq k_2$

$\Pr[h(k_1) = h(k_2)] = \frac{1}{m}$

a, b ← random between 0 and p-1

p → prime number > |U|

Analysis
total expected chain length
$n/m = \alpha$
$O(1+\alpha)$ for

How to choose m? ← O(n) to make space linear
- want $m = \Theta(n)$
  ↑ $\Omega(n)$ in order for α to be constant

To do this, start small: m = 10 ← for example
grow and shrink as necessary

if n > m: grow table → m → m'
- make new table of size m'
- build new hash h'
- rehash:
  for item in T
    T' insert(item)
  $\Theta(n + m + m')$

if m changes we need a new hash function

Table doubling

k inserts take $\Theta(k)$ time, is $\Theta(1)$ amortized
Also: k deletions take $\Theta(k)$ time

m' = m + 1 $\Theta(1+2+3 \ldots n) = \Theta(n^2)$ BAD!
m' = 2m
table doubling
$\Theta(1+2+4+8+\ldots+n) = \Theta(n)$

there are only a few operations that are expensive, the rest are constant. This idea is called amortization

- operations take T(n) amortized if k operations take ≤ k·T(n).
- Takes T(n) on average over all operations

Deletion
- if $m = \frac{n}{2}$ then shrink → $\frac{m}{2}$
But if I delete and insert ($2^k \to 2^k + 1$) we can get $\Theta(n)$ operations
BAD :(

- if $m = \frac{n}{4}$, then shrink → $\frac{m}{2}$
amortized time is $\Theta(1)$

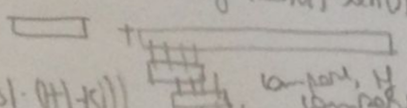
Python lists: are resizable arrays, each operation is amortized constant (append, pop);
they use table doubling

STRING MATCHING
Given two strings s, t: does s occur as a substring of t

e.g. what google does

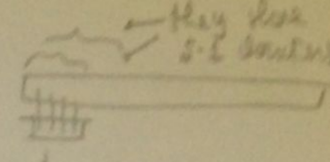
simple algorithm
any(s == t[i : i+len(s)]
  for i in range(len(t)-len(s)))

s = '6.006' t = inbox



$\Theta(|s| \cdot (|t| - |s|))$ compare, if match continue to compare, else shift one place.

Instead of checking the string every time we could check the hash function of the string.
Hashing s to a reasonable size and fit it in a word



Rolling hash

- given a hash value r, we'd like to be able to append a character and delete the first character in x
  r. append (c)
  character after s ←
  r. skip (c) ← first character
  r maintains the string x
- r() hash value of x = h(x)

If we can do all of this in constant, we can do string matching with the following!

KARP-RABIN ALGORITHM

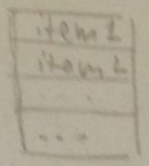
How to build this:
division method
h(k) = k mod p
  random prime

Treat x as multidigit number u in base a
  alphabet size

u = u·a + ord(c)
  append
u → u − c·a
  skip

Open addressing

Another approach for collision



one item per slot

$m \geq n$

m: # slots, n: # elements

Probing

We try to insert something, if you fail we compute a slightly different hash for that

This way, we continue to probe until we find an empty slot

$$h : U \times \{0 \dots m-1\} \to \{0 \dots m-1\}$$

U: keys universe; $\{0 \dots m-1\}$: Total count; produces a number from 0 to $m-1$

You ensure that you are using open addressing properly you want

$h(k,1)\ h(k,2) \dots h(k,m-1)$

arbitrary key k → must be a permutation of $0, 1 \dots m-1$

Neither there is some free slots, eventually the sequence of probes will allow you to insert into one

Insert (k, v) — Name flop

keep probing until an empty slot is found

Search (k)

If any of the slot encountered are not k, keep probing until either encounter key or find an empty slot

Delete (k)

Comes search to find empty slot, so we replace delete item with 'delete me' flag, so that used can still work. (Insert will now overwrite delete me flags)

Probing strategies

ordinary hash function

Linear probing: $h(k,i) = (h'(k) + i) \bmod m$

clusters, consecutive groups of occupied slots, which grows whenever an insert encounters it ✗

permutation ✓

Double hashing

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

If $h_2(k)$ is relatively prime to m → permutation ✓ good ✓

Uniform hashing Assumption

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

$\alpha = \frac{n}{m}$ cost of operation $\leq \frac{1}{1-\alpha}$

Graph search

"explore" a graph

Applications

- web crawling
- social networking
- network broadcast
- ...

