Design & Analysis of Algos

~~Tractable~~ P: Solvable in Polynomial time.
$O(n^k)$.

NP: class of probs verifiable in poly time.

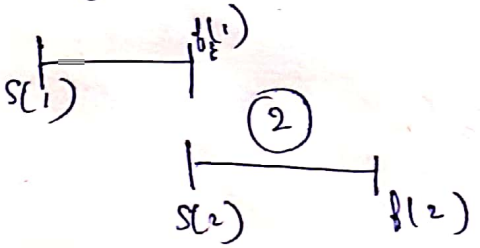
NP-Complete problems are the hardest in NP. (Eg. Hamiltonian cycle)

Interval scheduling

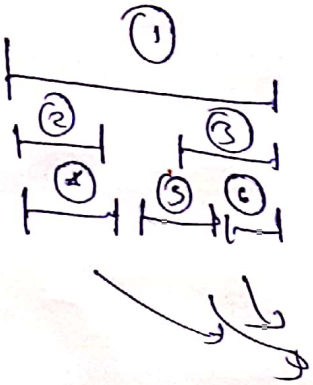
- we have resources & requests.
  i -> request
  s(i) -> start time f(i) -> finish time
  $s(i) < f(i)$

two request & compatible if they don't overlap ~~(s(i) < f(j)) or~~

① on this case. & $f(i) \leq s(j)$ or $f(j) \leq s(i)$



for



→ select compatible subset of requests of max size.

max size = 3

Claim: can solve using a greedy algo.

greedy algo: doesn't look ahead, maximum the first thing it does. Is myopic algorithm.

greedy Interval scheduling.

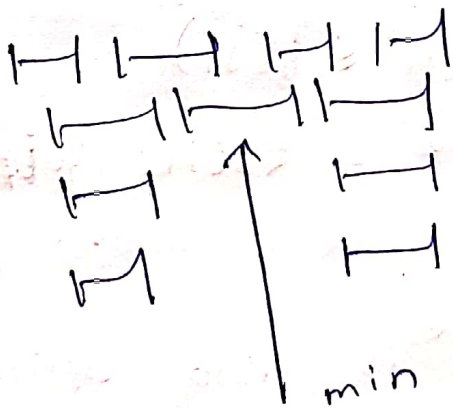
1. Use simple rule to select request.
2. Reject all requests not compatible with i.
3. Repeat 1 & 2 till all request are processed.

how to select? (the rule)

choose smallest?

in this case, Smallest will be skipped.

what if we choose one with least incompatible request?



if we select min, then we won't get optimum (the top row)

soln: pick one with earliest ~~sch~~ finish time.

Proving this:

Claim:
given a list of intervals L, greedy algo with earlist finish time produces k^* intervals, when k^* is max.

Induction on k^*.

Base case $k^* = 1$ Any interval works

Suppose claim holds for k^* and we are given a list of intervals whose optimal schedule has k^*+1 intervals

$$S^*[1, 2, \ldots, k^*+1] = \langle s(j_1), f(j_1)\rangle, \ldots \langle s(j_{k^*+1}), f(j_{k^*+1})\rangle$$

$$S[1, \ldots k] = \langle s(i_1), f(i_1)\rangle, \ldots \langle s(i_k), f(i_k)\rangle$$

$$f(i_1) \leq f(j_1)$$

$$S^{**} = \langle s(i_1), f(i_1)\rangle, \langle s(j_2), f(j_2)\rangle, \ldots \langle s(j_{k^*+1}), f(j_{k^*+1})\rangle$$

S^{**} is also optimal.

Define L' = set of intervals $s(i) \geq f(i)$ (only compatible ones)

Since S^{**} is optimal for L, $S^{**}[2, \ldots, k^* + 1]$ is optimal for L'.

$\therefore$ optimal schedule for L' has k^* size.

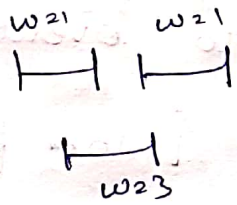
By inductive hypothesis, run greedy algo on L', should produce a schedule of size k^*.

By construction, greedy L' gives $S[2 \ldots k]$
$\downarrow$
$k-1$ size.

$\therefore \underline{k^* = k-1}$

Weighted interval scheduling

Schedule with max weight.



Solving this with DP:

one way to do it:

Subproblem:

$R \rightarrow$ total no of requests

$R^* = \{$ request $j \in R \mid s(j) \geq x \}$

$x \rightarrow$ finish time of all other request.

$n \rightarrow$ no of requests.

subprobs $= n$

Solve each subproblem once & memoize.

subproblems $\cdot$ time to solve 1 subproblem.

(assume order for lookups)

DP guessing:

try each request i as possible first request.

$$= \max_{1 \le i \le n} \left(w_i + opt(R[b[i]]) \right)$$

$O(n^2)$

Divide & conquer.

- given a problem size n,
- divide it into 'a' subproblems of size n/b, $b > 1$, $a \ge 1$.
- Solve each subproblem recursively.
- Combine solutions into overall soln.

$$T(n) = aT(n/b) + [\text{work for merge}]$$

Convex hull

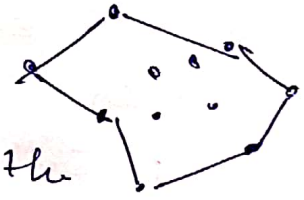
- given n points in a plane

$$S = \{(x_i, y_i) \mid i = 1, 2, \ldots n\}$$

assume no 2 have same x coord, y coord, no 3 in a line

Convex hull: smallest-polygon containing all points in S.

Specify it as sequence of points on bounding of the hull in clockwise.



Brute force.

Draw line b/w 2 points. See if all other points are in one side of the line. If yes then its a proper segment of the convex hull.

$O(n^2)$ segments

$O(n)$ test complexity.

total $O(n^3)$

Using Divide & conquer:

- Sort points based on x coords.

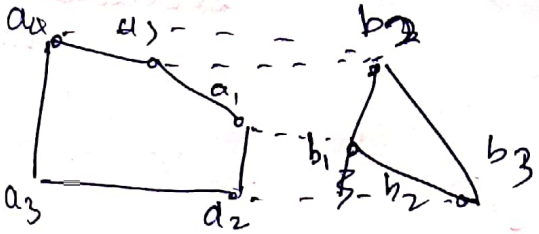
For input S,

Divide left half A & right half B by x coords.

Compute CH(A) & CH(B) recursively

Combine.

- How to merge?



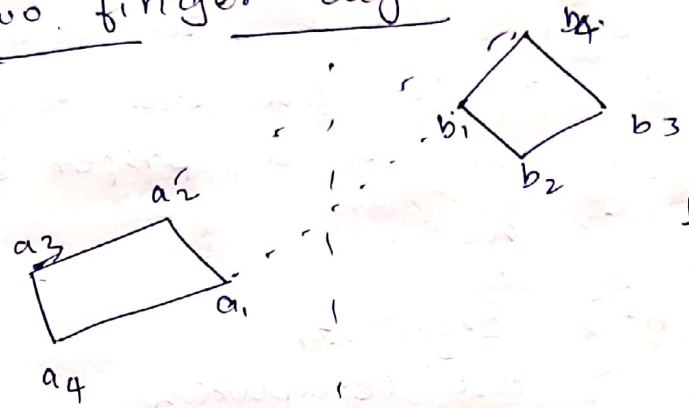
Connect the points of the right & left and see if forms a segment.

$a_4 b_2$ forms upper tangent

$a_2 b_3$ forms lower tangent

obvious merge algo $O(n^2)$ looking all points in A & B

Two finger algorithm.



for each pair of point, find the intercept.

- starting with $a_1 b_1$. we fix a_1, move CW to b_2, find $a_1 b_2$ intercept but it is lower. So we go to b_4 and find intercept. which is higher. So we fix b_4. but we need to change a_1. So we check $a_2 b_4$ & $a_4 b_4$. We fix a_2. Then we see around b_4. Then we go a_3. See around b_4. fix b_4. Then see ~~a routine~~ a_4 and fix a_3.

$\therefore$ $a_3 b_4$ is upper tangent. (It has highest intercept)

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

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

(only for merge) excluded sort time

Median finding

- Better than $O(n \log n)$.

