Table Of Contents

Time Complexity ($O(n)$, $\Omega(n)$, $\Theta(n)$)

* $f(n) \in O(g(n))$ if for some c & n_0, for all $n > n_0$, $f(n) \leq c \times g(n)$.

* $f(n) \in \Omega(g(n))$ if for some c & n_0, for all $n > n_0$, $f(n) \geq c \times g(n)$

* $f(n) \in \Theta(g(n))$ if for some c_1, c_2 & n_0, for all $n > n_0$, $c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$

Note; c_1, c_2, c, n_0 must be constants

Q) Is $n^{2.003}$ in $O(n^2 \log n)$?

$\Rightarrow n^{0.003}$ vs $\log n$

Is there some n_0 where $\forall n > n_0$, $n^{0.003} \geq c \times \log n$

No, since for large values of n, $n^{0.003} \not\leq c \times \log n$

(As we increase c, we can increase n to always ensure $n^{0.003} \geq c \times \log n$)

Q) For two functions f & g, is it true that one of

a) $f(n) \in O(g(n))$
b) $f(n) \in \Omega(g(n))$
c) $f(n) \in \Theta(g(n))$, holds true?

→ Sinusoids (sin vs cos)

* Master's Theorem; (Extended)

- $T(n) = a \times T(n/b) + f(n)$, $a \geq 1$ & $b > 1$

($f(n)$ should be a growing function)

Case 1: $f(n) = O(n^{\log_b a - \varepsilon})$ → Some constant ($\varepsilon > 0$)

Case 2: $f(n) = \Theta(n^{\log_b a} \times \log^k n)$

Case 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$

⇒ In Case 1: $O(T) = \Theta(n^{\log_b a})$ → $n^{\log_b a}$ dominates $f(n)$

Case 2: $O(T) = \Theta(n^{\log_b a} \times \log^{k+1} n)$

Case 3: $O(T) = \Theta(f(n))$ → $f(n)$ dominates $n^{\log_b a}$

Sorting Algorithms;

⇒ QuickSort (selection)

⇒ $T(n) = T(n/2) + O(n)$ → $\Theta(n)$?

⇒ ~~$f(n) =$~~ ~~$n^{\log_b a}$~~ $f(n) \geq n^0$

QuickSort in worst case;

$T(n) = T(n-1) + O(n)$

∴ $O(n^2)$ solution in worst case

Quicksort Optimal Pivot Selection; (Quickselect)

- Divide your input into groups of size 5, the store all medians of each group in an array.
- Then recursively find median of this array (Median of medians) (After each recursive call, do partition)

(Assume n = Power of 5)

ex) 12, 7, 15, 4, 6, 8, 9, 20, 15, 11, 17, 8,
12, 3, 25, 14, 17, 1, 21, 30, 18, 9,
3, 8, 5, ~~[illegible]~~

G_1	G_2	G_3	G_4	G_5
12	8	(17)	14	18
(7)	9	18	(17)	9
15	20	12	1	3
4	15	3	21	(8)
6	(11)	25	30	5

⇒ Array of medians = 7, (11), 17, 17, 8

Median of medians = 11 //

⇒ Now, If we sort G_1 to G_5 in order of medians

G_1 G_5 (G_2) G_3 G_4 } n/5 groups

MOM lies here.

⇒ Now G_2 will have atleast $\left(\frac{n}{10}-1\right)$ groups below it.

$\left(\frac{n}{10}-1\right) \times$ groups

⇒ Now each of these $\left(\frac{n}{10}-1\right)$ groups will have 3 elements less than MOM.

∴ $\left(\left(\frac{n}{10}-1\right)\times 3\right)$ elements are less than MOM & then 2 elements in group of MOM are also less

∴ $\left(\frac{n}{10}-1\right)\times 3+2$ // minimum

$\therefore \frac{3n}{10} - 1$, and $\frac{7n}{10} + 1$ are the 2 groups

So we will always have groups of $\frac{3n}{10} - 1$ elements in worst case.