graph G(V, E)

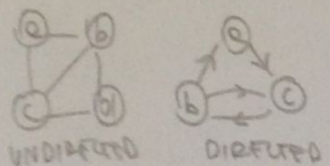
V = set of vertices

E = set of edges

UNDIR ↳ unordered pairs e {v, w}

DIR ↳ ordered pairs e (v, w)

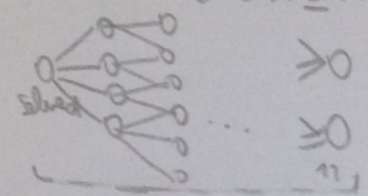
↗ Array representation is really terrible and expensive



E = {{a,b}, {b,c}, {c,d}, ...}   E = {(a,c), (b,c), (c,b), (b,a)}

Pocket cube (2×2×2)

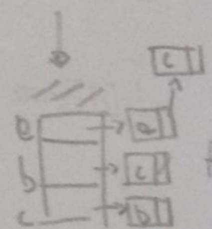
- configuration graph
- vertex for each possible state (# vertices 8! · 3^8)
- edge for each possible move   O–O undirected



diameter

Breadth First search

- Visit all nodes reachable from given s ∈ V
- O(V+E) time
- Look at nodes reachable in 0, 1, 2, 3 moves [is Adj[s]]
- Careful to avoid revisiting vertices



Graph representation

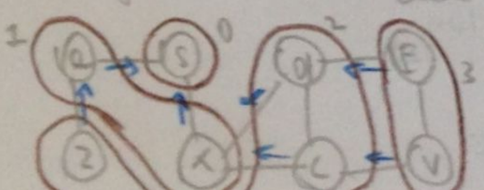
Adjacency lists:

array Adj of |V| linked lists

- For each vertex v ∈ V Adj[v] stores its neighbours (could be array, hash table, objects, so on)

$\Theta(V+E)$ space required

↘ code next page



shortest paths

- v ← parent[v] ← parent[parent[v]] ← ... is the shortest path from s to v. Its length is level[v]

```
BFS (s, Adj):
  level = {s : 0}
  parent = {s : None}
  i = 1
  frontier = [s]
  while frontier:
    next = []
    for u in frontier:
      for v in Adj[u]:
        if v not in level:
          level[v] = i
          parent[v] = u
          next.append(v)
    frontier = next
    i += 1
```

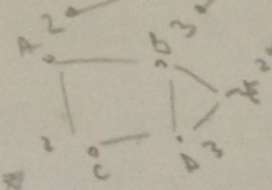
$\sum_{v \in V} |Adj[v]| = \begin{cases} 2|E| & \text{undirected} \\ |E| & \text{directed} \end{cases}$ ← Handshaking Lemma

O(V+E)

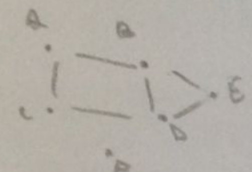
↳ you touch every vertex once when added to level

graph

degree, number of edges.



connected, there's a path from any node to any other node

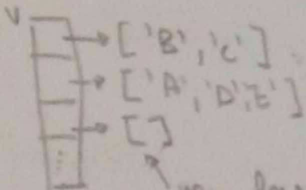


unconnected.

...

If you add up all E edges, you get 2E. Handshaking Lemma

Python graph

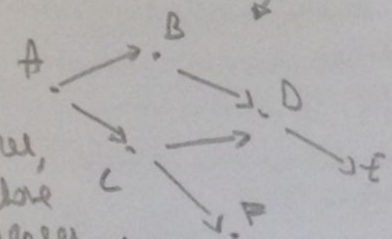


Adjacency lists

you have empty lists, I have to pay for it. ~O(V)
Better to not store anything ~O(1)

O(V + E)
↑
assuming empty lists

list neighbors
O(deg(v))



degree, you have indegrees and outdegrees only.

outdeg (A) = 2
indeg (A) = 0

↳ the # of indegrees and outdegrees are equal

Adjacency matrix

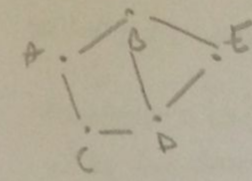
	A	B	C	D	E
A	1	1	1	0	0
B	1	1	0	1	1
C	1	0	1	1	0
D	0	1	1	1	1
E	0	1	0	1	1

2E + V

list neighbors
O(V)

