Introduction to Algorithms

Peak finding (toy problem)

1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i

find a peak.

peak → 2 is a peak if $b \geq a$ & $b \geq c$

9 → $i \geq h$

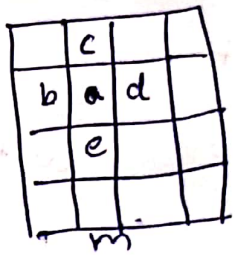
1 → $a \geq b$

1) Straight forward
- start from left, traverse all values
O(n).

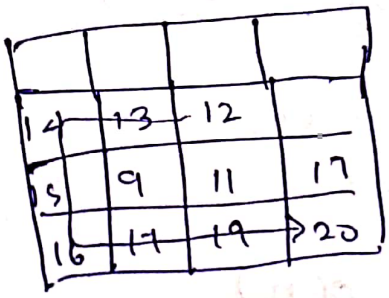
2) Divide & conquer
- look at n/2...
$O(\log_2 n)$

2D version

1) greedy ascent



a → 2D peak. if
$a \geq b$,
$a \geq d$
$a \geq c$
$a \geq e$.



- make a choice where to start

O(nm)

2) divide & conquer
- pick mid col j = m/2.
find 1D peak (i, j)
Use (i, j) as a start to find 1D peak on row i

O(n²) for square

→ Incorrect

Another method.

- pick mid col $j = m/2$.
- Find global max on col j at (i, j)
- Compare $(i, j-1)$, (i, j), $(i, j+1)$
- Pick suitable col.

$$T(n, m) = T(n, m/2) + O(n)$$

(for max)

$$= O(n \log_2 m)$$

Models of Computation

- Random access model
- pointer model

Python model

1) list → array L[i] = L[j] + 5 O(1)

2) OOP with O(1) attributes

x = x.next O(1)

L.append() → O(1)

(table doubling)

x in L → O(n)

len(L) → O(1)

L.sort() → O(n log n)

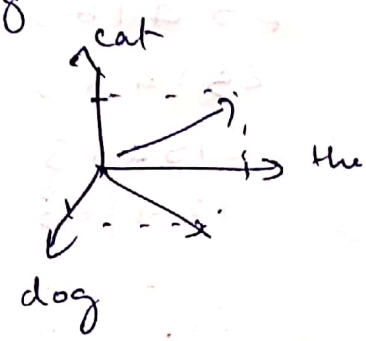
Document distance

$d(D_1, D_2)$

- document indexed as $D[w]$ (word)

D_1 = 'the cat'

D_2 = 'the dog'



- dot product

$$D = D_1 \cdot D_2$$

- cosim sim.

$$= \frac{D_1 \cdot D_2}{|D_1||D_2|}$$

Algo

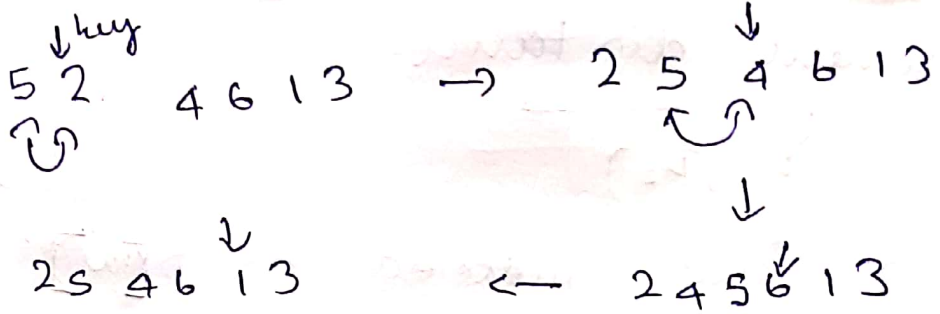
1) split doc int words $O(n)$
2) Compute word frq $D[w]$ $O(n)$
3) Compute dot product

Sorting

Insertion sort:

For i in 1 to n

insert $A[i]$ into sorted array $A[0:i-1]$ by pairwise swaps down to the correct position



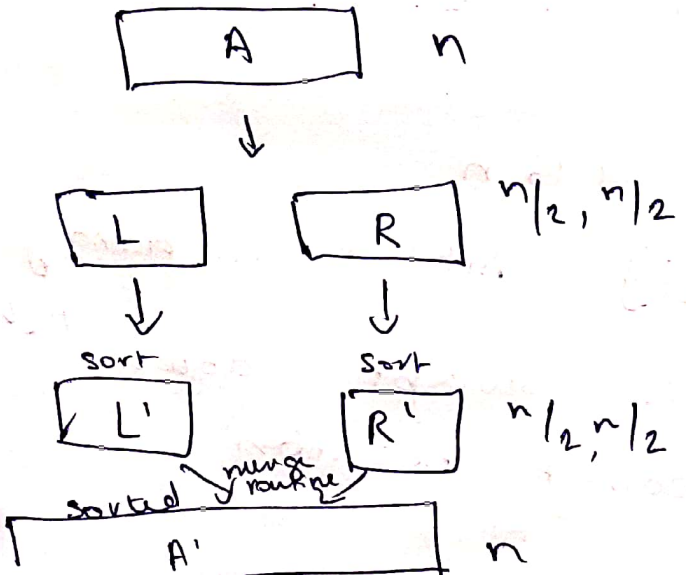
2 5 4 1 6 3
2 5 1 4 6 3
2 1 5 4 6 3
1 2 4 5 6 3
1 2 4 5 3 6
1 2 4 3 5 6
1 2 3 4 5 6.

O(n) key positions
Each step is O(n) compars & swaps
[So $O(n^2)$ algo]

complexity for compares & swaps are roughly the same for most cases.

- Do Binary search instead of pairwise swaps as 0:i-1 is already sorted. - Binary Insertion sort.

Merge Sort



Merge: Two sorted arrays as i/ps.

L R

20 12 → formed by recursive tree

13 11

7 9

2 1

Compare 2 numbers and see which is smaller (two finger algo)

20 12

13 11

7 9

2

merge → O(n)

20 12 and so on.

13 11

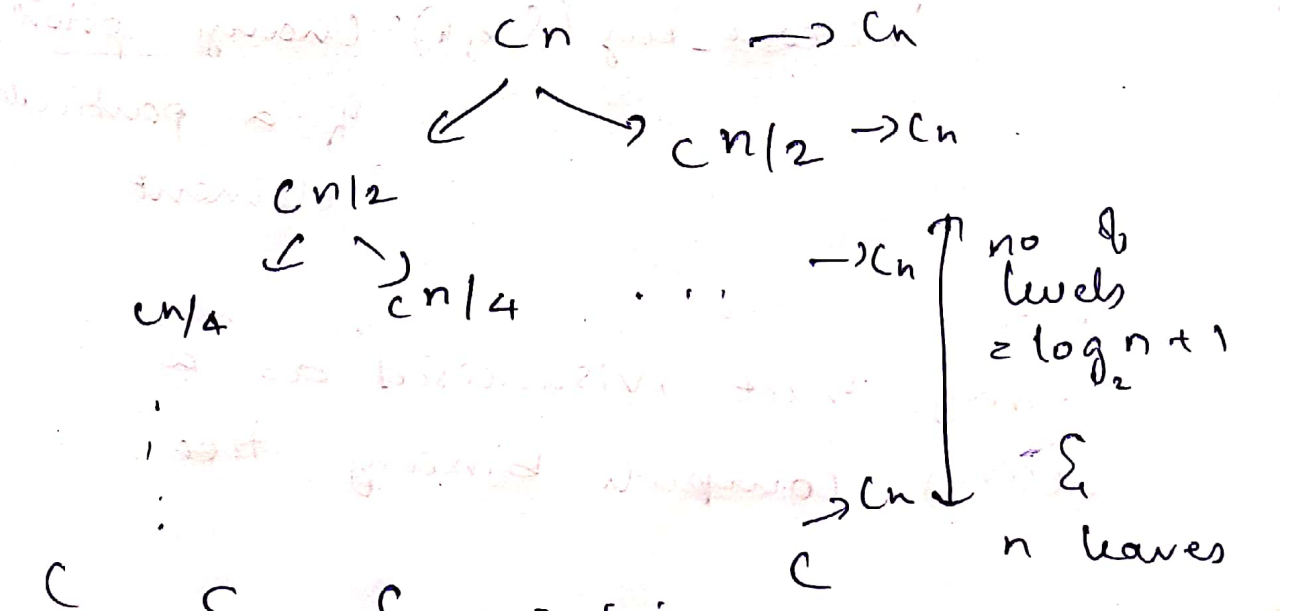
7 9

2 1

Complexity

$T(n) = c_1 + 2T(n/2) + c \cdot n$

c_1: dividing the array; $2T(n/2)$: recursive; $c \cdot n$: merge



$\therefore T(n) = (1 + \lg n)\, cn$

$= O(n \lg n)$

In merge sort $\Theta(n)$ auxillary space needed.