⇒ Time Complexity;

⇒ $T(n) = T(n/5) + T(7n/10) + O(n)$

↓ To get MOM

⇒ Let us assume, $T(n) \leq an$

⇒ $T(n) \leq a(n/3) + a(7n/10) + cn$

⇒ $T(n) \leq n(c + 9a/10)$

At $a \geq 10c$

$T(n) \leq an$, So choose $a \geq 10c$

(Now if we choose groups of size 3, then the complexity becomes worse) (Check)

Code for above;

```
find smallest (A, size = n, k)
1. Divide A into groups of size 5.
2. Find median of each group.
3. M = array of medians of each group
4. mom = find smallest (M, n/5, n/10)
5. l = Partition (A, mom) } Number of elem before mom
```

6. If $k = l+1$, return num

$k \le l$, find smallest $(A[\ldots l], l, k)$

$k > l+1$, find smallest $(A, [l+2, \ldots n], n-l-1, k-l-1)$

~~return -1~~

Divide & Conquer:

- $T(n) = \sum_{i=1}^{k} T(n_i) + D(n) + C(n)$

  $D(n)$: Time to divide into k parts

  $C(n)$: Time to combine the k parts.

⇒ In mergesort, $D(n) = O(1)$, $C(n) = O(n)$

⇒ In quicksort, $D(n) = O(n)$, $C(n) = O(1)$

get pivot

Partition Based Approach

1) Matrix Multiplication:

- Recursion divide & conquer

  (If size 2×2 matrix, then calculate manually)

⇒ $T(n) = 8 \times T(n/2) + O(n^2)$

Recursive call; To add the submatrices

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$$
$$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11}$$
$$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10}$$
$$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11}$$

$O(n^2)$ + 8 x 7

⇒ Using masters theorem

$$n^{\log_2 8} = n^3$$

⇒ Since $n^{\log_2 8} > n^2$

⇒ So its case 1

$$\therefore O(T) = \Theta(n^3)$$

Note: To solve $(a+ib)(c+id)$
$= ac - bd + i(ad+bc)$
with 3 multiplications,

do $T_1 = ac$
$T_2 = bd$
$T_3 = (a+b)(c+d)$ } 3 multiplications

⇒ $ac - bd = T_1 - T_2$
$ad + bc = T_3 - T_1 - T_2$

- (Use Strassen's Algo to speed it up by reducing 8 recursive calls to 7)
- (We try reduce multiplications so that the time complexity changes)

⇒ So the recurrence relation becomes

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$\Rightarrow f(n) = \Theta(n^2)$$

$$n^{\log_2 7 - \delta} > n^2$$

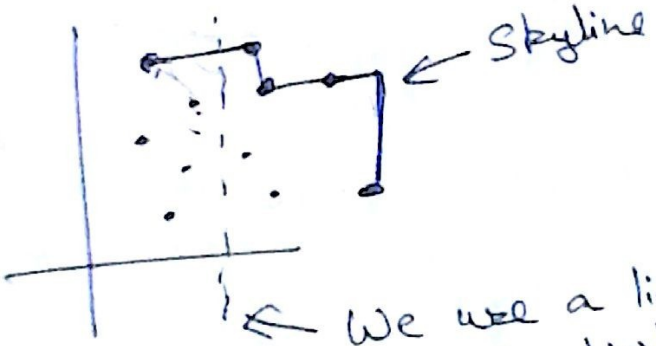
$\therefore$ Case 1; $T(n) = \Theta(n^{\log_2 7})$

⇒ We are however disregarding the huge $14 \times \Theta(n^2)$, so when the size of n becomes significantly small, then we can use the standard $\Theta(n^3)$ algorithm.