BFS

we have a graph



[A]
B C
↓
A E D
→ already visited
already visited from B

For visited vertices we use dictionaries
seen [v] → True

[A, B, C, D, E] as a queue

O(V · V)
↑ visit each node once
adjacency matrix

O(V + E)
↑ adjacency list
O(E)
↑ work in python you don't need to initialize when using a dictionary

We want to find the shortest path from starting node (A in example)

DFS used to explore the whole graph (not only the part reachable from s, like BFS)

- Recursively explore the graph, backtracking as necessary
- careful not repeat vertices

```
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)
```

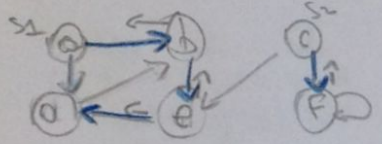
↳ Iterating over outgoing edges from s

top-level

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

↳ iterating over all of s

"we have a vertex s, look at all outgoing edges. For each edge we check whether I already visited it (by seeing parent)"



Analysis

$\Theta(V+E)$

- visit each vertex once in DFS alone $\Theta(V)$
- DFS-Visit called once per vertex v
  - ↳ Pay $|Adj[v]|$
  - ↳ $O(E)$ combining lemma for directed graphs

"Tree edges and backward edges can also exist in another unguided graph"

"an edge can only be of one type"

Edge classification

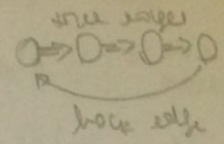
- tree edges (parent pointers) visit new vertex via that edge. They form a tree, like arrows on ex.
- forward edges: goes forward along the tree (like a→d)
- backward edges: goes to an ancestor in the tree (like d→b)
- cross edges all other edges

Edge classification are handy for cycle detection and topological sort

Cycle detection

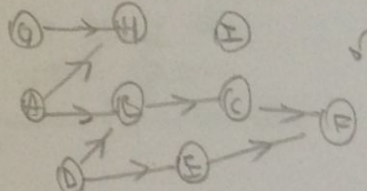
G has a cycle iff DFS has a back edge

Proof



Job scheduling

We are given a directed acyclic graph. I want to order vertices such that all edges point from lower order to higher order

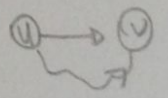


Things to do in order

topological sort: Run DFS and output the reverse of finishing times of vertices

For any edge $e=(u,v)$, v finishes before u finishes.

case 1. u starts before v



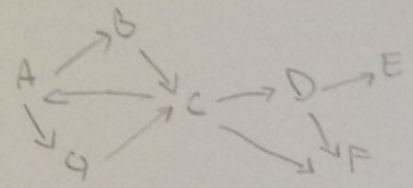
writes u before u finishes

case 2 v starts before u

v finish before visit u

adjacency list

```
{ 'A' [ 'B', 'G' ]
  'B' [ 'C' ]
  'C' ...
  'D'
  'E'
```



class G

al = {...}

vertices():
  return al keys

neighbors(v):
  return al[v]

DFS result
~~visited#[v] = True~~
parent[v] = some other node

DFS(g, s): (g = graph, s = starting point)
  r = DFS result()
  r.parent[s] = None
  DFS_visit(g, s, r)

DFS_VISIT(g, v, r); r.visited[v] = True
  for n in g.neighbors(v)
    if n not in r.parents
      r.parent[n] = v
      DFS_VISIT(g, n, r)

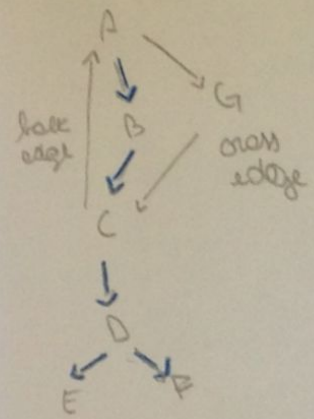
DFS('A')
DFS_VISIT(A) B G
DFS_VISIT(B) C
DFS_VISIT(C) A D F
DFS_VISIT(D) E F
DFS_VISIT(E)
DFS_VISIT(F)
DFS_VISIT(G) C

parents: 'A': None, 'B': 'A', 'C': 'B', 'D': 'C', 'E': 'D', 'F': 'D', 'G': 'D'

not in parents, we must visit it

tree edges (connect to of diff types)

dfs order



if DFS follows it: tree edge
if v descendant v: forward edge
else if v ascendant v → back edge
else cross edge