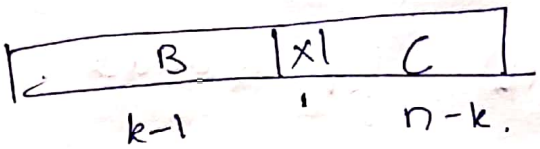
given n numbers, define Rank $R(x)$, no of elem in $S_1 \leq x$

Select (S, i)

- Pick $x \in S$
- Compute $k = \text{rank}(x)$

$B = \{y \in S \mid y < x\}$

$C = \{y \in S \mid y > x\}$



if $k = i$,
return x.

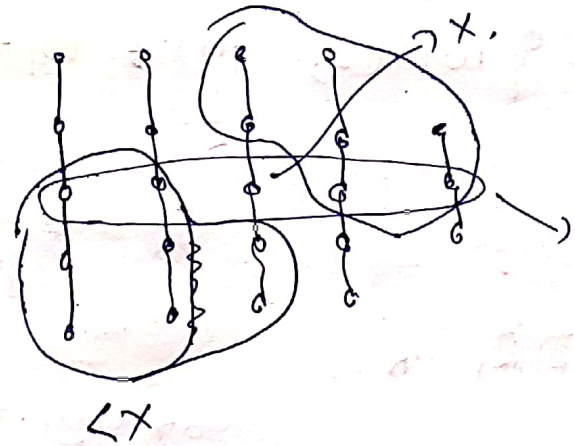
else if $k > i$,
return select(B, i).

else
return select $(C, i-k)$ (-k to keep index right wrt full list)

$O(n^2)$ for bad selection.

pick x cleverly:

arrange S into columns of size 5,
Sort each column, - linear time
(largest or top)
Find median of medians as x,



How many elems guaranteed $> x$?

Half of $\frac{n}{5}$ groups contribute atleast 3 elem $> x$, except 1 group with < 5. and 1 group that contains x,
at least $3\left(\frac{n}{10} - 2\right)$ elems are $> x$.

Similarly,

$3\left(\frac{n}{10} - 2\right) < x.$

Recurrence:

$T(n) = \begin{cases} O(1) & \text{for } n \le 140 \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n) & \end{cases}$ (some n)

$T\left(\left\lceil \frac{n}{5} \right\rceil\right)$ → (median of $\frac{n}{5}$ column)

$T\left(\frac{7n}{10} + 6\right)$ → discarding $\frac{3n}{10} - 6$ elements, so remain elem left

$O(n)$ → sorting of all cols

$O(n)$.

Weighted Interval scheduling.

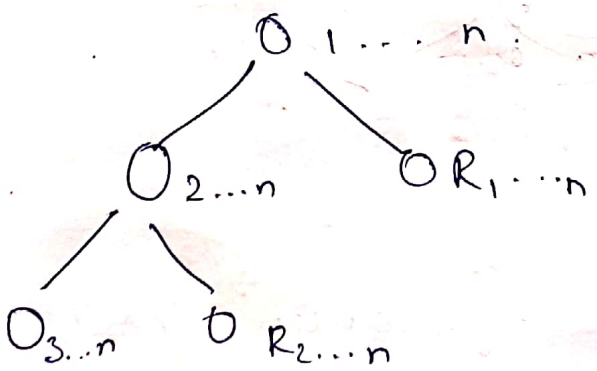
idea is we should try out every possible start time interval

$WIS(1, 2, \ldots n)$.

select 1 or not?

$\max\{WIS(2, \ldots, n), w_1 + WIS(R')\}$

R' → every request that starts after 1 finish

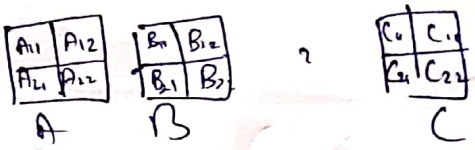


sorting → $n \log n$

recursion → $O(n)$

this was n^2 before

strassen algo (matrix mul)



A B C. $O(n^3)$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad \rightarrow n/2.$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22} \quad = 8\cdot\left(\frac{n^3}{2}\right)$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad O(n^3)$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

$$T(n) = 8T\left(\frac{n}{2}\right) + O\left(\frac{n^2}{2}\right)$$

Strassen algo creates 7 matrices M (addition)

such that

each M requires only 1 multi.

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

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

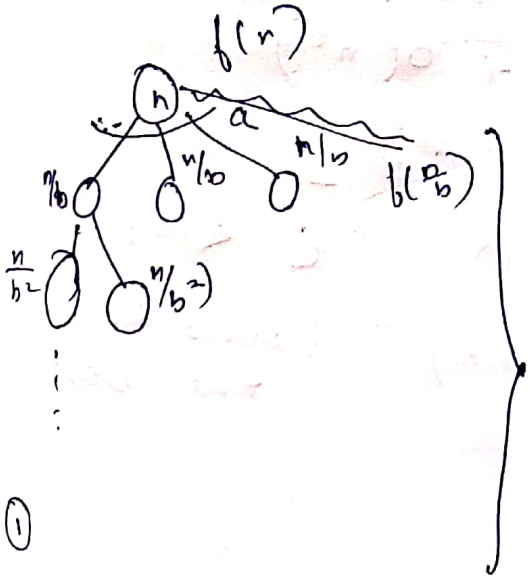
Master theorem.

given recursion $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ recursion merging.

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^c) \\ \Theta(n^c \log_n^{k+1}) & f(n) = \Theta(n^c \log_n^k), \ c = \log_b a \\ \Theta(f(n)) & f(n) = \Omega(n^c) \end{cases}$$
$c < \log_b a$

$c = \log_b a$

$c > \log_b a$



$$T(n) = f(n) + af\left(\frac{a}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots + a^t f(1)$$
$$= f(n) + af\left(\frac{a}{b}\right) + \ldots a^t f(1)$$

$$\sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

first case.

$$\sum a^i f\left(\frac{n}{b^i}\right) = \sum a^i \left(\frac{n}{b^i}\right)^c$$

$$= O\left(n^c \sum \left(\frac{a}{b^c}\right)^i\right) \quad c < \log_b a$$

↓ increasing seq.

$$= O\left(n^c \left(\frac{a}{b^c}\right)^t - 1\right) \qquad t = \log_b n.$$

$$= at$$

matrix mult

for ~~strassen~~ algo,

$$T(n) = 8T\left(\frac{n}{2}\right) + O\left(\left(\frac{n}{2}\right)^2\right)$$

$$a = 8, \quad b = 2, \quad c = 2 < \log_b a = 3$$

$$= n^3$$

for Strassen algo

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

$$a = 7, \quad b = 2, \quad c = 2$$

$$= O(n^{\log_2 7}) \qquad c < \log_2 7.$$

$$= O(n^{2.8\ldots}) \qquad \text{(used to be the best)}$$

FFT

$$A(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$$

$$= \sum_{0}^{n-1} a_k x^k$$

operations on polynomials:

① evaluation: $A(x)$ & $x_0 \rightarrow A(x_0)$

Horner's rule

$$A(x) = a_0 + x(a_1 + x(a_2 + \dots))$$

$O(n)$

② add: $A(x), B(x)$. $C(x) = A(x) + B(x)$ $\forall x$.

$$c_k = a_k + b_k$$

$O(n)$.

③ multiplication:

$A(x), B(x)$. $C(x) = A(x) \cdot B(x)$

$$c_k = \sum_{0}^{k} a_j b_{k-j} \qquad O(n^2)$$

convolution of vectors & reverse (B)

(inner product of all possible shifts)

polynomial represented by roots:

$r_0, r_1, \dots, r_{n-1}$:

$$C(x - r_0)(x - r_1) \dots (x - r_{n-1})$$

polynomial represented by samples

(x_k, y_k) for $k = 0, \dots, n-1$

$A(x_k) = y_k$ x_k's distinct.