---

2) Skyline Points;

- Divide & Conquer Approach

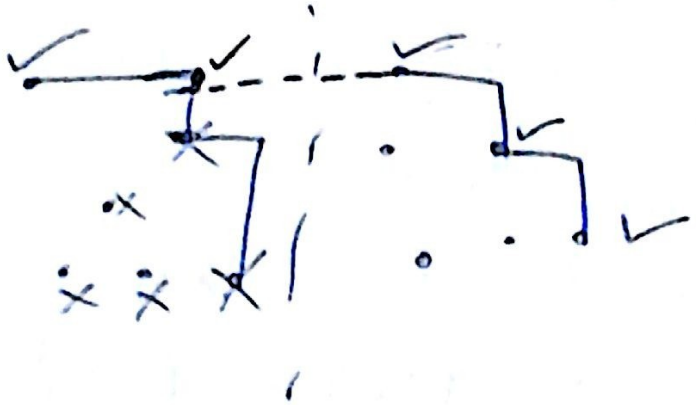


⇒

⇒ Then we ~~combine~~ check the 2 skylines to get a joint skyline

- Since the points in final skyline either belong to
  - left skyline or right skyline.

⇒ In the above example

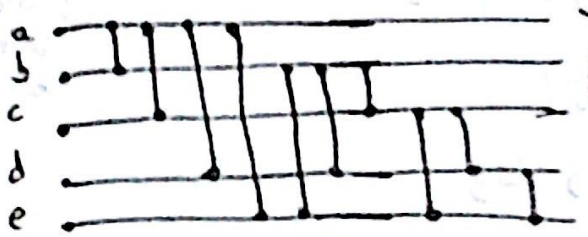


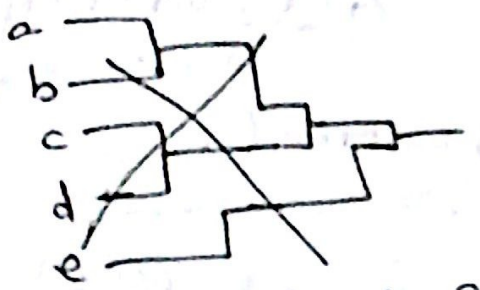
→ To merge the 2 skylines, take point with highest y on right and draw line parallel to x, and discard all points on left with y less that that line.

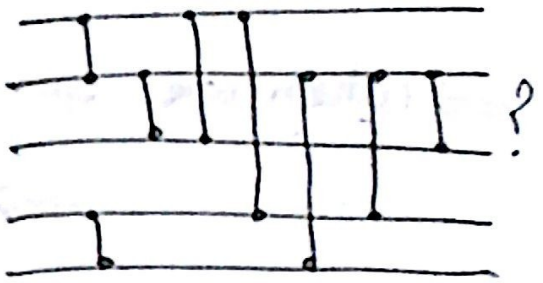
⇒ You can solve right subproblem first & then discard points on left before trying to solve

——x——

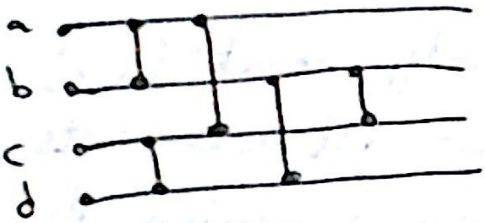
Sorting in Networks:







⇒ Using Divide & Conquer we can reduce the number of comparisions



———x———

⇒ Bitonic Sequences:

- Sequences which increase & then decrease

ex: 11, 16, 20, 23, 34, 17, 9, 5, 1

(A sequence which increases & decreases after a circular shift is still a bitonic sequence)

ex: 5, 1, 11, 16, 20, 23, 34, 17, 9

↓ This can be shifted to

11, 16, 20, 23, 34, 17, 9, 5, 1

↓ Bitonic

———x———

Bitonic Sort; (Depth of log(N) ⇒ for parallel processing)

⇒ Sequence ⇒ 11, 16, 20, 23, 34, 17, 9, 5 } Given bitonic sequence

$L(x) = \{11, 16, 9, 5\}$

$R(x) = \{34, 17, 20, 23\}$

⇒ When we compare $A[i]$ with $A[i + n/2]$ $\forall\ i = 0$ to $n/2$

⇒ $L(x) = \min(a, b)$

$R(x) = \max(a, b)$

⇒ Both $L(x)$ and $R(x)$ are bitonic sequences of length $n/2$.