---

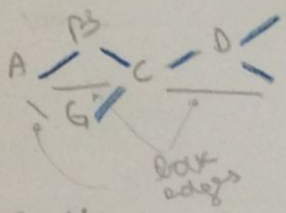
topological sorting !

graph of class prerequisites
like 6042 for 6.006

You can't have a cycle because you wouldn't have a dependency there
(In the graph in the picture page, you cannot start from A because you need C first)
must be directed

DAG (Directed, acyclic, graph)

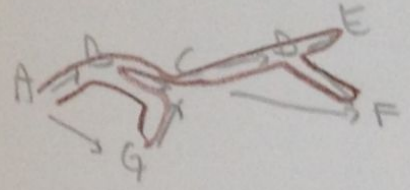
For an undirected graph



I can't have cross edges and forward edges

(when you're at an edge while DFS, in a directed graph you also see the forward edge later. In directed you see that immediately and you therefore make it as backward edge)

(I only edges are also seen before, and are marked as tree edges)



E F D C B G A

Shortest path

$G(V, E, W)$ — $W: E \to \mathbb{R}$

weight (maps edges to weights)

Bellman-Ford

$O(VE)$ works on positive and negative weight edges

Dijkstra

non-negative weight edges

Most of the times E dominates. Dijkstra is a nice algorithm because it is linear on E

$O(V \log V + E)$

$E = O(V^2)$ size

"W doesn't exist in the complexities!"

Weighted graphs

$v_0 \xrightarrow{p} v_k$

path from v_0 to v_k

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

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

"a path is a sequence of edges, each of those must be in the graph"

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

"sum of the weight of edges"

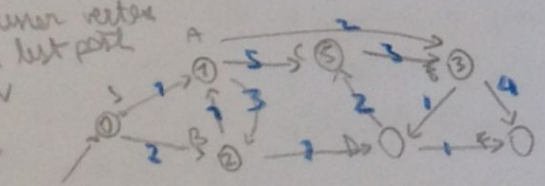
Find p with minimum weight!

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

"Initially everybody is infinity, starting point is 0, you try to reduce those infinities for all reachable vertices

$\pi(v)$ predecessor vertex on the last path to v

$\pi[s] = \text{NIL}$

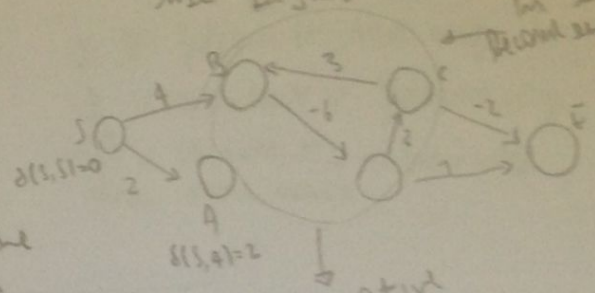


$\delta(v)$ weight (shortest weight to get there)

becomes delta and you're done

then iterate from the first point to find shortest path

Negative weights why? Leave hell label network



negative weights are a problem because of negative cycles that makes the vertex in the cycle indeterminate

Negative cycle: you can get to B with weight -1, there is no more a lower bound of 0

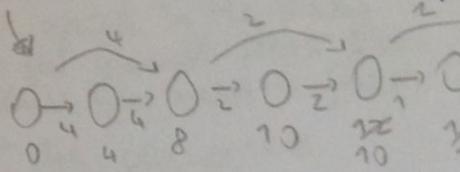
Bellman Ford computes δ for all positive vertices and marks the others as indeterminate

Initialize for $v \in V$ $d[v] = \infty$ $\pi[v] = \text{NIL}$

$d[s] = 0$

Repeat select edge (u,v) → visit all edges here $d[v] \le d[u] + w(u,v)$

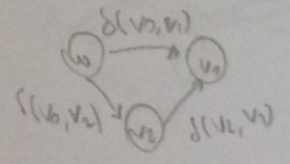
Relax edge (u,v) → means (if $(d[v] > d[u] + w(u,v))$ $d[v] = d[u] + w(u,v)$ $\pi[v] = u$



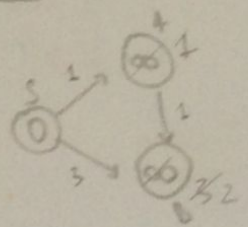
exponential relaxations the problem was, this algorithm is not good

Optimal substructure

- subpaths of the shortest path are shortest paths



DIJKSTRA



$d[v]$ = length of the current shortest path from s

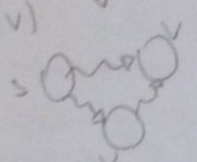
$\delta(s,v)$ = length of a shortest path