In-place sort has $O(1)$ space.

Heaps

Priority queue:

implements a set S of elements, each elem is associated with a key.

O/ps on priority queue:

Insert (S, x) : insert x into set S

max (S) : return elem of S with largest key

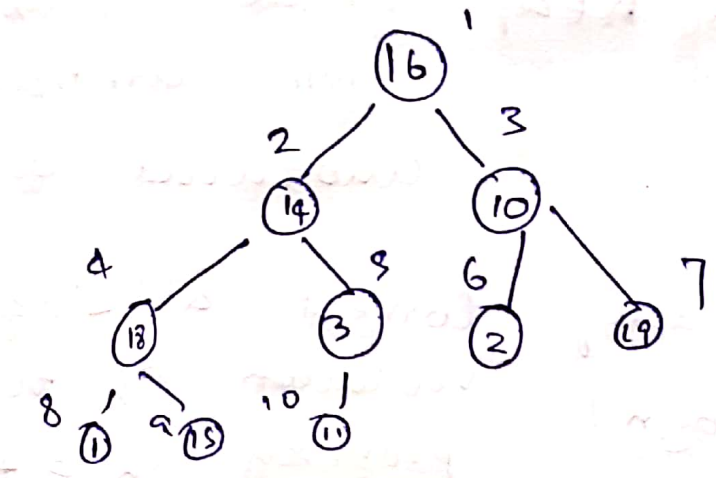
extract_max (S): remove ~~max~~ elem ~~key~~ or elem with largest key

Increase_key (S, x, k): Change priority of a particular element

Heap:

array struct, visualised as a nearly complete binary tree.

1	2	3	4	5	6	7	8	9	10
16	14	10	18	3	2	19	1	15	11



Heap as a tree

root of tree → first elem i=1

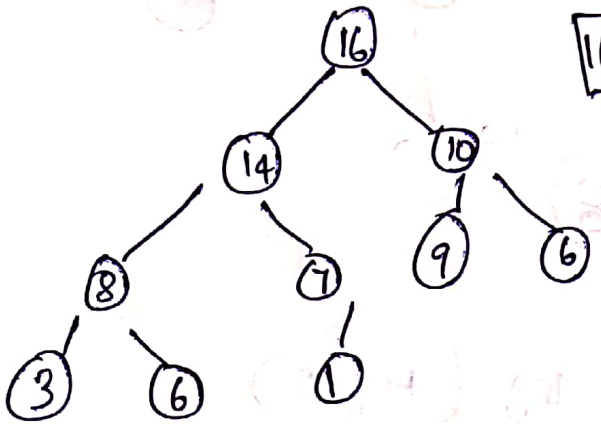
parent of i = i/2.

left (i) = 2i

right (i) = 2i+1

Max heap property:

key of a node ≥ keys of children



16	14	10	8	7	9	6	3	6	1

16 > 14, 10

14 > 8, 7 etc.

Similarly, min heap property

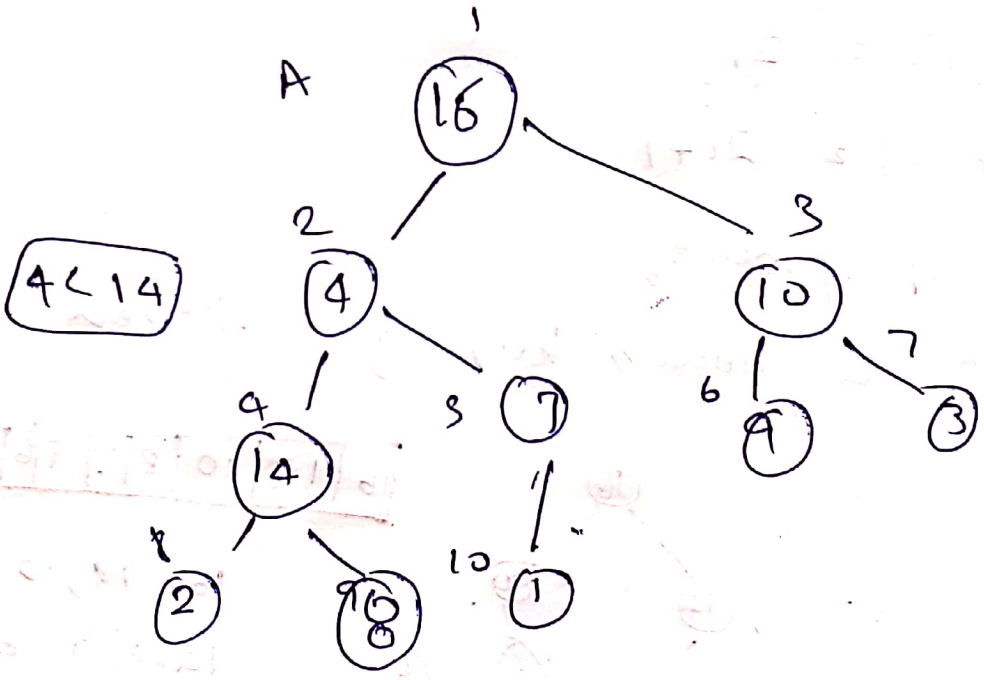
max (s) trivial to perform with max heap

Heap operations

Build max heap: creates max heap from arbitrary / unordered array.

max_heapify: Correct a single violation of heap property in a ~~subtree~~ & subtree's root
O(log n)

Assumption that ~~trees~~ rooted at left(i) & right(i) are max heaps.



max-heapify (A, 2)

Subtrees of 2, are max heaps.

- exchange A[2] & A[4]
(exchange with biggest child)

Then we observe max heap property isn't observed at 4.

∴ max-heapify [A, 4]

Then exch any A[4] & A[9].

- Convert array to max heap

Step1: Build max heap (A):

for i = n/2 down to 1
do max-heapify (A, i)

(A[n/2+1] ... A[n] are all leaves)

Complexity of build max heap:

observe max-heapify takes O(1) for nodes that are one level above leaves & in gen O(l) time for nodes that are l levels above.

n/4 nodes with 1, n/8 with level 2, ..., 1 node at log n level

Total amount of work

$= n/4 (1*c) + n/8(2c) + \ldots 1(\lg n\, c)$

Set $n/4 = 2^k$

$= c\, 2^k \left(1/2^0 + \frac{2}{2^1} + \frac{3}{2^2} + \ldots + \frac{(k+1)}{2^k} \right)$

the series is bounded by constant.

$\therefore = C 2^k \cdot C'$

$\rightarrow O(n)$

Step 2: find max elem A[1] O(1)

Step 3: swap elem A[n] with A[1] O(1)

(n swaps)

now max is at end of array.

4: discrade node n from heap by decrementing heap size to n-1.