⇒ So now recursively sort $L(x)$ & $R(x)$, note how all elements of $L(x) \leq R(x)$, so merging the sorted arrays is just $L(x) + R(x)$.

—— x ——

Note; Even if we circular shift the bitonic sequence, the values that are compared are the same, so any bitonic sequence can be sorted.

—— x ——

Q) 11, 16, 26, 23, 34, 17, 9, 5

⇒ $L(x) = \{11, 16, 9, 5\}$

⇒ $L'(x) = \{9, 5\}$ ⇒ $L''(x) = \{5\}$, $R''(x) = \{9\}$

$R'(x) = \{11, 16\}$ ⇒ $L''(x) = \{11\}$, $R''(x) = \{16\}$

$R(X) = \{34, 17, 20, 23\}$

$\Rightarrow L'''(X) = \{20, 17\}$

$\Rightarrow L'''(X) = \{17\}$

$R'''(X) = \{20\}$

$R'''(X) = \{34, 23\}$

$\Rightarrow L'''(X) = \{23\}$

$R'''(X) = \{34\}$

~~$L''(X) + R''(X) = \{5, 9\}$~~

$L(X) = \{5, 9\} + \{11, 16\}$

$= \{5, 9, 11, 16\}$

$R(X) = \{17, 20\} + \{23, 34\}$

$= \{17, 20, 23, 34\}$

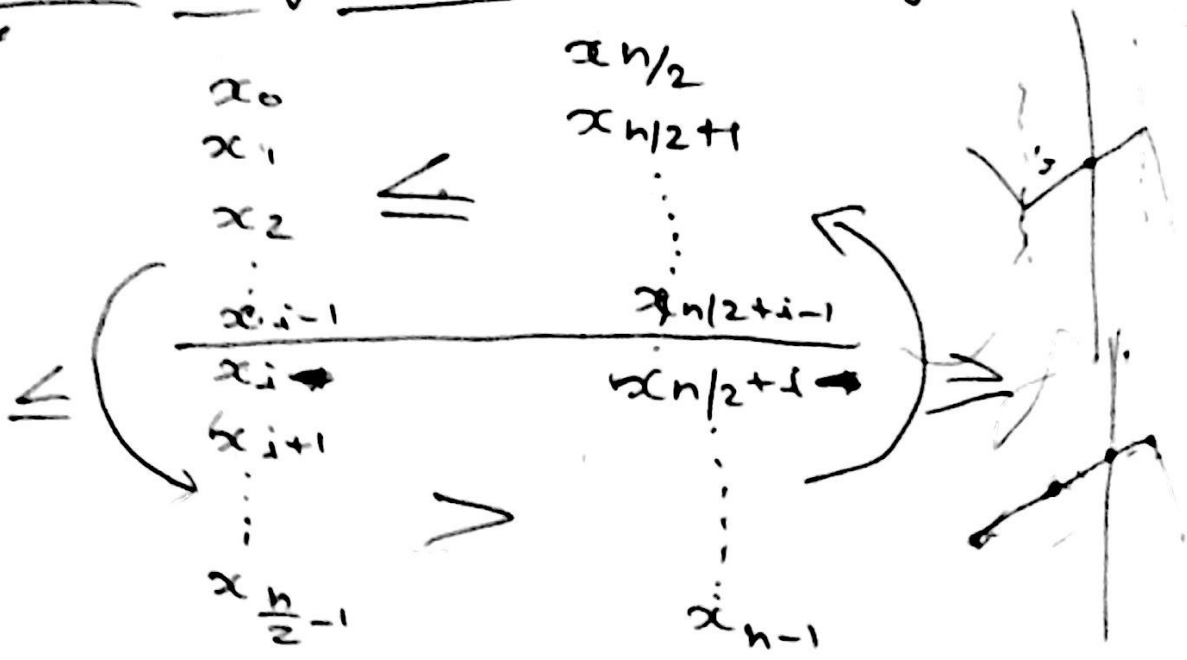
$L(X) + R(X) = \{5, 9, 11, 16, 17, 20, 23, 34\}$