$\pi[v]$ = predecessor of v on shortest path from s to v

Relax (u,v,w)
if $d[v] > d[u] + w(u,v)$
$d[v] = d[u] + w(u,v)$
$\pi[v] = u$

Lemma
The relaxation algorithm maintains the invariant that $d[v] \geq \delta(s,v)$ for all vertices
(you'll never get something that is less than your $\delta(s,v)$)

Proof. by induction on the # steps. Assume
$d[v] \geq \delta(s,v)$ (for use)
By Δ inequality $\delta(s,v) \leq \delta(s,u) + \delta(u,v)$
$\delta(s,v) \leq d[u] + w(u,v)$



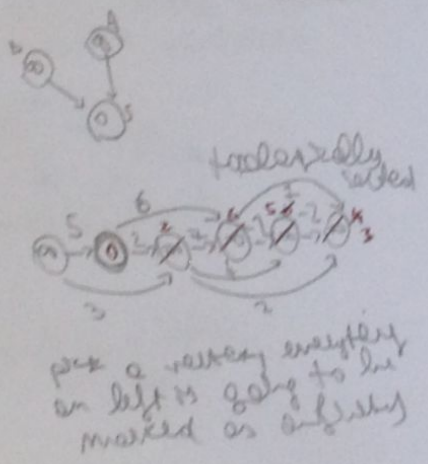
Δ inequality
the best way could from s to v, or could pass through u.

DAGs
can't have (+ve) cycles
(can have negative edges)

1) Topological sort DAG
path from u to v implies u is before v in ordering

$O(V+E)$ time

2) One pass over vertices in topologically sorted order relaxing each edge that leaves each vertex



By taking the demo and pull it up (for Dijkstra).
String tense on top are the predecessors.
If you take another vertex the order will still be right

Dijkstra (g, W, s)
Initialize (g, s)
$s \leftarrow 0$
$V[s]$
as we process vertex from q, we move them to explored

$d[s]=0$

just picks the min from Q and processes it

that makes it greedy

Q is a priority queue, pulling out min values

I'm sure I know the shortest path

```
while Q ≠ 0:
    u ← extract-min(Q)
    delete u from Q
    S ← S ∪ {u}
    for v ∈ Adj[u]
        Relax(u, v, w)
```

Speeding up Dijkstra

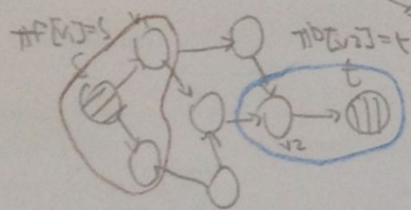
"Not overall complexity but average case"

We're solving single source to any/all destinations (single source single target) s → t → If you have a target t

Dijk. pseudo: Initialise () ← d[s]=0 d[v≠s]=∞
Q ← V[G]
while Q ≠ 0
do u ← extract_min(Q) ← (stop if u == t)
for each vertex v ∈ Adj[u]
do Relax (u, v, w)

↑ update priority, modifies parent pointers

Bi-directional search → 1 step standard Dijkstra from s, 1 step backward Dijkstra from t



We have to make another priority queue

$d_f[s]=0$ $d_b[t]=0$

↑ everything else is infinite

distance in forward search — distance backward search

Edges must be traversable backward also (directed graph)

Conclude search when frontiers meet, specifically?

When you pull out a node from Qf and Qb, finish the search

e.g.: If w was processed first from both Qf, Qb

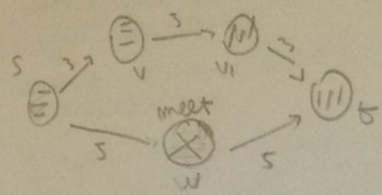
find sp using πf from s to w.

Find sp using πb from t to w

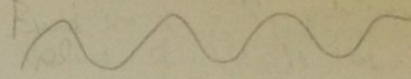
Qf ← priority queue forward — Qb ← priority queue backward

πf — predecessors forward; πb — predecessors backward

w may not be in sp



Gives d_b and d_f of 10, passing from w



Find a v (any from w) with minimum $df[v] + db[v]$

↳ After finishing search, check other two vertices

$df[v] + db[v] = 3+6 = 9$

$df[v_1] + db[v_1] = 6+3 = 9$

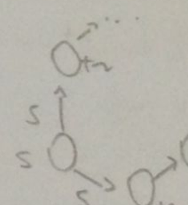
one less than 10 (value found)