Algos	coeff	roots	samples
eval	$O(n)$	$O(n)$	$\boxed{O(n^2)}$
add	$O(n)$	$\boxed{\infty}$	$O(n)$
mul	$\boxed{O(n^2)}$	$O(n)$	$O(n)$

$n \log n$

Divide & conquer algo:

goal compute $A(x)$ $\forall x_i \in X$.

divide into even & odd.

$$A_{even}(x) = \sum_{0}^{n/2} a_{2k} x^k = \langle a_0, a_2, \dots \rangle$$

$$A_{odd}(x) = \sum_{0}^{n/2} a_{2k+1} x^k = \langle a_1, a_3, \dots \rangle$$

combine $A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$

In conquer: recursively compute $A_{even}(y)$ & $A_{odd}(y)$ for $y \in X^2 = \{x^2 / x \in X\}$

$$T(n, |X|) = 2 \cdot T(n/2, |X|) + O(n * |X|)$$

$$= O(n^2)$$

how to reduce ?

Collapsing set x if.

$|x|^2 = |x|/2$ & x^2 is collapsing.

or $|x| = 1$

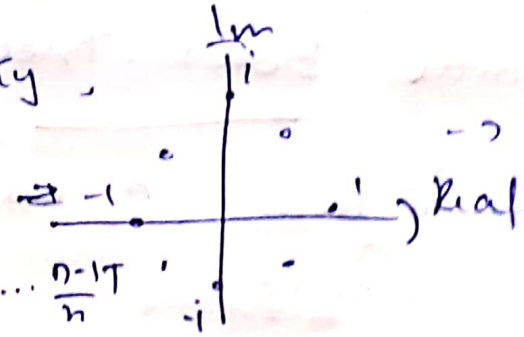
$|x|=1$ $x = \{1\}$

2 $x = \{-1, 1\}$

4 $x = \{i, -i, -1, 1\}$

8 $x = \{\frac{\sqrt{2}}{2}(1+i), \frac{\sqrt{2}}{2}(i-1), i, -i, -1, 1\}$

n^{th} root of unity.



→ unit circle

for $\theta = 0, \frac{\pi}{n}, \frac{2\pi}{n}, \ldots \frac{n-1}{n}\pi$

$\pi = 2\pi$

$= \cos\theta + i\sin\theta$

$= e^{i\theta}$

~~$e^{i\theta}$~~ $x_k = e^{ik\pi/n}$.

$= o(n \log n)$

FFT → divide & conquer algo for discrete fourier transform

: V.A for $x_k = e^{ik\pi/n}$

$V_{jk} = x_j^k = e^{ijk\pi/n}$.

Fast polyn mult.

$A^* = FFT(A)$

$B^* = FFT(B)$

$C_k^* = A_k^* B_k^*$ —

$C = IFFT(C^*)$

<u>Claim</u>: $V^{-1} = \bar{V}/n$.

$x_k^{-1} = e^{-i\xi k/n}$. ⟵ $n \log n$

Van Emde Boas Trees

goal: maintain n elements among $\{0, 1, \ldots, u-1\}$

subject to insert, delete, successor.

done in $O(\log\log u)$

if $u = n^{O(1)}$ or $n^{\log(O(1))} n_k$

then $\lg\lg u = O(\lg\lg n)$

used in n/w routers

where might $\lg\lg u$ come from?

binary search on levels of tree.

① Consider storing in an array of size u.

0 = absent 1 = present.

| 0 1 | 0 0 0 | 0 | 0 1 | 1 | 0 0 1 |

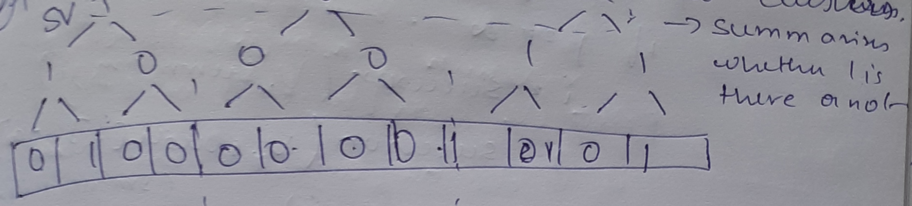
$u = 12$

$n = 4$.

Insert / delete = $O(1)$

Successor - $O(u)$

② Split our universe (array) into clusters.



③ Consider doing OR of two element & make binary tree.

SV → Summary vector.

should be split into $\sqrt{u}$ clusters.

Insert = $O(1)$

Successor (x) → look in x's cluster
→ look for next 1 bit in SV.
→ look for first 1 in that cluster

$O(\sqrt{u})$

if $x = i\sqrt{u} + j$ $0 \le j < \sqrt{u}$

i → cluster no, j → position of x in that cluster.

③ recurse : V

- V.cluster $[i]$ = size $\sqrt{u}$
  $0 \le i < \sqrt{u}$

  V.summary = size $\sqrt{u}$

Insert (V, x)

Insert (V.cluster[high(x)] , low(x))

Insert (V.summary , high(x))

$O(\log u)$

Scanned with CamScanner

Successor (V, x):

i = high(x)

j = successor(v.cluster[i], low(x))

if j == ∞:

i = successor(V.summary, i)

j = successor(V.cluster[i], -∞)

(smallest item)

return index(i, j)

$T'(\log u) = 2T\left(\frac{\lg u}{2}\right) + O(1)$

$= O(\lg u)^{\log_2 3}$

④ store min:

Insert(V, x):

if x < V.min

V.min = x.

~~Successor(V~~

Successor(V, x):

j = V.cluster[i].min

O(lg u)

store max also.

Insert(V, x)

min, max

then

Successor(V, x): if(x < v.min) return v.min

i = high(x)

if low(x) < V.cluster[i].max:

j = successor(V.cluster[i].low(x))

else

i = successor (V, summary.high(x))

j = V.cluster[i].min

return index(i, j))

$O(\log \log u)$ for successor.

how to make insert faster?

① don't store min recursively.

if min is blank, store it there.

i.e

if V.min = None: V.min = x, V.max = x

return.

if x < V.min

swap x ↔ V.min.

if V.max < x: V.max = x.

if V.cluster[high(x)].min = None:

A Insert (V.summary, high(x))

B Insert (V.cluster[high(x)], low(x))

if we do A, B takes constant time

otherwise B is one recursive call.

$\therefore O(\lg \lg u)$.

How to delete?

```
Delete (V, x):
    if x = V.min:
        i = V.summary.min
        if i = None: V.min = V.max = None  return
        x = V.min = index(i, V.cluster[i].min)
    Delete(V.cluster[high(x)], low(x))
    if V.cluster[high(x)).min = None:
        Delete(V.summary, high(x))
    if x > V.max:
        if V.summary.max = None
            V.max = V.min
        else i = V.summary.max
            V.max = index(i, V.cluster[i].max)
```

Lower bound:

$\Omega(\lg\lg u)$ for $u = n^{\lg(1)n}$, space $O(n \operatorname{poly}\lg n)$

How to reduce space from $O(u)$?

- store only non empty clusters, V.cluster changed from array to dict

$O(n \lg\lg u)$ space

↓ can be done.

$O(n)$

Amortization.

just look at total computation to solve a problem.

Then amortize cost per operation

= (total / #ops)

if total = $O(n)$

then per op = $O(1)$, with n ops.

Aggregate method:

$$\frac{\text{total cost of } k \text{ ops}}{k}$$

Amortized bounds:

- assign a cost for each op, s.t. Am.

such that sum of costs is preserved.

$$\sum_{op} \text{amortized cost} \geq \sum_{op} \text{actual cost}$$

accounting method

- allow an operation to store credit in a bank account (non neg balance)
- allow op to take coins out, pay for time using credit stored in bank.

Claim:

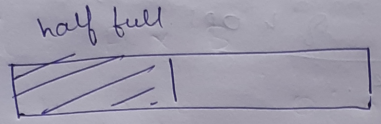
$O(\lg n)$ per insert, 0 per delete, amortized

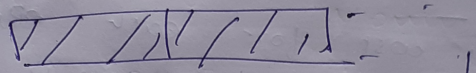
table doubling.

- when insert, add coin on it
  worth $c = O(1)$
  Eg: used in hashing.

Charging method

- allow ops to charge cost to the past.
- amortised cost = actual cost − total charge to past + total charge in future.





charge it to prev o/ps.

only charge once. (they have to pay in future)

Table doubling & halfing.