Bitonic Sorter Algo:

⇒ **Note:** To create the bitonic sequence from any arbitary sequence, we start with 2 elements, then we go to pairs of 4 elements.... and at each time, we join A to reverse of B (where A & B are the pair), to get a longer bitonic sequence. (But for this we need sorted A & B, so we use our bitonic sort at every stage). This is recursive.

Once we have the bitonic sequence do bitonic sort one last time to get ans

Note; Unique Crossover Property;



- If $x_0, \ldots\ldots, x_{n-1}$ is a bitonic sequence then for some i, we can draw a line, and we can see that $L(x)$ & $R(x)$ are inc + dec, dec + inc sequences so they are also bitonic.

Bitonic Sorter;

$$S(n) = 2S(n/2) + O(n \log n)$$

$$\therefore S(n) = O(n \log^2 n)$$

Closest Pair Problem;

Note; Standard = $O(kn^2)$
↓
k dimensions.

⇒ Suppose k=1, (so points are on a single line)

- ⇒ $O(n\log n + n)$ → Sort then traverse
  ⇒ $O(n\log n)$
- We can also use, Divide & Conquer → Divide over median
  where smallest(n) = smallest(0, n/2) or smallest(n/2, n) or dist b/w rightmost point of first half & leftmost of second half.

$$T(n) = 2 \times T(n/2) + O(n) \quad (\text{can be brought down to } O(1))$$
$$\therefore T(n) = n\log n$$

For 2D space;

(Split over median of x)

- The divide step remains the same, but the combine step changes.

⇒ When we combine, for each point on left, we can search for only points on right which are lesser than d* (ie. within a circle of radius d*). Now we know that on the right, no elements are closer than dR, so no element pairs are closer than d*, so we can at most have 3-4 points within the circle having atleast d* distance,

→ current min (dL, dR)

Because if its greater than we don't care

between each of them.

∴ For each of the n/2 points on left we check atmost 3-4 points.
(Lets call these friends)

(To find friends of all points, it takes O(nlogn))
(Sorting based on y)
(Then binary search)

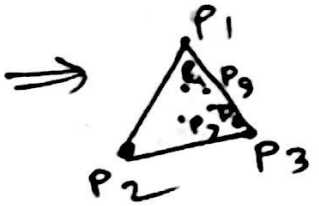
∴ $T(n) = O(n \log n) \times$ ——

(For d dimensions,
$T(n) = O(n \log^{d-1} n)$ → Can be reduced to O(nlogn) with more techniques)

—— x ——

* Convex Combinations; (Find Convex Hull CH(P))

(Convex polygon)



⇒ Any point within convex hull can be represented by α, β, ... and the ↑ Constants points on the convex hull

⇒ Here P_1, P_2 & P_3 form the convex hull

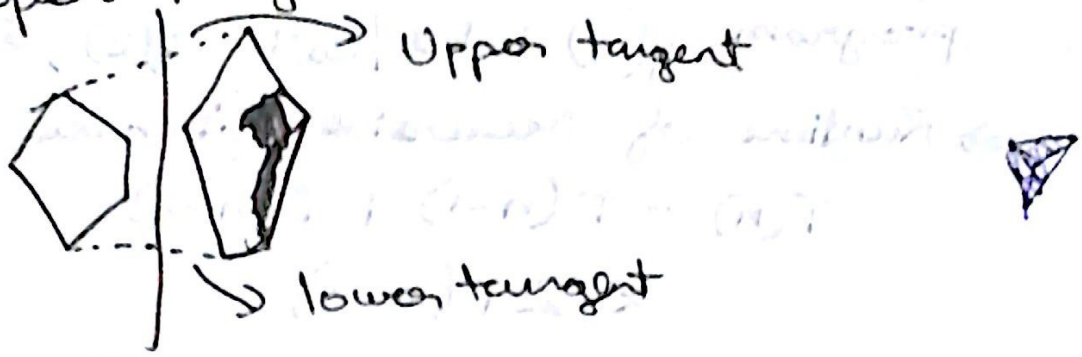
1) Divide & Conquer Approach;