Directed search

Modify edge weights with potential functions

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

↑ new weight



steered to go downhill and terminate quicker

Correctness:

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

↑ any path from s to t will be shifted of this amount. You will discover it faster

Landmark $l \in V$

precompute $\delta(u, l)$

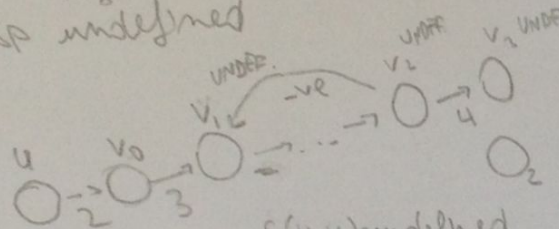
$\forall u \in V$ → for any vertex, find sp to l

$\lambda_t(u) = \delta(u, l) - \delta(t, l)$

With the correct choice of u, Dijkstra will run faster

Bellman-Ford

- Greedy wh negative edges, in the case of negative cycles, this also returns sp undefined



δ(u,v1)=undefined
δ(u,2)=∞
δ(u,v0)=2

Generic sp algo

InitSingleSource For v∈V d[v]←∞ π[v]←Nil
d[s]←0

Main repeat, select edge [somehow]
Relax(u,v,w) ← sets π pointers to be correct
Until you can't relax any more

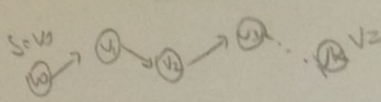
Bellman-Ford(G,w,s)
Initialise
for i=1 to |V|-1
  for each edge(u,v)∈E
    Relax(u,v,w)
for each edge(u,v)∈E ← If you can relax edges report that -ve cycle exists
  if d[v]>d[u]+w(u,v)

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

O(VE)

theorem: if G does not contain -ve weight cycles, then BF after execution returns d[v]=δ(s,v) for all v∈V

corollary: If a value d[v] fails to converge after |V|-1 passes, there exists -ve wt cycle reachable from s



p: v1, v2, v3 ... vk
k ≤ |V|-1
↳ else cycle

theorem (proof)

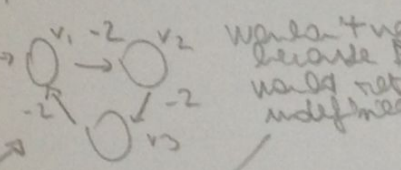
Let v∈V p=⟨v0,v1,v2,vk⟩ v0=s to vk=v.
This path p is a shortest path with min # edges
No -ve weight cycles ⇒ p is simple ⇒ k≤|V|-1

- After 1 pass through E, we have d[v1]=δ(s,v1), because we will relax (v0,v1)
- After 2 pass, ... d[v2]=δ(s,v2) because in 2nd pass we relax (v1,v2)
- After k passes, d[vk]=δ(s,vk)
  |V|-1 passes ⇒ all reachable vertices have δ value

After |V|-1 passes we find an edge that can be relaxed
⇒ current sp from s to some vertex v is not simple because I have a repeated vertex. Found a cycle, that is -ve

corollary proof

Longest path

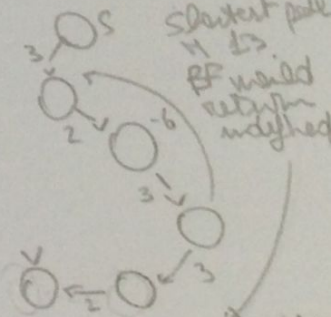


Make weights -ve and run Bellman-Ford

Won't work because BF would return undefined

It is also NP-hard

There is no algorithm better than exp time



Shortest path is its. BF would return undefined

We have a -ve weight cycle, finding the sp is a NP hard problem

Dynamic Programming

general, powerful way to design algorithm

DP = careful brute force

↓

to get polynomial time algorithm

DP = "subproblem + "re-use"

take a problem split into subprob. solve them re-use the solutions

"Why it is called DP? Was invented by Richard Bellman to hide the term math research and sounded cool!"