5: new root may violate max heap property but children are max heaps. -> max_heapify to fix this O(lg n)

$\downarrow$

$O(n \log n)$

Binary Search Tree

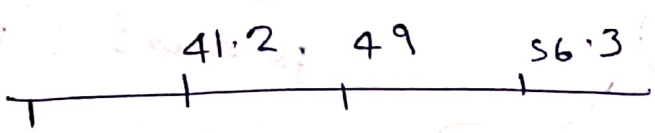
Runway reservation system:

- Airport with singa runway
- reservations for futun landings
- reserve request - landing time t
- add t to set R of landing tim

if no other landings are scheduled within K minutes

Remove from R when plan lands

How to do this in logn time?



request for 53 ✓

44 ✗

20 ✗ (passed)

different approaches:

1) Unordered list/array:

~~everything~~ almost everything is linear.

O(n)

2) sorted array:

can use binary search

logn for search

const for comparision

insertion will require shifting

↳ ~~log~~ O(n)

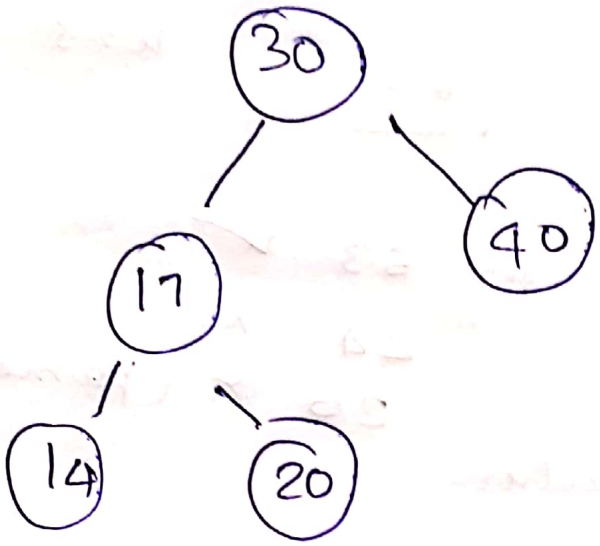
3) sorted list:

Insertion is const time,

but can't do binary search on list

* Heaps:
search for elem and comparison like tech O(n)

BSTs



node x : key(x)

pointers : parent(x)
left(x)
right(x)

For all nodes x, if y is in the left subtree of x

$key(y) \leq key(x)$

and if y is in the right subtree, then

$key(y) \geq key(x)$

(40 > 30)

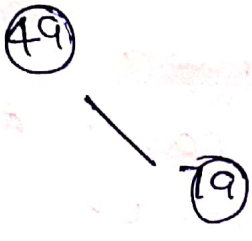
(17 < 30
14 < 30
20 < 30)

Insert

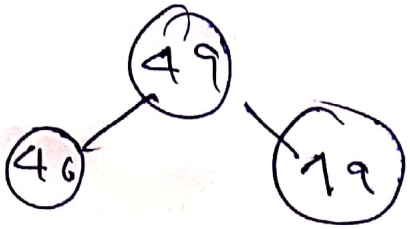
Insert 49.

Insert 79.

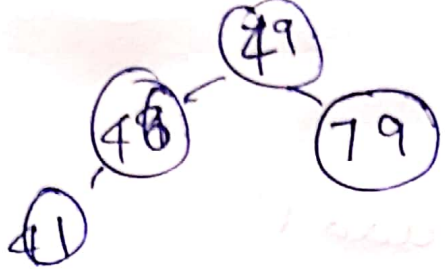
79 > 49 ∴ right



46 →



41



Insert 42, k=3

Compare 42, 49, ✓
42, 46 ✓
42, 41 ✗

h → height of tree.

Insertion with a c/o check is O(h)

other o/ps

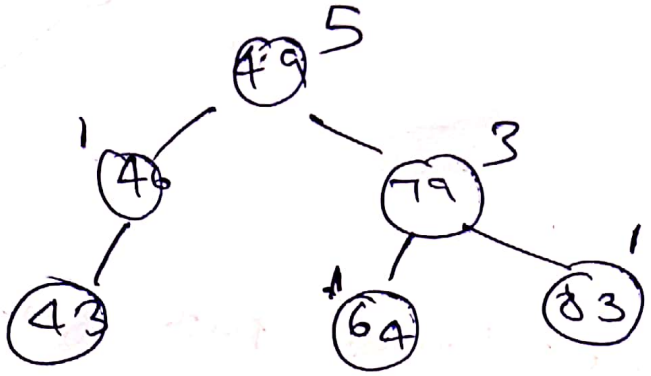
find min O(h) ~~left~~ ~~bottom~~ left leaf.

next largest than x O(h)

---

new req: rank(t) → how many plans are scheduled to land at times ≤ t.

Augment the BST struct.



insert numbers which correspond to subtree sizes.

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

let t = 79

look at 49.

49 < 79 add 1

add 2 (subtree 46)

se 79 add 1

add 1 (subtree 64)

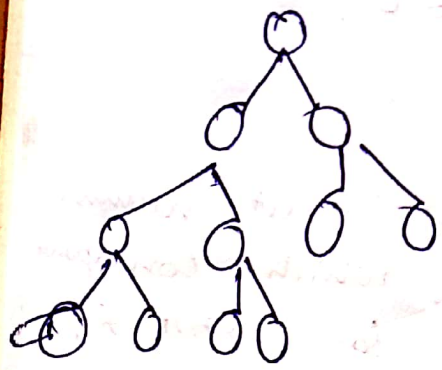
= 5

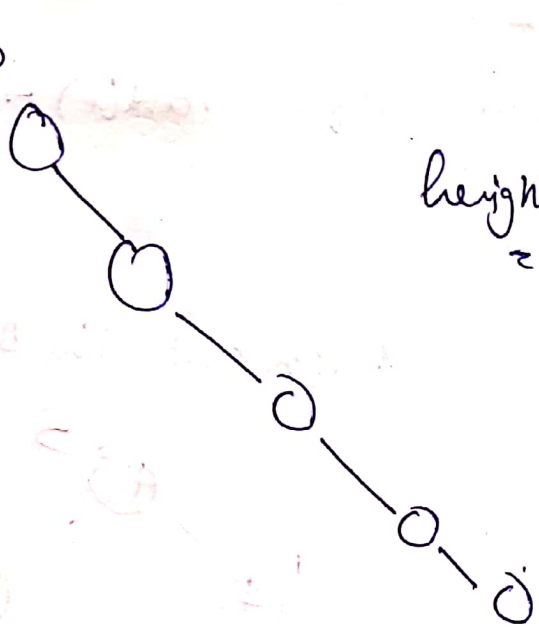
Tree needs to be balanced so that ht = O(log n)

Balanced or not

↓

height = log n





height = n.

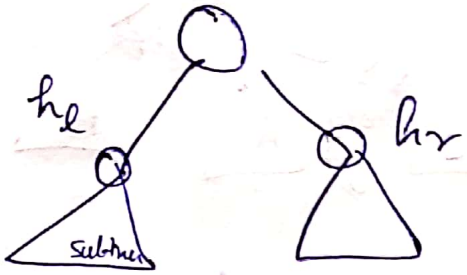
height of a node - longest path from node to the leaf.

= max(height of left + right right) + 1.

(leaf node always given - 1 height) to be summation for its children)

AVL trees

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



$$|h_l - h_r| \leq 1$$

AVL trees are balanced.