100% full → double it
25% full → half it.

charge halving to $\geq \frac{m}{4}$ deletions since last resize.

doubling to $\geq m/2$ insertions from last resize.

Potential Method

- difine potential function ϕ mapping data structure config → non neg int.

amortised cost = actual cost + $\Delta\phi$ (after − before)

Σ amortised = Σ actual + ϕ(end) − ϕ(begin)

⇒ pay ϕ(begin) at beginning.

- should be constant or zero.

what are they useful for?

Binary Counter: 0011010111

+ 1

0011011000

using potential method:

Many bits change but only constant no change in amortised sense.

- increment has a clear cost $O(1 + \#\text{trailing 1s})$

but ϕ = no of 1 bits

increment destroys t 1 bits & creates 1 1-bit.

So, the Amortised cost = $O(1+t-t+1)$

$= 2$

Big O

$= O(1+t) - ct + k$

$= 2$

Randomised algorithm.

- something that generates a random number, make decisions based on this value.

Then algo can be probably correct & probably fast (& both)

Matrix product (P C)

$C = A \times B$

Simple algo : $O(n^3)$.

get $O(n^2)$ algo (probably correct Monte Carlo)

if $A \times B = C$, then prob[output = YES] = 1.

if $A \times B \neq C$, then prob[output = YES] $\leq 1/2$

↓ bounded false pos,

this prob can be brought down by randomizing every time

entries $\in \{0, 1\}$

Frievald's algorithm:

Choose random binary vector $r[1 \ldots n]$ such that $P_r[r_i = 1] = \frac{1}{2}$ independently for $i = 1, \ldots, n$.

If $A(Br) = Cr \rightarrow$ YES

$r : 1 \times n$

$C : n \times n$

else

$\downarrow$

$O(n^2)$ NO

if $AB = C$, $A(Br) = (AB)r$

Analysing correctness, $AB \neq C$.

$Prob[ABr] = Cr] \geq 1/2$.

Let $D = AB - C$

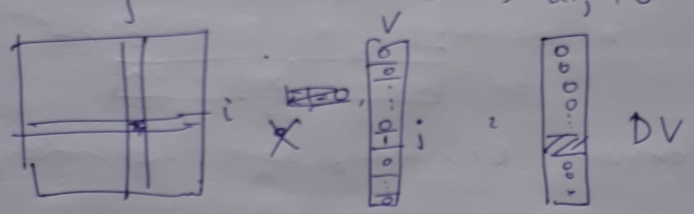
$D \neq 0$

(one entry atleast)

Show that there are many r such that $Dr \neq 0$.

$D \neq 0$ but $Dr = 0 \rightarrow$ bad r.

$\Rightarrow d_{ij} \neq 0$



$DV \neq 0$

$(DV)_j = d_{ij} \neq 0$

Take any r that can be chosen such that $Dr \neq 0$

$r' = r + V$. (mod 2 arithmetic)

$Dr' = D(r+V) = 0 + Dv \neq 0$

r to r' is 1 to 1. if $r' = r+V$

$= r'' + V$

then $r = r''$

just change the value where it is supposed to be 0.

atleast Half of them can be good rs.

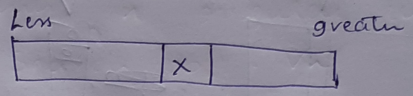
Quick sort

- practical performance.
- divide & conquer randomized.
- in-place, all work in divide step
- 3 diff variants:
  - Las vegas Quicksort (probably fast algo)

n - element array A.

Divide 1. Pick a pivot element x in A.

Partition the array into sub arrays.



Conquer Recursively sort subarrays.

Basic quick sort:

pick pivot to be A[1] (value) or A[n]

Create ↑ this is O(n) matrix based on this.

worst case is $O(n^2)$

Selecting pivot such that ~~O(n)~~ O is O(nlgn).

- & median finding algorithm.

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

due to median and pivoting. — median selection — divide.

(basic quick sort better in practice)

x is chosen at random from array.
Expected time is $O(n \log n)$

Paranoid quick sort

- Be afraid of unbalanced partitions.

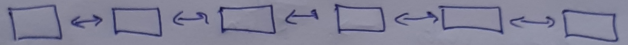
Repeat
- choose pivot to be randn elem of A.

Until resulting partition is such that

$|L| \leq \frac{3}{4}|A|$ & $|G| \leq \frac{3}{4}|A|$

Skip List

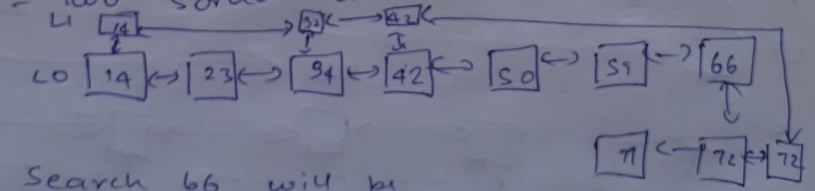
- Randomised data list
- Maintains dynamic set of n elements.
- Do it in $O(n \lg n)$ time.
- Begin with a linked list



Search $O(n)$

Sorted LL, search $O(n)$

- two sorted linked list



Search 66 will be

14 → 42 on L1

42 → 42 L1 → L0

42 → 66 L0

Search(x): (2 LL)

- walk right on top LL L_1 until going right would go too far.
- walk down to bottom list L_0 and walk right until element is found or not.

Analysis:

cost: $|L_1| + \frac{|L_0|}{|L_1|}$.

minimise this when terms are equal

$|L_1|^2 = |L_0| = n$.

$|L_1| = \sqrt{n}$

$\sqrt{n}$ elements in regular intervals

So Search cost: $\sqrt{n}$

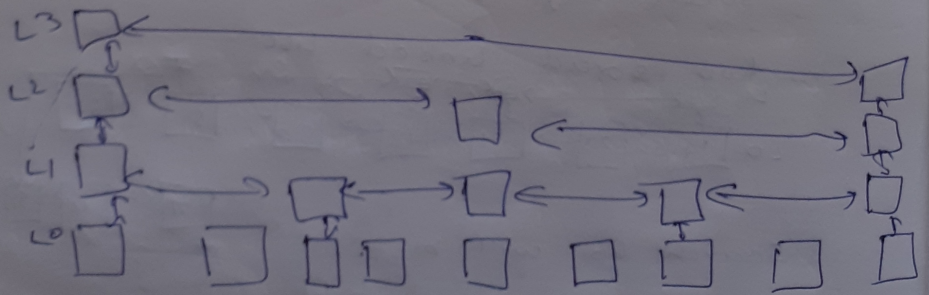
2 sorted lists → $2\sqrt{n}$

3 sorted lists → $3\sqrt[3]{n}$

⋮

k sorted lists → $k\sqrt[k]{n}$

lgn sorted lists → $\lg n \sqrt[\lg n]{n} = 2 \lg n$



Insert(x)

- To insert an elem x into a skip list
- search(x) to see where x fits into the bottom list.

Insert into some of the lists (which ones?)

Flip fair coin.

if head, promote x to next lvl & repeat (can be newly created)

else stop.

↑ O(n)

Lemma

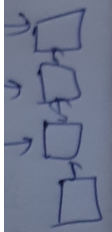
lvls in n-element skip list $O(\lg n)$ whp. (with high prob)

Hashing: Dictionary problem.

- maintain set of items (with key)
- insert, delete, search.
  item item key

Hashing: O(1) per operation

- u = # space of universe → all possible keys
- n = # keys in DS currently
- m = # slots in table
- $h: \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, m-1\}$



Hashing with chaining $O(1+\alpha)$ α load factor
↓
n/m

- requires assuming keys are random.

Universal hashing

- choose h randomly from H.
- require H to be universal hash family.

$$\Pr_{h \in H}\{h(k) = h(k')\} \leq 1/m \quad \text{all } k \neq k'.$$

Theorem: for n arbitrary distinct keys for random $h \in H$ ↗ universal

$E[\#\text{keys colliding in slot}] \leq 1 + \alpha$.