Fibonacci

$F_1 = F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

↑ find it

Naive rec algo

```
fib(n):
    if n <= 2: f = 1
    else: f = fib(n-1) + fib(n-2)
    return f
```

EXPONENTIAL

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

$\geq F_n \approx \varphi^n$ golden ratio

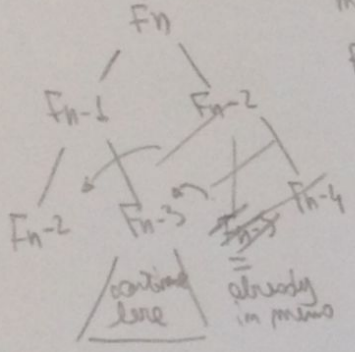
$T(n) \geq 2T(n-2)$

$= \Omega(2^{n/2})$

Memoization ← make IF better!

```
memo = {}

fib(n)
    if n in memo:
        return memo[n]
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```



fib(k) only recurses the first time it's called $\forall k$

- memoized calls cost $\Theta(1)$
- # of non-memoized calls is n: fib(1), fib(2) ... fib(n)
- non recursive work per call = $\Theta(1)$

time = $\Theta(n)$

DP - memoize (remember) & re-use solutions to subproblems that help you solve the initial problem

time = # subproblems · time per subprob

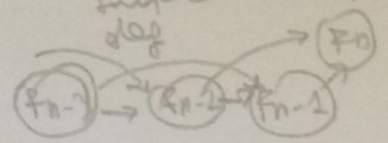
Bottom-up DP algorithm