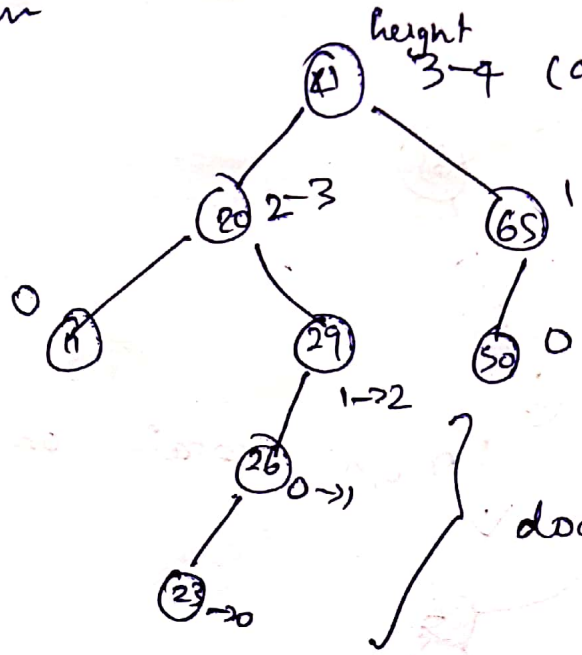
worst case $h \approx 1.44 \log n$

AVL insert

① Simple BST insert

② fix AVL property.

given

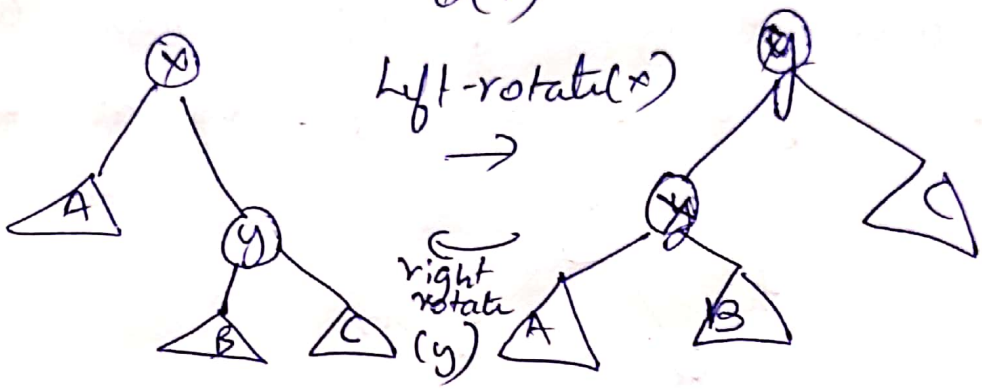


insert 23

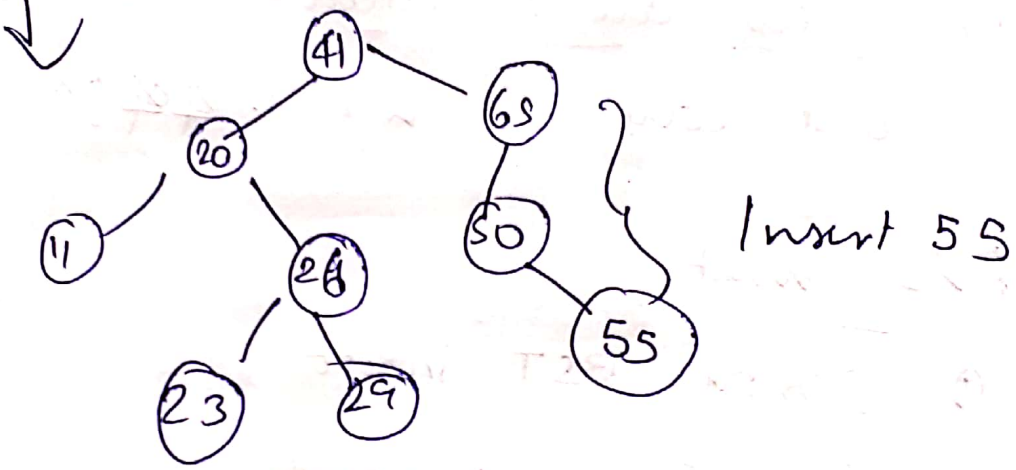
doesn't satisfy AVL property

Rotations:

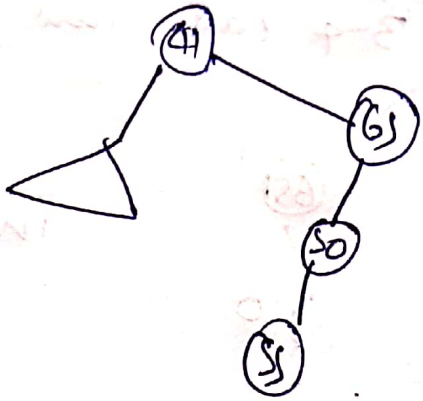
O(1)



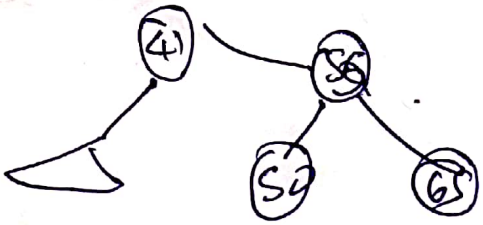
AxByC = AxByC



↓ rotate wrt 50



↓ right rotate on 65



(3) from the changed node, up

AVL Sort

- Insert n items $\Theta(n \log n)$
- in-order traversal $\Theta(n)$

Comparision Model

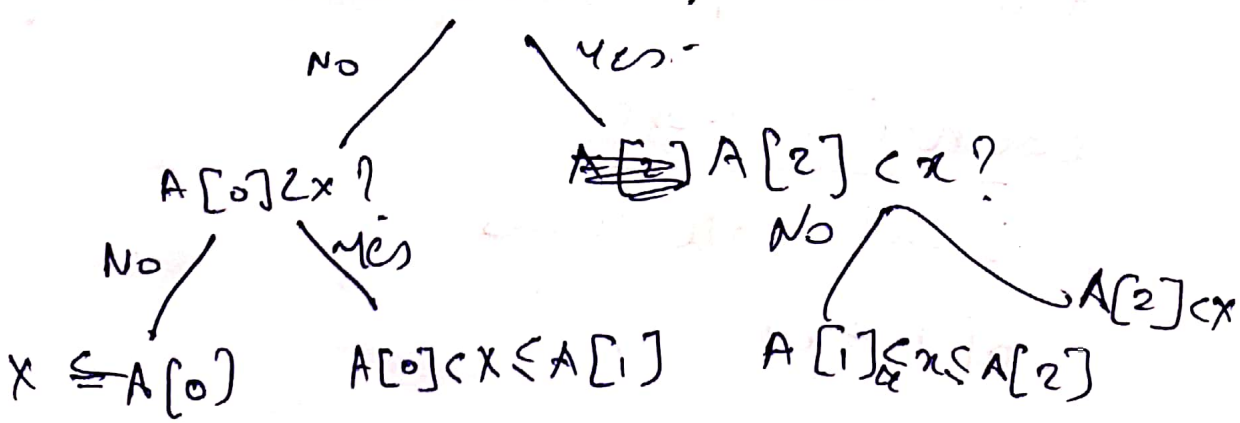
- all i/ps are black box (Abstract data type)
- only ops allowed are comparisions.

Decision tree

any comparison algo can be viewed as free of all possible outcomes comparisions & outcome & the resulting answer.

for any particular n

eg: binary search n=3



decision tree
internal node
leaf
root - to - leaf
length of path
height of the root

algorithm
binary decision
found answer
algo exe
running time
worst case running time

Searching lower bound:

n preprocessed items (sorted in a AVL tree etc)
finding a given item among them
in comparison
model require $\Omega(\lg n)$ in worst case
height is the worst case.

linear time sorting

- assume keys for sorting an int
- assume integer $\{0, 1, \ldots, k-1\}$
and each fit in machine word

counting sort

count all items
$O(n+k)$

Radix Sort

imagine breaking integers to bunch of columns

no of digit = $d \cdot \log_b k + 1$

- Sort integers by least significant digit
  :
  sort by most signi digit

Sort each using counting sort $O(n+b)$

total time $= O((n+b) \cdot d)$

$b = n$.

$= O((2n)d)$

$= O((2n) \log_n k)$

if $k \leq n^c$

$= O(nc)$

Hashing

set of items, each item has a key.