Dot product hash family:

- assume m is prime.
- assume $u = m^r$ r is int.
- view key k in box m

$k = \langle k_0, k_1, \ldots, k_{r-1} \rangle$

- for key $a = \langle a_0, a_1, \ldots, a_{r-1} \rangle$

define $h_a(k) = (a \cdot k) \bmod m$.

$$= \sum_{0}^{r-1} a_i \cdot k_i \bmod m$$

$H = \{h_a \mid a \in \{0, 1, \ldots, u-1\}\}$

Word RAM model: manipulating (individual word values)
- O(1) words take O(1) time
& keys fit in the word

Another method:

$$h_{ab}(k) = [(a \cdot k + b) \bmod p \bmod m]$$

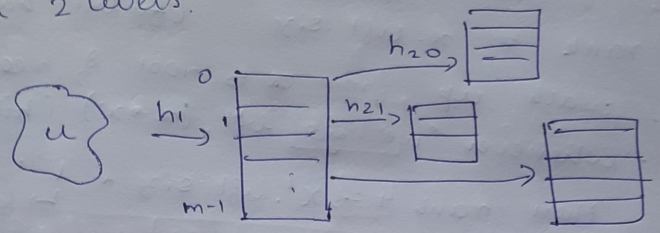
↘ ↙

uniformly random values.

$p > m$, p → prime

Static dictionary problem (perfect hashing)

- only search. O(1) time, O(n) space.
- use 2 levels.



$m = O(n)$ ↓ zero collisions in second level.

- polynomial build time whp.

Augmentation: Range trees.

Easy tree augmentation.

- we have a tree, for every node x, some function of subtree rooted at x is stored.
- suppose $x.f$ can be computed in O(1) from x children & from f value stored in the children.
- if set of nodes change, cost is:
  O(# ancestors of s) to update f fields.
- O(lg n) update in AVL trees, 2-3 trees.

Order statistics trees.

ADT:
- insert
- delete
- successor
- rank: index of k in overall sorted order
- select: find key of rank i

- use easy tree aug with f(subtree) = no of nodes
  ⇒ x.f = 1 + sum(c.f for every child c)

- rank = x.left.size (left has <x)
- walk up from x to root of tree
- whenever we go left, (x → x') (right child of x' is x)
  then rank += x'.left.size + 1
  ↳ for x' itself

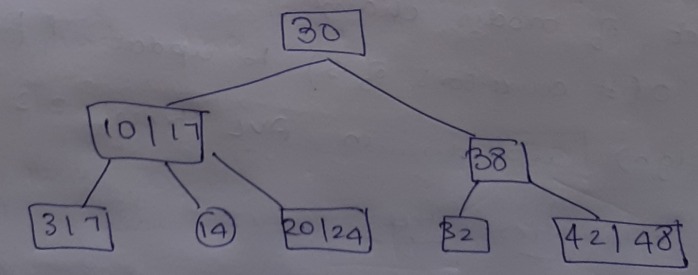
↳ logn times (logn steps, O(1) per step)

Select(i):
- x = root
- rank = x.left.size + 1
- if i = rank, return x
  if i < rank, x = x.left
  if i > rank, x = x.right
  i -= rank.
- repeat

(if we insert, say val<min, then rank of everything. Shift by 1, it'll take linear time to do it)

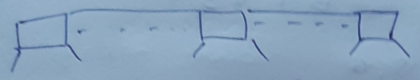
Level Linking 2-3 trees

2-3 trees



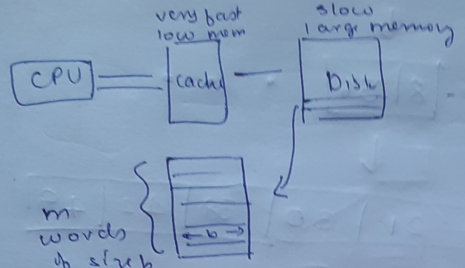
- every node has 2 or 3 children.
  Every node can have 1 or 2 keys

B- trees



depth is $\log_n$.

n+1 children



we want to utilise the b block efficiently. So generally make no of nodes something like word size in cache.

Branching factor

- In 2-3 → 2 (B

$2B \geq$ no of children $\geq$ Branching factor B

$2B-1 >$ # keys $\geq B-1$

- all the leaves at same ~~level~~ depth.

Searching:

Bring in key, look at all keys in the node we are looking it, if in between two, then go down & continue

$O(\log n)$

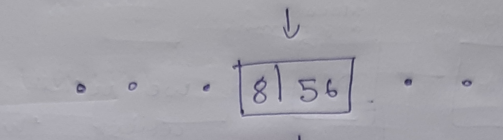
Insertion:

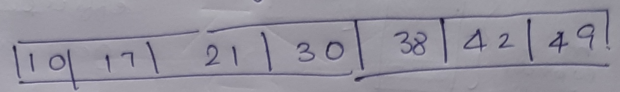
once we find the place to insert, if we insert directly, then the node will be overflow.

Eg: let B = 4

min no of keys = B-1 = 3

of keys = 3, 4, 5, 6



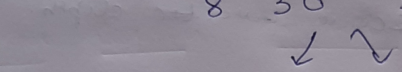


during overflow, perform split.

take the middle element & remove it.

Split the node into 2 parts

Insert the middle ~~no~~ inbetween the previous.

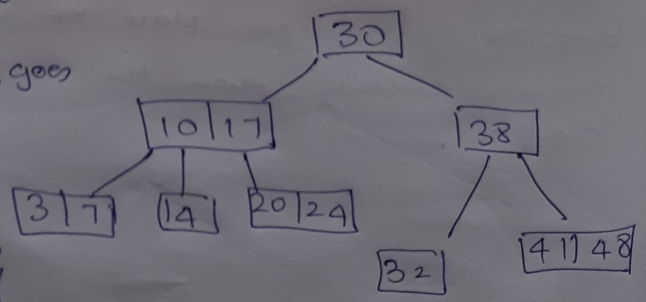


but now, the parent node might overflow. so keep splitting again.

If it overflows the root, split the root and that's it.

Insert 16

- less than 30, goes left.
- b/w 10 & 17, goes middle. So 14 becomes 14 16



Insert 2

- goes left.
- goes left.

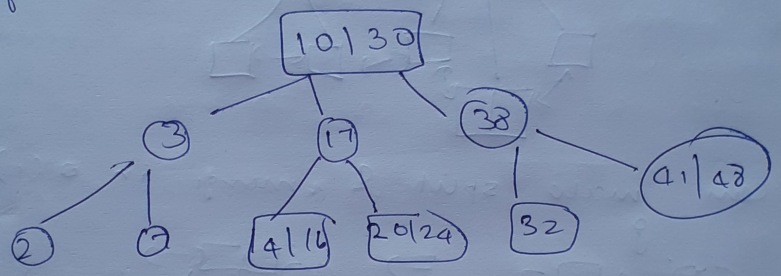
then 2|3|7

2 overflows the node

So take middle elem 3 and push it up.

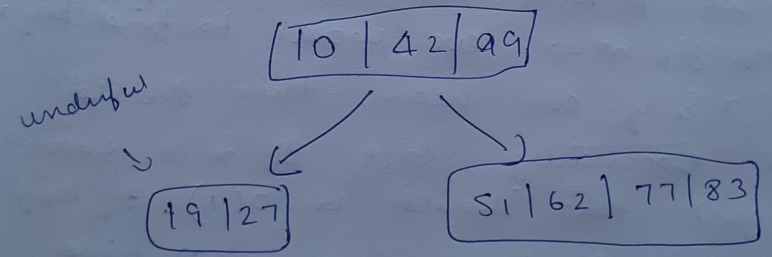
Now the 3|10|17 overflows. So split and send 10 to root.

So final tree is

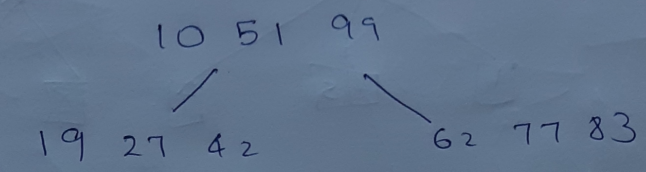


Deletion

node is underflow now.