$T(n) = T(n/2) \times 2 + O(f(n))$

- Here in the merge step, take max & min of y coordinates on both sides, then just add those

to the convex hull removing inner boundaries of previous convex hulls (of left & right halves).

⇒ These two lines we add are called upper tangent & lower tangent.



- But this might fail for some cases, so we use an $O(n)$ solution for getting the tangent.

(With selecting rightmost x of left & leftmost x of right)

⇒ Then we move either the left point or the right point up the convex hulls until we get it to be the upper tangent (similarly for lower tangent).

If line is

↓

then move left up

else

↓

move right up.

$\therefore T(n) = 2 \times T(n/2) + O(n)$

$\therefore T(n) = n \log n$ //

— x —

Dynamic Programming;

(We can do it with recursion but we get repeated subproblems)

ex; If we calculate fibo with a recursive program, f(7) takes/calls f(2), 8 times.

⇒ Runtime of recursive fibonacci

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

$\hookrightarrow \therefore 2^n$ is the runtime

→ Actual runtime ⇒ $\Theta(\phi)^n$ → Golden ratio (1.7)

—×—

*Note;

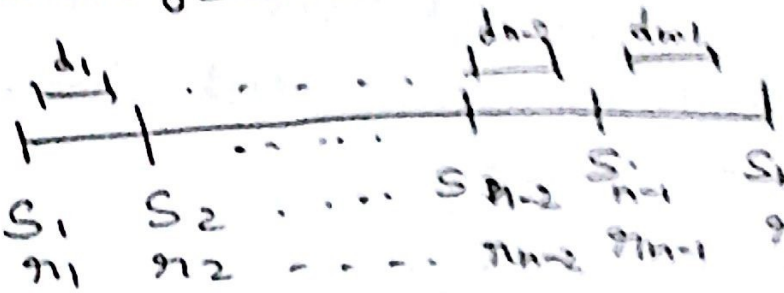
*⇒ If we get non-unique subproblems, then we use D.P.

—×—

Fibnocci (with storage) (with recursion)

```
calc
fibo(n):
    sum=0
    if mark(fibo(n-1))
    {  sum += fibo(n-1)
    }
    else fibo calc(n-1)
         sum += fibo(n-1)
    if (mark(fibo(n-2))
        sum += fibo(n-2)
    else
        calc(n-2)
        sum += fibo(n-2)
    fibo(n) = sum
    mark(fibo(n)) = 1
```

1) Railway Station Problem:

⇒ 

⇒ How many stations can we select such that we get maximum revenue and gap b/w 2 stations is atleast R.

⇒ Let us denote Last(i) = Index of previous possible station if the ith sation is included. → ie gap of atleast R

Ans(i) = max (a Revenue(i) + Ans(Last(i)), Ans(i-1))

or just plain D.P ← ⇒ Assuming this is done recursively with storage of answers, then T(n) = O(n) (Assuming Last(i) is precomputed)

**Note; Memoization;

- Storing answers to subproblems, so that we can avoid some recomputations. (Used in DP)

Note; If we do the above problem without memoization then in the worst case the solution could be ~~O(n!)~~ O(2^n) (Recursive)