- insert
- delete
- search $\rightarrow O(1)$

simple approach

Direct access table (table lookup)

disadvantages:
- keys may not be non neg int
- large memory

Soln for

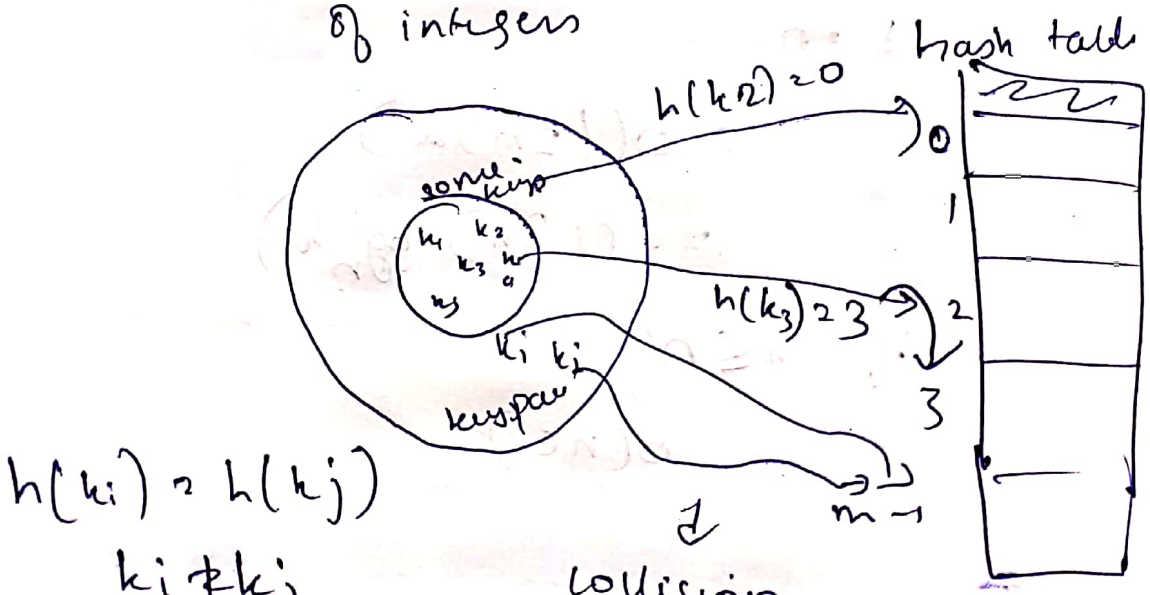
prehash
- maps keys to non-neg int

In theory, keys are finite & discrete (string of bits)

In python 'hash(x)' is the prehash of x

Soln for large space

hashing
- reduce all keys to small set of integers



$h(k_i) = h(k_j)$

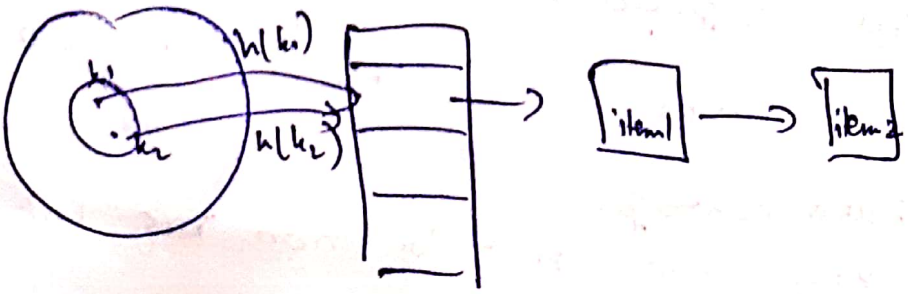
$k_i \neq k_j$

collision

collision is dealt with chaining & open addressing

Chaining: colliding items.

Store it as a linked list



In practice it works well because most keys will have some length, it randomised.

Assumption as simple uniform hashing (not True)

how to construct h?

① Division method.

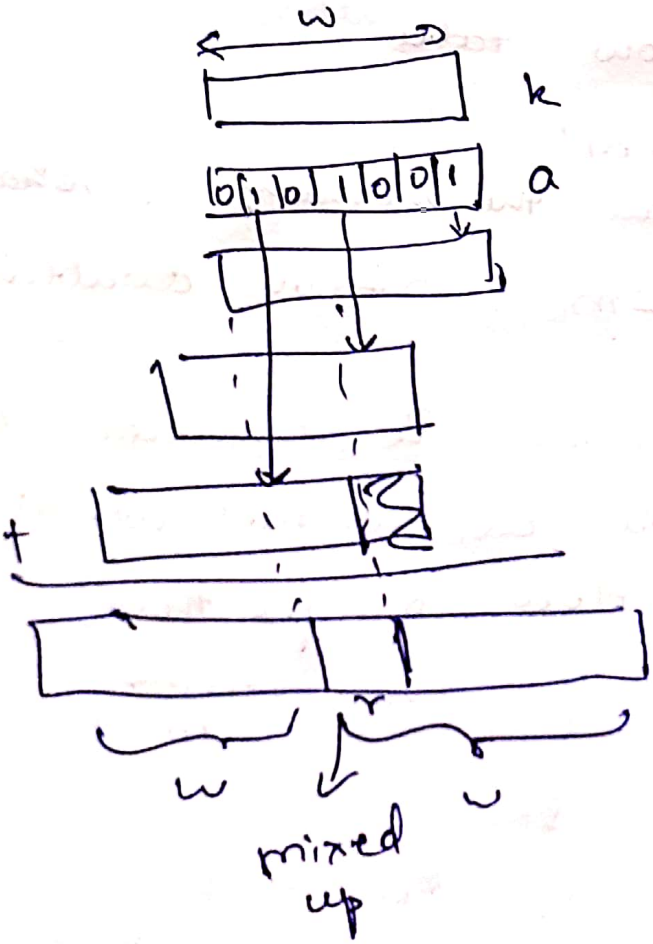
$h(k) = k \bmod m$

if m is prime & not close to pow(2), pow(10) it is good.

② Multiplication method

$h(k) = \lfloor (a \cdot k) \bmod 2^w \rfloor >> (w - r)$

↳ w bit machine



$m = 2^r$

③ Universal hashing

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

$a, b \rightarrow$ random

$p \rightarrow$ prime (very large)

for worst case keys $k_1 \neq k_2$

$$P(h_1) = h(k_2) = 1/m$$

what should m be?

we want m to be $O(n)$

idea:

start with small m,

$m = 8$

grow / shrink as necessary

if $n > m$; grow table

$m \rightarrow m'$

allocate the memory & rehash

$m' = 2m$ → table doubling

If we do n inserts & n delete, all values in table will be null with large size $m = n$. How to fix this?

1) If $m = \frac{n}{2}$, then shrink $m \rightarrow m/2$

this is slow

If $2^k \leftrightarrow 2^{k+1}$

then linear time table doubling

2) if $m = \frac{n}{4}$ then shrink $\rightarrow m/2$

Then this becomes constant time.

even python lists using table doubling

String matching
searching for substring

how to do this?

```
any (s2 == t[i : i + len(s)]
    for i in range (len(t) - len(s))
```

$O((|t| - |s|)|s|)$

how to make this faster?

Rolling hash ADT:

r.append(c) add char c to end of x
r.skip(c) delete first char of x.
r maintains a string x.

(as we search for string with same',
we delete our previous ~~s~~ searched
string's first letter & add ~~first~~ another
letter which comes after the searched
string)

Karp Rabin algorithm

```
for c in s: rs.append(c)
for c in t[: len(s)]:
                rt.append(c)
if rs() == rt(): ... (check if hash
                      for an equal)
for i in range (len(s), len(t))
    throw away first, add next
    ht.skip(t[i - len(s)])
    ht.append(t[i])
    if rs() == rt():
```

then potentially string of xs, rt matches, But it may not due to collision.

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

if equal:

found match

else:

happens with prop $\leq \frac{1}{|S|}$

Thm O(1) expected time

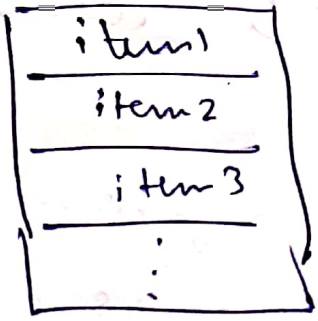
To make this ADT,

simple division method hash fn where m = random prime $\geq |s|$ works.