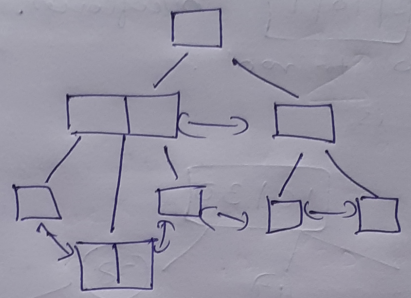
the right node is 1 over min. So take the left most key from right node, bring parent down, move sibling up



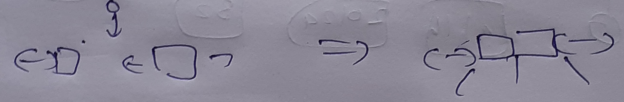
What if othe nodes are at min. Then can't take from that.

Then, move parent down, merge child nodes.

level linking



can make split & merge in O(1)

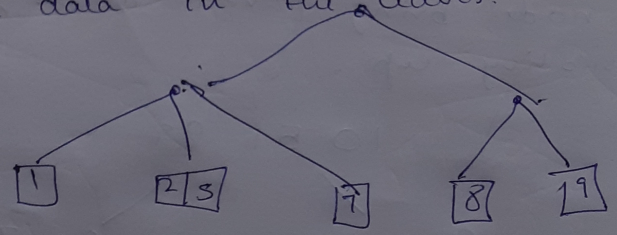


Finger search property

If we perform successful search for key y, now we want to find key x. If x & y are near, this should run especially fast. If x is successor of y(1) this should be O(1). If they are far away then log n time.

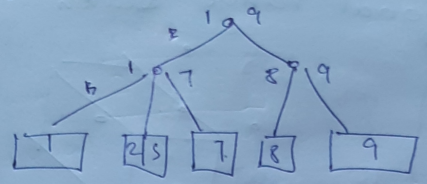
$$\log |rank(x) - rank(y)|$$

Store the data in the leaves.



Search:

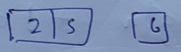
data struct aug. store min & max key in subtree.



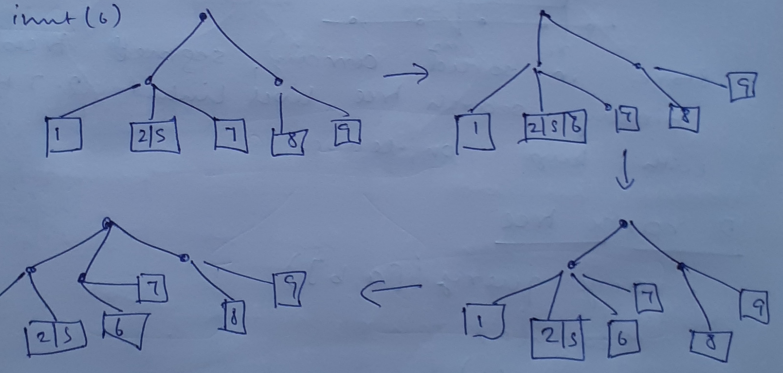
look at min & max and go to the required node while searching. O(lgn)

Insert:

if we have to insert 6, we do 2|5|6, now this is overflow. Instead of doing split & merge with parent, just split directly.



If the parent child bound is exceeded, then split parent into 2.



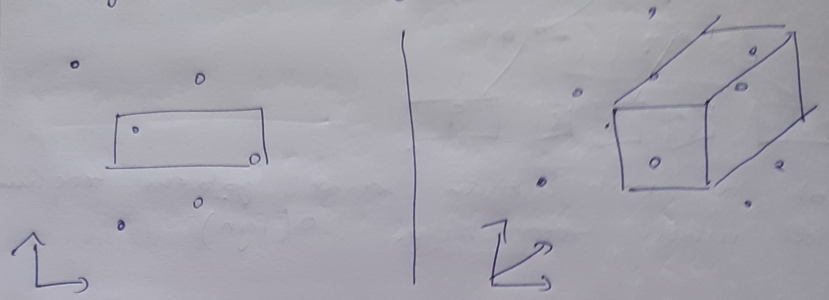
Search (x from y):

finger search.

- let v = leaf containing y.
- if v.min ≤ x ≤ v.max: regular search in v's subtree
- else x < v.min, v = v.level-left
  else x > v.max, v = v.level-right
- v = v.parent

Range trees

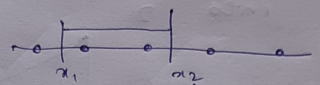
They solve orthogonal range search.



To preprocess them into a data structure; then will be queries. In 2D, queries are rectangle, in 3D, is 3D box

$O(\log d$ $O(\lg^d n + |\text{output}|)$

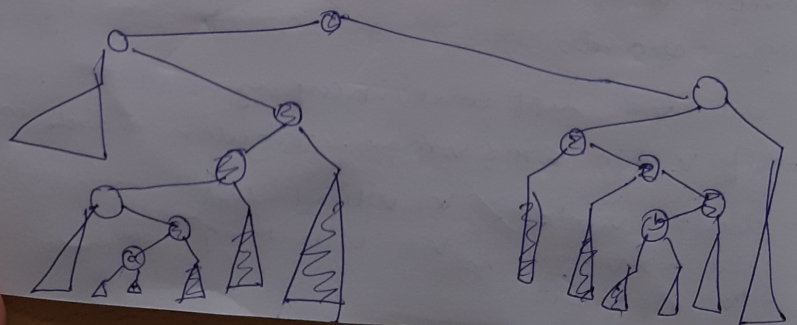
for 1D



Store in sorted array, segment trees, ~~etc~~ Binary search tree, with linked 2-3 trees with keys on leaves.

1D range tree.

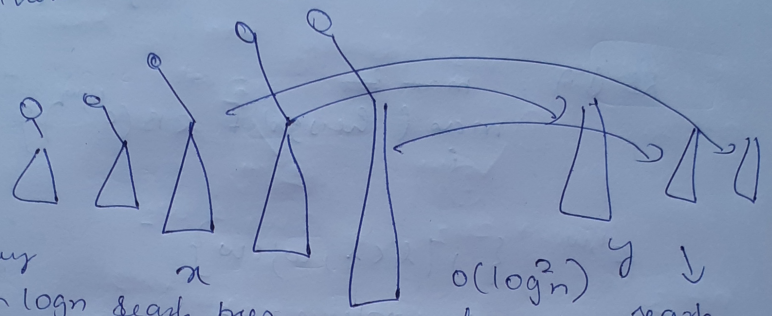
- range query ([a,b])
  - search(a)
  - search(b)
  - trim common prefix.



all the shaded elems are b/w a & b.
$O(\log n)$

2D range tree

- store a 1D range tree on all points by x.
- for each node v in x-tree, store 1D range tree by y on all points in them rooted subtrees.

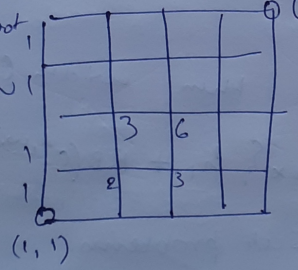


each lay lives in $\log n$ search trees.

$O(\log^2 n)$ for this search, search here again.

$\therefore O(n \log^{d-1} n)$ space

Dynamic programming.



Consider robot going form 1,1 to M,N

How many ways? Consider subproblem from every point from start.

runtime $O(nm)$

Make change

$S_1 = 1 \quad S_1 < S_2 < \cdots < S_n$ → bunch of coins.

What is min no of coins needed to make a change of n cents?

pick s_i:

$MC(N) = \min\{MC(N-s_i)+1\}$

Runtime: # subproblems (min & m terms)
$O(N \cdot m)$

Rectangular blocks

$\{1, 2, \ldots, n\}$ l_i, w_i, h_i

j on top of i, require $l_j < l_i$ & $w_j < w_i$

$RB(1..n) = \max_i \{ \text{height} + RB(c^{l_i, w_i}) \}$

$c^{l,w} = \{ j \mid l_i < l, w_j < w \}$

Runtime: #subproblems = n

#work per subprom = n (max over n elements)

$\therefore O(n^2)$ also this is the cost of making the sets c.

another method:

- sort by length & then width $(1, 2, \ldots, n)$
- $RB(1, \ldots n) = \max\{ h_1 + RB(c^{1,w_1}), RB(2, \ldots n) \}$

Runtime: n sub problems.

each subp: $O(1)$

$\therefore O(n)$ + cost for c & sort $n \log(n)$

if cost for c > $O(n \log n)$ then total = $O(n \log n)$

else total = $O(n^2)$

Advanced DP

Longest palindromic sequence

given a string $x[1:n]$ $n \geq 1$

Longest palindrome that is a subsequence.

charactu → carac

```
def L(i,j):
    if i==j : return 1
    if x[i] == x[j]:
        if i+1==j: return 2
        else
            return 2 + L(i+1, j-1)
    else:
        return { max( L(i+1,j) , L(i,j-1))
```

$$T(n) = \begin{cases} 1 & n=1 \\ 2T(n-1) & n>1 \end{cases}$$

$$O(2^{n-1})$$

how to add memoization?