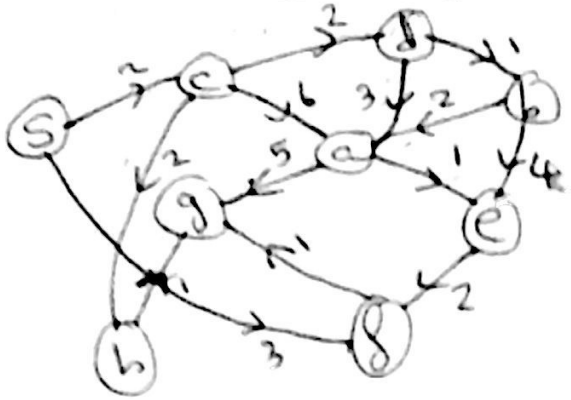
(Worst case; T(n) = T(n-1) + T(n-1) ↑ T(last(1)))

---

2) Shortest Paths in DAGs; (Directed Acyclic Graph)

- Given node v, get shortest distances to all other nodes.

ex;



⇒ Considering s as source,

s → a = 7
b = 5
c = 2
d = 4
e = 8
f = 3
g = 4
h = 4

s, a = s, c + w(c→a)
or
s, d + w(d→a)
or
s, b + w(b→a)

⇒ s, c = 2
s, d = s, c + w(c→d)
= 4
s, b = s, d + w(d→b)
= 5

⇒ s, a = 7
⇒ s, e = s, a + w(a→e)
or
s, b + w(b→e)
= 8
⇒ s, g = s, a + w(a→g)
or
s, f + w(f→g)
or
s, h + w(h→g)
⇒ s, f = s, e + w(e→f)
or
w(s→f)
= 3

$s \to h = s \to c + w(c \to h)$

$= 4$ //

$\therefore s,g = 4$ //

⇒ Given a source (in an Directed Acyclic Graph), we can find shortest distance to all other nodes with D.P. //

T = O(M+N) //

```
Find dist (i)
    for j | vi vj ∈ E do
        if d(j) = ∞ → Initialized & Not Computed
            d(j) = Find dist (j)
        endif
        d(i) = min (d(i), d(j) + wt(i,j))
    return d(i).
```

---

3) Knapsack Problem; (Generalized) (0,1 - Knap Sack)

(If we have rational weight then rationalize denominator)

- Our best solution is O, sack
- Now starting from n items & weight W.

⇒ Now starting from n items & weight, W.

1) If o_n is included, in O, our profit is p_n & total profit = p_n + (profit of sack with weight $W - w_n$.

2) Else, total profit = profit in n-1 elements with weight W.

⇒ $S(n,W) = \max\{p_n + S(n-1, W-w_n), S(n-1,W)\}$ // (Recursive calls required / Recursively calculate)

(Recursive 2^n solution)

⇒ Using 2D DP, O(n x W) solution // (Don't need to fill entire table)

Note; Knapsack Problem is called [pseudo-polynomial] algorithm. (Since $O(nW)$ → W can go very large even with input size of W being small (However if n↑, input size also increases)

dependent on magnitude of W.

* Note; Counting Sort (Buckets) → Dependent on max of magnitude of each input
ex: Numbers (n) with values less than n^2, $O(n^2)$

Pseudo Polynomial ← $O(n+L)$ → Max magnitude of input.
↓ number of elements

Note; Radix Sort → Based on digits

* Note; Pseudo-Polynomial algorithms are those whose runtime depends on magnitude of input.

4) Fractional Knapsack; (We can take parts (fractions) of objects)

- A simple greedy solution.

5) LCS (Longest Common Subsequence)

ex: LCS of
a) ACCGTACTGAG
b) CAGTAATGACG
is
AGTATGAG ⇒ LCS

⇒ Let LCS = $\langle u_1, u_2, \ldots, u_k \rangle = U$
where string1 is of length n & string2 is of length m.

$u_k = S_m = T_n$ &

a) ⇒ If $S_m = T_n$, then the rest of U is the LCS of $S[1, m-1]$ & $T[1, n-1]$

b) ⇒ If $S_m \neq T_n$, then rest of U is the LCS of $(S[1, m-1]$ & $T)$ or $(S$ & $T[1, n-1])$ → Maximum of these.