Open addressing - ~~dealing~~ ~~avoiding~~ with collision, no chaining

Probing:

hash fn specifies order of slots to probe for a key.



one item per slot

$m \geq n$.

$$h : \underbrace{U}_{\text{all keys}} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

final count

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

arbitrary key k to be a permutation of $0, 1, \ldots, m-1$

Inserting
Insert 586
$h(586, 1) = 1$

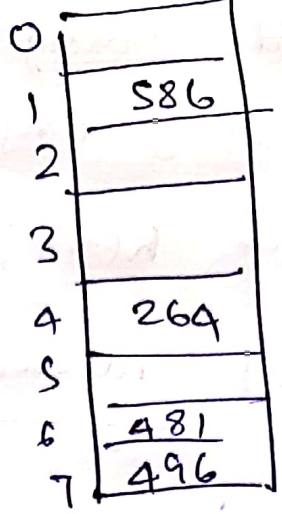
Insert $h(481, 1) = 6$
Insert $h(496, 1) = 4$

But 4 is occupied

$h(496, 2) = 1$ (second probe)

But 1 is occupied

$4(496, 3) = 7$ ✓ 3 trials.



Similar for search, check key to be filled elements and compare

double hashing

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

if $h_2(k)$ & m are relatively prime then permutation is possible

$m = 2^r$, $h_2(k)$ for all k is odd.

this assum UHT.

Uniform Hashing theorem

- Each key is equally likely to have any 1 of the $n!$ permutations as its probe sequence.

$\alpha = \frac{n}{m}$ Loss of op insert $\leq \frac{1}{1-\alpha}$

Password storage

one-way crypographic hash.

given $h(x)$, very hard to find x.

Irrational numbers

Catalan numbers

Set P of balanced parenthesis strings

1. $\lambda \in P$ (λ is empty)
2. If $\alpha, \beta \in P$, then $(\alpha)\beta \in P$

Every empty balanced parenthesis string via Rule 2 from unique α, β

High precision multi

2 n digit no

$0 \le x, y < r^n$ $\quad x = x_1 r^{n/2} + x_0$ $\quad x_1 \to$ high half, $x_0 \to$ low half

$0 \le x_0, x_1 < r^{n/2}$

$0 \le y_0, y_1 < r^{n/2}$

let $z_0 = x_0 \cdot y_0$, $\quad z_1 = x_0 y_1 + x_1 y_0$

$z_2 = x_2 y_2$

$= x_1 y_1 r^n + (x_0 y_1 + x_1 y_0) r^{n/2} + x_0 y_0$

4 multiplications of $n/2$ numbers

$\Rightarrow O(n^2)$ time

Karatsuba's Algorithm

$z_0 = x_0 y_0$

$z_2 = x_2 y_2$

$z_1 = (x_0 + x_1)(y_0 + y_1) - z_0 - z_2$

only 3 multiplications

$$O(n^{\log_2 3}) = O(n^{1.58})$$

To compute the millionth decimal digit of $\sqrt{2}$,

~~$\sqrt{2}$~~ ⇔ find last int digit of

$\sqrt{2^{2d}}$ $\quad d$

$\sqrt{2 \cdot 10^{2d}}$ $\quad d = 10^6$

Using newton's method,

$x_0 = 1$ (initial guess).

$$x_{i+1} = \frac{x_i + a/x_i}{2}$$

This has quadratic rate of convergence

To have d digits of precision,

you need $\log d$ iterations.

other methods exist for very large numbers.

High precision division

high precision rep of a/b.

Newton's method

$f(x) = \frac{1}{x} - \frac{b}{R}$ — 0 at $x = \frac{R}{b}$

$f'(x) = \frac{-1}{x^2}$

$$x_{i+1} = x_i + x_i^2\left(\frac{1}{x_i} - \frac{b}{R}\right)$$

$$= 2x_i - \frac{bx_i^2}{R}$$

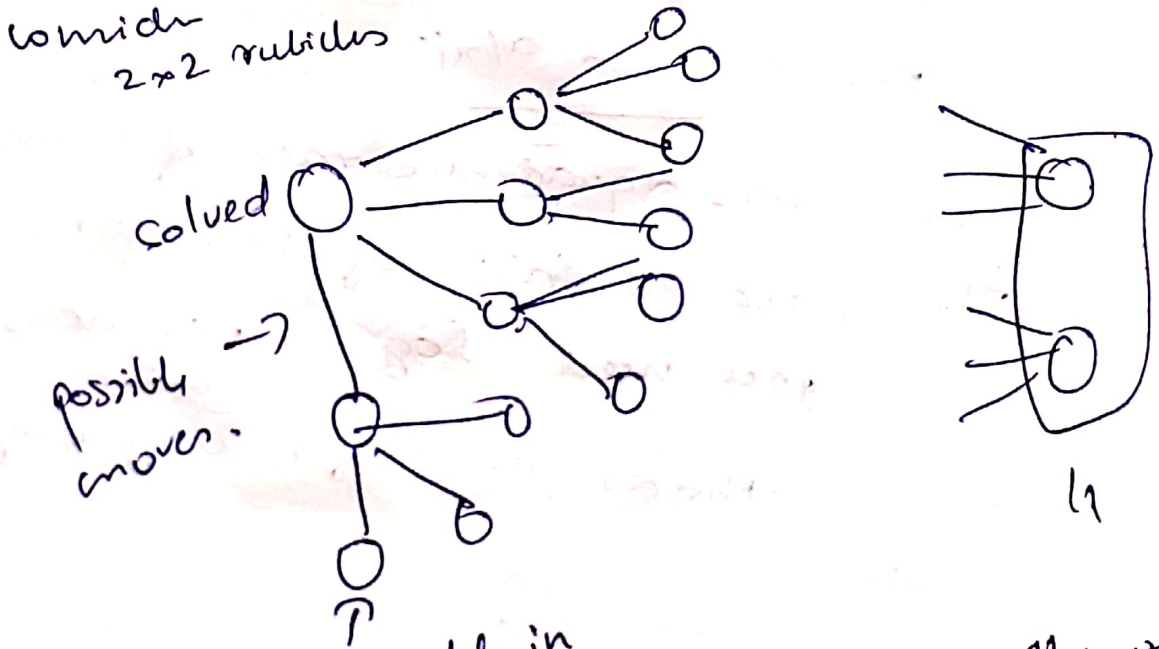
Breadth first Search

graph

$G = (V, E)$

$E = \{v, w\}$ unordered pair (undirected)

$E = (v, w)$ ordered pair (directed)



If ~~positi~~ moves it can be solved for worst

graph representation

adjacency list → $O(|E|+|V|)$

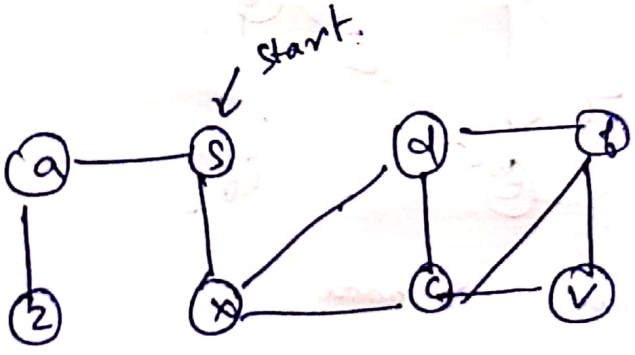
array Adj of $|V|$
each elem is a pointer to linked list.
for each vertex $u \in V$
Adj[u] stores its neighbours

Implicitly represented:
Adj(u) is a fn.