→ look at $L[i,j]$ & don't recurse if already computed.

subprobms × cost for each SP assuming lookup is $O(1)$

This will be $O(n^2)$

Optimal BSTs

we have keys $k_1, \dots k_n$ to store in BST. $k_1 < k_2 \dots < k_n$

weights on each keys $w_1, \dots, w_n$ $k_i < k_j$

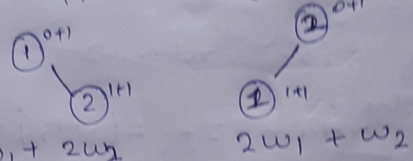
Find BST that minimises

$$\sum_{i}^{n} w_i (\text{depth}_T(k_i) + 1)$$

So height weight nodes should have lower depth.

we have exponentially many trees.

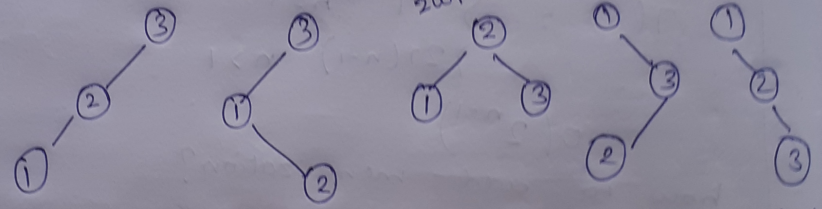
$n = 2$.



$w_1 + 2w_2$

$2w_1 + w_2$

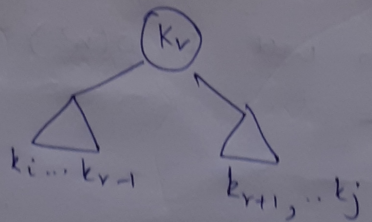
$n = 3$



Doing this for all will take exponential time.

greedy soln

+ pick k_r in some greedy action.



if k_r = highest, then ⊖ it going to unbalanced.

and this failed for many cases.

Using DP:

We don't know what node to use for ~~DP~~ root, so guess.

$$e(i,j) = \begin{cases} w_i & , i \geq j \\ \min_{i \leq r \leq j} \left(e(i,r-1) + e(r+1,j) \right) + ~~w_{ij}~~ \\ \qquad\qquad w(i,j) \end{cases}$$

$w(i,j)$ ↓ sum of all weights $i \to j$

Alternating Coin game

Row of n coins of values $V_1 \cdots V_n$

Select either the first or last coin from the row, remove permanently but add it to value and recede the value.

4 42 39 19 25 6 ↓1

pick 6.

4 42 39 19 25 ↓2

pick 25

4 42 39 19 ↓1

pick 4

4 42 39 ~~19~~

↓2 pick 42

42 39

↓2 ↓1

First player ~~picks~~ a $V_1 + V_3 + \cdots > V_2 + V_4 + \cdots$ pick first based on that.

how to maximize the score? assume you move first

V(i,j) max we will def coin if it is our turn & only coin V(i) → V(j) remain.

V(i, i+1) → pick max.

$v_1 \ldots v_i \ldots v_j \ldots v_n$

V(i,j) = max { range is i+1, j $+v_i$; range is i, j-1 $+v_j$ }

pick v_i | pick v_j

opponent moves in this range. with the worst can based on what opponent does.

V(i+1, j)) sub problem with opponent pick.

~~V(i+1)~~

we can guaranteed

$\min\{V(i+1, j-1), V(i+2, j)\}$

$V(i,j) = \max\{ +v_i , +v_j \}$

$= n^2 O(1) = O(n^2)$

All pairs shortest path in graphs.

Situation	algorithm	time
unweighted	BFS	$O(V+E)$
non-neg weights	Dijkstra	$O(V \lg V + E)$
gens	Bellman-ford	$O(VE)$
acyclic (DAGs)	topological sort + Bellman ford	$O(V+E)$

given $G = (V, E, w)$

find $\delta(u,v)$ for all u, v

the other methods $O(V^2)$

become $O(V^2 \lg V + VE)$

better general case algorithm, $O(V^2 E)$

Johnson's algorithm

$O(V^2 \lg V + VE)$

Dynamic Programming.

subproblem: what is weight of shortest path from u to v. (But this is cyclic)

So let it be.

$d_{uv}^{(m)}$ = weight of shortest path that uses at most a given no of edges. (m edges)

2) guessing: last edge (x, v)

3) recurrence: $d_{uv}^{(m)} = \min(d_{ux}^{(m-1)} + w(x,v))$ for x in V

$d_{uv}^{(0)} = \begin{cases} \infty & \text{otherwise} \\ 0 & \text{if } u = v \end{cases}$

④ topological ordn.

for $m = 0, 1, \ldots, |V|-1$;

for u, v in V

⑤ original problem.

ensure no neg weight

(Use Bellman ford

& look at d_{uv}

diagonal, and

else if val is -ve)

for m in range(1, ~~m~~, size of |V|

for u in V;

for v in V,

relaxation step (triangle inequality) (if $d_{uv} > d_{ux} + d_{xv}$:

$d_{uv} = d_{ux} + d_{xv}$

$O(V^4)$

How to make this better?

Matrix Multiplication

$C = A \times B$

$n \times n \quad n \times n \quad n \times n$

Standard $O(n^3)$

Strassen $O(2n^{2.8})$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

define $\odot$ as $+$ & $\oplus$ as min

$$D^{(m)} = D^{(m-1)} \odot W = W^{(m)} \quad W^{(0)} = \begin{pmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{pmatrix}$$

$$D^{(m)} = \left(d_{ij}^{(m)}\right)_{ij}$$

$$W = \left(w(i,j)\right)_{ij}$$

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

$$D^{(m)} = D^{(m-1)} \odot W = W^{(m)}$$

$\downarrow$ DP

$W^{(1)} \odot W^{(1)} = W^{(2)}$

$W^{(2)} \odot W^{(2)} = W^{(4)}$

} repeated squares.

$O(V^3 \lg V)$

Transitive closure

is there a path from i to j.

$\odot$ is AND ring

$\oplus$ is OR

$O(n^{2.3728\ldots})$

2nd approach DP - Floyd Warshall algo

① subproblems $c_{uv}^{(k)}$ = weight of shortest path $u \to v$ whose intermediate vertices $\in \{1, 2, \ldots, k\}$

② guessing: is $k \in$ path?

③ recurrence

$$c_{uv}^{(k)} = \min \left\{ c_{uv}^{(k-1)}, c_{uk}^{(k-1)} + c_{kv}^{(k-1)} \right\}$$

$$c_{uv}^{(0)} = w(u,v)$$

$O(V^3)$

Johnson's algorithm:

① find function $h : V \to \mathbb{R}$ such that

$$w_h(u,v) = w(u,v) + h(u) - h(v) \geq 0 \quad \forall u,v.$$

if all are +ve, we can use Dijkstra instead of Bellmanford.

② Run Dijkstra on (V, E, w_h) $\forall u,v$. $\to \delta_h(u,v)$.

③ claim that $\delta(u,v) = \delta_h(u,v) - h(u) + h(v)$

how to find h?

$w(u,v) + h(u) - h(v) \geq 0$

$h(v) - h(u) \leq w(u,v)$ $\forall u,v$

system of distance constraints.

This works when there are no -ve weight cycles.

Run Bellman ford once. So it handles neg weight cycles then finds valid h.