```
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]
```

Does exactly the same computations of the memoized algorithm

General one

- same computations as memo
- topological sort of subproblems dependency dag



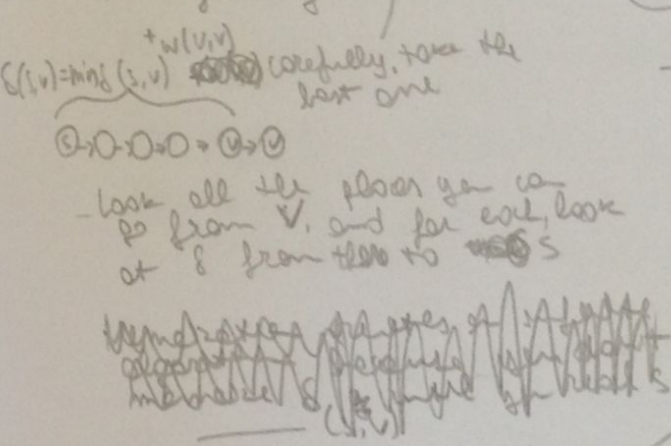
- can often save space (can need the last two values)

Shortest path $\delta(s,v)$ $\forall v$

guessing: try all guesses

DP = recursion + memoization + guessing

$\delta(s,v) = \min(\delta(s,u) + w(u,v))$ carefully, take the best one



- Look all the places you can go from V, and for each, look at δ from there to s

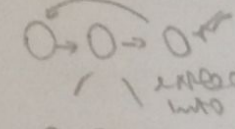
Memoization: infinite time on graphs with cycles. (because memo is not set)

On DAGs runs in $O(V+E)$

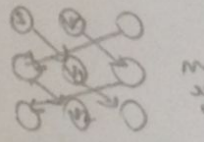
time for subproblem = indegree(v) + 1

$\sum_{v \in V} (\text{indeg}(v) + 1) = \Theta(E) + V$

cycle graph to acyclic



unroll it into layers



make go edges from one layer to another

subproblem dependencies are acyclic (topological sort like for DAG in Bottom-up)

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

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

subproblems = V^2

$\Theta(V \cdot E)$

DP = careful bruteforce   Review

DP ≈ guessing + recursion + memoization

- guessing ← I'll try all possibilities for a choice and take the best one
- recursion ← in polytime-ish, usually exponential → memoization: then it becomes polynomial

DP ≈ shortest paths in some DAG

↑ dynamic programming is the solution that that problem, given that problem is any DP problem

time = # subproblems · time/subproblem

---

5 easy steps to DP (~~not to apply in that order~~)

1. Define subproblems ← # subproblems for analysis
2. Guess (part of the solution) ← # choices for guess
3. Relate subproblem solution ← time/subproblem
4. Recurse & memoize OR build DP table bottom-up ← check recurrence is acyclic (i.e. has topological order)
5. Solve original problem

Text justification

split text in good lines

text = list of words

If you do all your own on most of the line, most of the spaces, ones will be even larger, so this is [illegible]

badness(i, j) — how bad it is to use words(i : j) as a line

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

1. the subproblems are suffixes (words[i:])
   # subprob: n
2. where to start next line given
   # choices ≤ O(n)
3. recurrence: DP(i) ← solution to subproblem
   DP(i) = min(badness(i, j) + DP(j) for j in range(i+1, n+1))
   first word → ...
   next word → ...
   time/subp = O(n)
4. topol. order: n, n-1 .. 0
   total: Θ(n²)
5. orig prob: DP(0)

parent pointers
remember which guess was best
parent[i] = argmin(..) = j
0 → parent[0] → parent[parent[0]]
1st line, 2nd line, 3rd line

Blackjack

- deck = c0, c1 ... cn-1
- 1 player & dealer
- $1 bet/hand

1. subproblem: suffix ci: (new hand) → # subp. is n
2. Guess how many times should I hit → # choices ≤ n
3. Recurrence BJ(i) = max(outcome ∈ {-1, 0, 1} + BJ(j) for j in range(i+4, n) if valid play)

DP.

Subproblems for strings/sequences

- suffixes $x[i:]$ $\forall i$
- prefixes $x[:i]$ $\forall i$
- substrings $x[i:j]$ $\forall i \le j$ — $\Theta(n^2)$

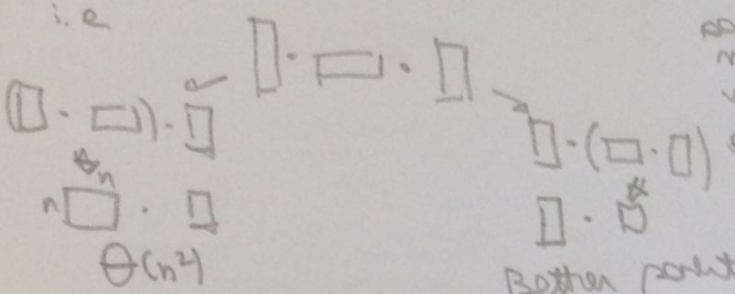
① in DP procedures, finding subproblems is usually the hard part in DP programming

Parenthesization

optimal evaluation of associative expression

$((A_0 \cdot A_1) \cdot \ldots A_{n-1})$ — when doing matrix multiplication, parenthesis make it more efficient so we use DP to find out what is the better parenthesization

i.e.



We start with guess

② the outermost multiplication

$(A_0 \ldots A_{k-1}) \cdot (A_k \ldots A_{j-1})$ — $O(j-i+1) = O(n)$ # of possible j's

substring $(x[i:j])$ — cost multiplication

③ Recurrence of $DP(i,j)$ is

① subproblem = optimal evaluation of $A_i \ldots A_{j-1}$ — # subprob is $O(n^2)$

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

time/subprob = $O(n)$ ...

④ time = $\Theta(n^3)$

BEST WE CAN DO FOR KNAPSACK — & not bad if S is small — pseudopolynomial — Time = $\Theta(n \cdot S)$ not polynomial

Edit distance

Given two strings x & y, what's the cheapest way to convert x to y through character edits:

- insert c
- delete c
- replace c → c'

{ the cost of each of these operations varies depending on the char. For replace, there's a matrix

Longest common subsequence

HIEROGLYPHOLOGY
MICHELANGELO

e.g. good for DNA comparison
C → G (cheap because common mutation)
C → A (more unlikely, high cost) — you consider how similar they are evolutionarily

You can model it as an edit distance problem with

- cost insert/delete = 1
- cost replace $\{0$ if $c = c'$, ∞ else

① subproblem

edit distance on $x[i:]$ & $y[j:]$ $\forall i,j$

subprobs. $= \Theta(|x| \cdot |y|)$

② Guess

- replace $x[i]$ with $y[j]$
- insert $y[j]$
- delete $x[i]$

③ Recurrence

$DP(i,j) = \min($ 1. cost of replace $x[i] \to y[j]$ + $DP(i+1, j+1)$), 2. ... (cost of insert $y[j]$ + $DP(i, j+1)$), 3. ... (cost of delete $x[i]$ + $DP(i+1, j)$))

④ topo. order — suffixes from smaller to bigger

for $i = |x| \ldots 0$
  for $j = |y| \ldots 0$

0 … j … |y| ; i ; |x| — we go towards the origin

⑤ $DP(0,0)$

time = $\Theta(|x| \cdot |y|)$

BEST ALGO FOR EDIT DISTANCE

KNAPSACK

You have

- List of items each has size s_i and personal value v_i — integers

"you can only bring what you can fit in your knapsack"

knapsack size S

max sum of values for a subset of items of size ≤ S

② Guessing if i is in subset or not

$DP(i, S) = \max(DP(i+1, X), DP(i+1, X - s_i) + v_i)$

① subprob = suffix $x[i:]$ of items and remaining capacity $X \le S$

$= \Theta(n \cdot S)$