BFS
- visit all nodes reachable from given $s \in V$
- $O(V+E)$ time.

moves	node
0	s
1	Adj[s]
:	

- avoid revisiting vertices



move	node
0	s
1	a, x
2	z, d, c
3	b, v

time = $\sum |Adj[u]|$
$= 2|E|$
undirected
$= |E|$

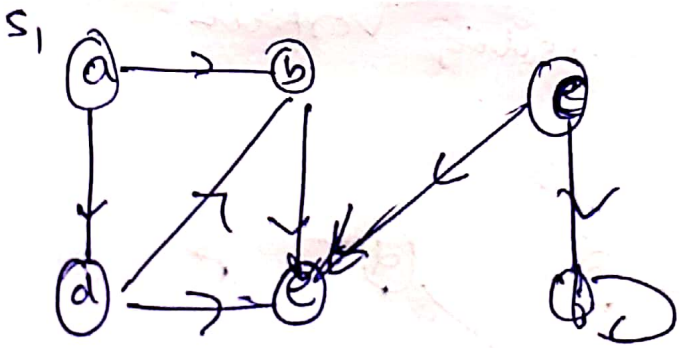
Depth first search

- recursively explore graph, backtracking which necessary

```
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, s)
```

- not repeat vertices

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



move	node	
0	a	
1	b	a->b
2	e	b->e
3	d	e->d
4	b	d->b; skip.

No other edges, backtrack

repeat this for every node as starting, skip if repeated.

0 c

1 b.

$O(V+E)$

- bold edges are tree edges

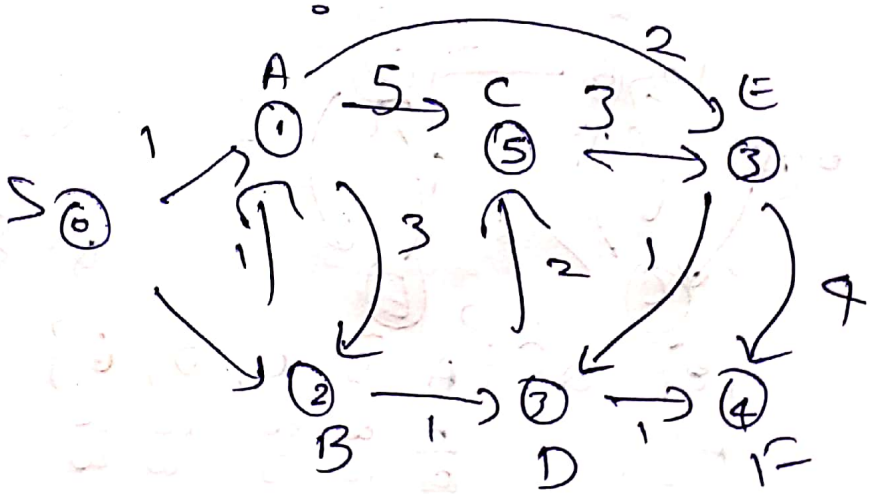
Shortest path

$G(V, E, W)$ $W \rightarrow$ weight

$E \rightarrow R$

p = path

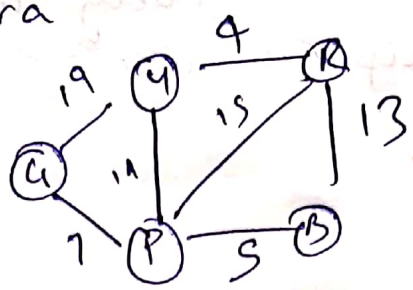
$$w(p) = \sum_{0}^{k-1} w(v_i, v_{i+1})$$



If there are -ve weights, and if it exists in a loop it can go into an infinite reduction of path weights of the nodes

Use Bellman ford in such cases, assigns $-\infty$ to the connected nodes. The nodes from there have fixed weights

Dijkstra



Q ← V[G]
S ← ϕ.

Initialize (G, s) → d[s] ← 0

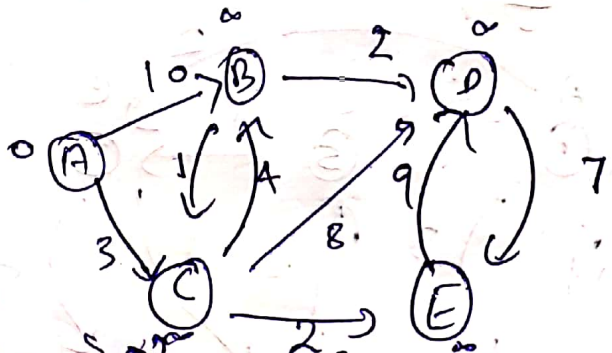
while Q ≠ ϕ

u ← extract min(Q) delete u from Q

S' = S ∪ {u}

for each vertex v ∈ Adj[u]

Relax(u, v, w)



S = { }

Q = {A, B, C, D, E}
0 ∞ ∞ ∞ ∞

S = {}

Q = {A, B, C, D, E}

S = {A}.

0	∞	∞	∞	∞
0	10	3	∞	∞
0	7	3	11	5

S = {A, C}

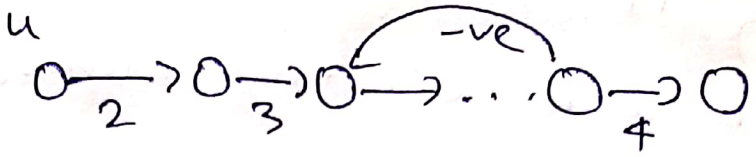
S = {A, C, E} . . .

↑

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

Bellman-Ford

for -ve weights, -ve weight cycles.



Bellman-Ford (G, w, s)

Initialise();

for i = 1 to |V|-1

for each edge (u,v) ∈ E

Relax every edge.

for each edge (u,v) ∈ E

~~relax~~

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

Then report -ve cycle exists

Dynamic programming

Fibonacci

$F_1 = F_2 = 1$

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

with fn overloading,

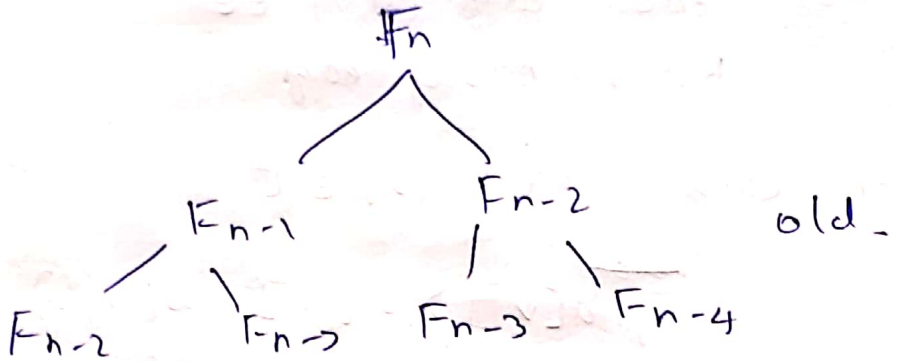
$T(n) = T(n+1) + T(n-2) + \Theta(1)$

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

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

Memoised DP algo:

```
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
```



old.

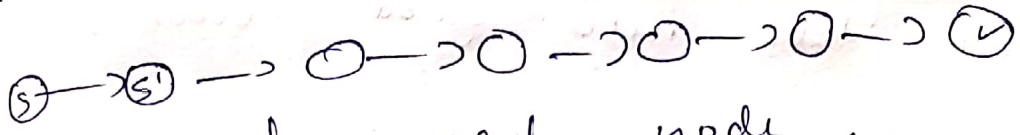
In DP, won't compute already computed things

↑ O(n)

Bottom up DP

```
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]
```

Shortest paths:

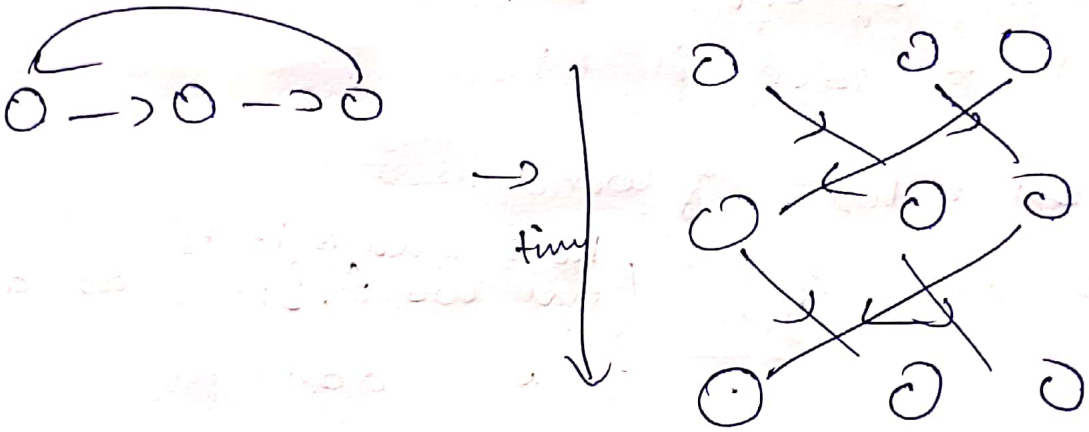


guess from each node,

But this is bad, so memoize computed path.

going to be inf on graphs with cycles.

main cyclic graphs acyclic

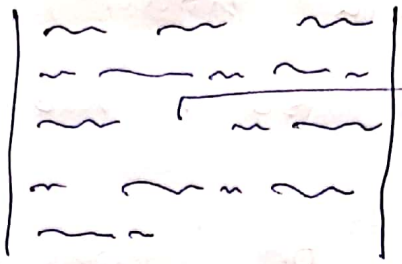


5 easy steps to DP

1. define subproblems
2. guess (part of solution)
3. relate subproblem solutions
4. recurse & memoize
   or build bottom up table.
5. solve original problem.

Text justification

split text into good lines,



How MS word used to do? greedy approach, fit as much as possible in 1 line, repeat. At least later uses DP to solve it.

text = list of word.

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

= { ∞ don't fit
(page width − total width)^3 otherwise

(later code)

how to solve.

guess: where second line begins.

try all possible words after the first word & see what if I started my second line here. At some point, there will be too much in first line, then abort.

Then need to guess where 3rd line begins and so on

Basically, have to set up sub problems such that after first

guess, if is the same problem again

① In this, subproblem = suffixes ⇒ (words[i:])

#sub probs : n

② guess: where to start next line.

#choices ≤ n-i = O(n)

③ recurrence: DP[i]
= min(DP(j) + badness(i,j)) for j in range(i+1, n+1)

④ topological order, i: n, n-1, ..., 0

total time = $O(n^2)$

⑤ Original prob DP(0)

Parent pointer: remember which guess was best
Store best j for each i.

Perfect-information Blackjack.

- Already know entire deck.

- deck = $c_0, c_1, \ldots, c_{n-1}$
- 1 player vs dealer
- 1$ bet per hand

② guess: hit or stand given a card
guess no of times to hit every hand

① Sub-problems: suffix c[i:] & (new hand)

#sub-problems: $2n$

guess choices $< n$

-1 -> lose dollar
0 -> tie
1 -> win dollar

③ recurrence:

$B_j(i) = \max($
outcome $\in \{-1, 0, 1\}$)
$+ B_j(j)$ for all j in range $(i+4, n)$
if it is valid play

$j \geq i + 4 +$ #hits + #double hits

Parenthesization

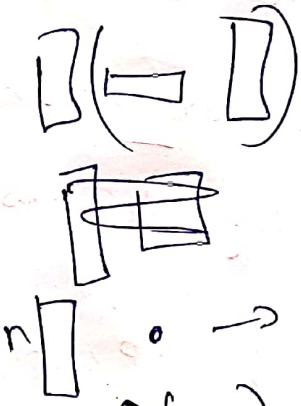
optimal eval of a associative expression.

matrix multipl: $A_0 \cdot A_1 \cdot \ldots A_{n-1}$

how to do it?

two ways

$O(n^2)$



$O(n)$

② guess outermost / last multi.

$(A_0 \ldots A_{k-1}) \cdot (A_k \ldots A_{n-1})$

last multi.

becomes subproblems

① subproblem:
optimal eval of $A_i, \ldots A_{j-1}$

$k \to O(j-i) \simeq O(n)$

③ recurrence:

$DP[i,j] = \min($

$DP(i,k) + DP(k,j) + A[i:k]A[k:j]$ (cost of)

for k in range $(i+1, j)$

$\hookrightarrow O(n^3)$

④ topological order:
increasing substring size

Edit distance:

given 2 strings x & y. what the cheapest way to convert x into y: (character edits)
↳ insert, delete, replace)

longest problem subsequence:

HIEROGLYPHOLOGY } longest
MICHAEL ANGELO } subsequence

① Subproblem:
edit distance on $x[i:]$ & $y[j:]$

#subprobs $= n^2$. $O(|x||y|)$

② ques: insert, delete & replace.

replace $x[i] \to y[j]$

insert $y[j]$

delete $x[i]$

③ recurrence

$DP[i,j] = \min($ cost of replace + $DP(i+1, j+1)$,
cost of insert + $DP(i, j+1)$,
cost of delete + $DP(i+1, j)$ $)$

④ topological order:

for $i \leftarrow |x| \dots 0$

$j \leftarrow |y| \dots 0$

$= O(n^2)$

Knapsack

- list of items
each has size & value,
(int)

knapsack of size S
maximize sum of value / a

② guessing : is item[i] included or not.

③ subproblem : items[i:]

DP[i] = max(DP[i+1],
DP[i+1] + v_i)

can't represent this.

subproblem = item[i:] + remaining capacity.

= O(n·S)

DP(i, s) = max(DP(i+1, x), ← remaining capacity
DP(i+1, x-s) + v_i)

if we don't choose i, x doesn't change.
if we choose, x, v change.

= O(n·S)

pseudo polynomial.
(if S is large, it is exponential)

2 kinds of guessing.

step 2 & 3: guessing (which subproblem to use)

in ①: add more ~~pre~~ subproblem to guess

Piano / guitar fingering

- given a musical piece, sequence of n notes, find fingering for each note.
  - fingers 1, ..., F
  - difficulties ~~d(p, f, q, g)~~

$d(p, f, q, g)$

note fingers

① Subproblem: suffixes notes[i:]

② guess: which finger to use for i.

③ recurrance: $\min_{f \text{ in fingers}} (DP[i+1] + d(i, f, i+1, ?))$

we don't know what finger to go to

① subproblem:
how to play notes[i:]
when use f for notes[i]

② guess: guess finger g for i+1

③ recurrence: $\min_{g \text{ in fingers}} (DP(i+1)(g) + d(i, f, i+1, g))$

④ topological order

for i in ~~range~~ reversed(range(n))

for f in F

⑤ original problem.

min (DP(0,b) for f in F)

$O(nF^2)$

Tetris

- given n sequence which fall
- each fall from top,
- full rows don't clear.
- can you survive n ?
- width is small.
- board is empty.

① subproblem: pieces[i:]

given board skyline

$n \cdot (h+1)^w$

② guess:

how to play i. (rotate)

4.w choices.

$O(n \cdot w \cdot (h+1)^w)$ problems

③ recurrence through max.

Same as previous.

Super Mario Bros:

- given entire level
- small $w \times h$ screen.
- configuration: everything on screen

velocity
score
time
screen vs level

$c^{w \cdot h} \cdot S \cdot T$ } large

total: $O(c^{w \cdot h} \cdot S \cdot T)$

total configurations

Computational Complexity

P - polynomial time for problems. (n^k)

EXP - solvable in exponential. (2^{n^c})

R - finite time.

Examples:

- negative weight cycle detection $\in P$
- $n \times n$ chess $\in$ Exp & $\notin P$
- Tetris (proper rules) $\in$ Exp
  don't know if P.
- Halting problem: given program, does it stop running? $\notin R$

Most decision problems are computable.

NP = { decision problems
solvable in poly time.
via a 'lucky' algorithm }

- non deterministic model
algo makes guesses. & says yes or no.
- guesses are guaranteed to give yes
answer if possible

another definition

NP = { decision problems with solutions
that can be checked in
poly time. }