Greedy algorithms: Minimum spanning tree

tree - connected acyclic graph.

Spanning trees - contains all vertices.

= subset of edges of G that form a tree & hit all vertices of G.

Minimum spanning tree:

given graph $G = (V, E)$

& edge weights $w: E \to R$.

find a spanning tree of min total weight $= \sum_{e \in T} w(e)$

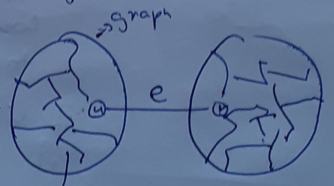
Greedy properties

- optimal sub structure: If you can solve sub problems optimally, then you can solve original problem.
- greedy choice property: locally optimal choice leads to globally optimal. Sdn (In DP, we keep guessing)

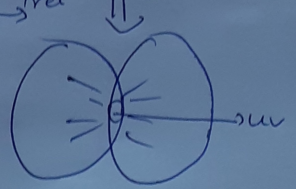
optimal substructure for MST:

if $e = \{u, v\}$ an edge for some MST

if e is deleted, we get 2 components. (Thru lots of other edges)



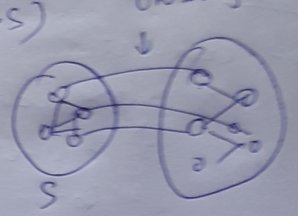
- contract the edge, merge u & v.



Thru can be duplicate edges if u & v have other connected nodes. take the min of edges for each connected node.

Greedy choice property of MST

- consider any cut $(S, V-S)$
- let e be least weight edge crossing the cut.
  $e = \{u, v\}$
  u in S v in $V-S$



That edge is in MST (guaranteed)

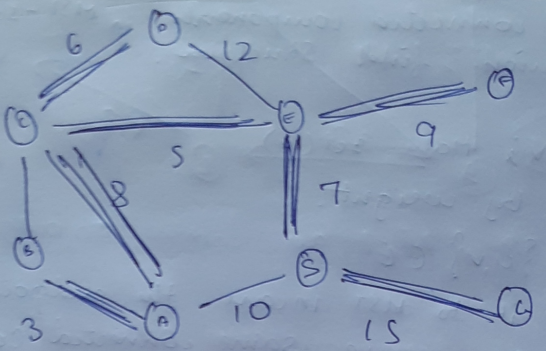
Prim's algorithm

Choose a single vertex as S. Then take minimum weight edge and ~~remove~~ cut.

- Maintain priority queue Q on $V-S$. $v.key = \min \{w(u,v) \mid u \in S\}$
- Initially Q stores V.
  $S.key = 0$ for arbitrary $S \in V$.
  for $v \in V - \{S\}$, $v.key = \infty$

until Q is empty (spanning tree on whole graph)
  u = Extract-Min(Q) => add u to S.
  for $v \in Adj[u]$
    (all neighbours v of u)
    (basically new node added to S, removed from $V-S$, so now the connection of the other node needs to be seen to find the min edge).
    if v in Q
      if $w(u,v) < v.key$
        $v.key = w(u,v)$
        $v.parent = u$

Eg:



Initially

Key(S) = 0

Key(A ... G) = ∞

at first smallest is S
add S to S.

neighbours of S, A E G are outside S.

weight S→E = 7 so key(E) = 7

Similarly Key(A) = 10

Key(G) = 15

now, smallest is E, so add E to S.

S now has S, E.

& update neighbours of E.

Key(C) = 8, Key(F) = 9

Key(D) = 12.

No Key(&C) = 8 is smallest.

∴ add C to S, S = S, E, C.

update neighbours of C.

Key(B) = 14

Key(D) = 6 next add D.

Key(A) = 10 next add A & B.

the bold lines denote MST.

time same as dijkstra.

Kruskal's algo

- maintain connected components in MST - so far in a union find structure.
- $T = \phi$
- for $v \in V$: Make-set(v)
- Sort(E) by weight.
- for $e = \{u, v\} \in E$
  - add it to MST if the endpoints of edge are not in same connected component. (Call find-set twice to see whether they are equal)
  - do a union(u, v)

$O(\text{Sort}(E) + E\alpha(V) + V)$

↓ union

More greedy algorithms:

Continuous coins make change

N nodals $\{1, 2, \ldots, N\}$, k_i weight, w_i available each nodal how value c_i/kg achieve some value $T

$\sum_i k_i c_i = T$, minimize $\sum k_i$

Sort by c_i (decreasing)

$c_1 \; c_2 \; \ldots \; c_n$

if $\frac{T}{c_1} > w_i$, then use it

Process scheduling

time t_i $i-n$

order P_i $i-n$ i

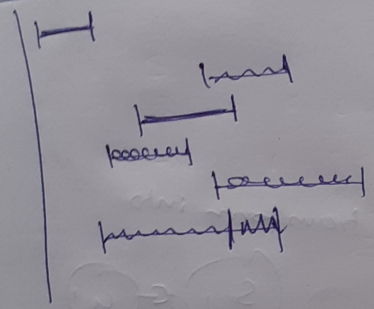
completion time $\sum_1 t_{P_j}$

minimize - Avg completion time $\frac{\sum C_{P_i}}{n}$

Sort by t_i (increasing)

done.

Event overlap



3 diff R procs. which can run w/o overlap

Find min no of procs.

Sort it, every time another begin before ending current, start a new proc. keep track of all procs

Incremental Improvement : Max flow, min cut.

Flow n/w

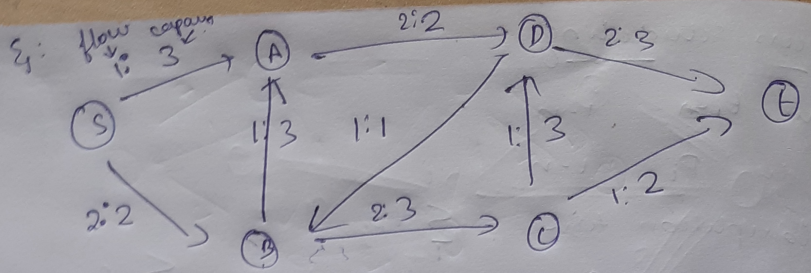
$G(V, E)$

two distinguished vertices, source s & sink t.

each edge going to have non neg capacity $c(u,v)$

if $u,v \notin E$ then $c(u,v) \geq 0$.

Eg: flow capacity

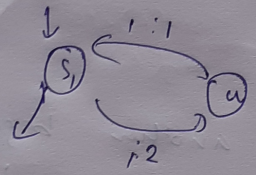


at A, flow | Eg | as coming so outgoing is 2.

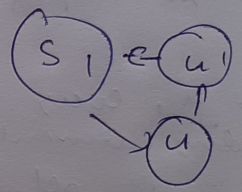
Similarly for all

By reducing flow on some edges, it can be increased on other. Flow should'nt exceed capacity.

- no self loop allowed.
- for edges like



transform into



Flow
- satisfies capacity constraint
- flow conservation
- Skew symmetry

Value of flow f denoted $|f|$

$$|f| = \sum_{v \in V} f(s,u) = f(s,V)$$

Simple property

$f(x,x) = 0$ (no self loop)

$f(a,b) + f(b,a) = 0$ (Skew symmetry)

$f(x,y) = -f(y,x)$

what goes out of source goes to sink.

Cut

a partition of nodes

a cut (S,T) of a flow $G=(V,E)$ is a partition V so that $s \in S$, $t \in T$.

If f is flow on G, flow on cut is $f(S,T)$

$$f(S,T) = \underset{S_a\ S_b}{(2+2)} + \underset{(d,a)\ d,b\ d,c\ d,t}{(-2+1-1+2)}$$

capacity of cut $= c(S,T) = \underset{S,a\ \ S,b\ \ d,b\ \ d,t}{3+2+1+3}$

$= 9$

value of any flow is bounded by capacity of any cut.

Residual n/w

$G_f(V, E_f)$ strictly positive residual capacities.

$G_f(u,v) = c(u,v) - f(u,v) > 0$

edges in E_f admit more flow.

If $(v,u) \notin E$ then $c(v,u) = 0$ but $f(v,u) = -f(u,v)$

↓ because of extra edges in residual n/w