⇒ We can do this in a recursive (top down) manner (Direct recursion with 2 conditions & base case) or we can use DP for an $O(n \times m)$ solution.

---

* DP solution; (Bottom Up)

$O(n \times m)$

$A[n-1][m-1]$ has the length of longest common subsequence)

```
for i=0; i<n; i++
    for j=0; j<m; j++
        if str1[i] == str2[j]
            A[i][j] = A[i-1][j-1] + 1
        else
            A[i][j] = max(A[i-1][j], A[i][j-1])
```

---

* (Q) How to convert one string to another with only add, remove, replace characters, or exchange adjacent 2 characters with minimum cost/operations. (Standard DP)

DP (vs) Divide & Conquer;

1) If subproblems in recursive solution are repetitive, use D.P. (Memoization)
2) If subproblems in recursive solutions are unique, use Divide & Conquer.
3) In DP, optimal solutions ~~embed~~ embeds optimal solutions to subproblems. (Optimal Substructure property).
4) In DP, trace table backwards to get actual solution & not just length etc. (Most of the times)

———— x ————

Begin Of Mid 2

Greedy Algorithms;

Generic MST Algorithm;

- Cut ⇒ Partition vertex set V to V' & V/V'
- Edges which lie on the cut (endpoints on different sides of partition) are said to "cross the cut".
- A cut respects a set P of edges if no edge in P crosses the cut.
- Light edge 'e' of a cut is the edge with least weight which "crosses the cut".
- This light edge is safe to add to A.

— x —

Proof of Correctness; (Selecting safe edge)

- If for some cut, we get light edge = {(u,v)} & adding this forms a cycle (To our set A), then there must be another edge (x,y) in our tree T (Hypothetical).

⇒ We can say there exists another tree T',

$T' = T - (x,y) + (u,v)$,

$wt(T') \leq wt(T)$, since $w(u,v) \leq w(x,y)$

— x —

ie: For any cut, if we say one edge $\in T$, then safe edge $\in T'$ & $wt(T') \leq wt(T)$

(For any cut, there must be 1 or more "edge on cut" edges since our MST "(T)" is not disconnected).

— x —

*Corollary Theorem:

- Let C & C' be two connected components in A, then if (u,v) is a light edge connecting C to C', then (u,v) is safe for A

—x—

1) Kruskals Algo:

- Our cuts are such that we isolate one component from the rest.
- Any edge added connects two components, so the edge added is also a safe edge, since we go in order of sorted weights.

—x—

* Generic MST:

```
Begin
    A = ϕ
    while A != Spanning Tree
        e = (u,v) which is safe for A for some cut
        A = A U (e)
    end
end
```

—x—

*2) Prims Algo:

Start with set of edges $A = \phi$, start with some vertex C.

⇒ **Invariant**: Graph induced by edges in A is a connected acyclic graph.

- Initially, make a cut which separates this connected component we have till now including C.
- Now the light edge which is a safe edge is added to A. Then for the next cut, we will have each connected component including this edge as well.

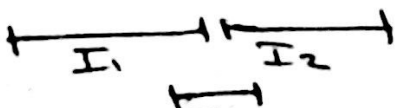
Greedy Approach; → Algorithms based on local rules.

- Each iteration requires you to make a choice (By looking at what is best for now)
- Solution is built incrementally.
- We get globally correct solution from the locally best choices.

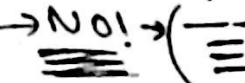
---

1) Interval Scheduling Problem; (Objective is to satisfy as many requests as possible)
(Early Finish)

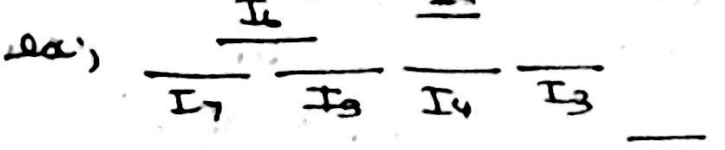
- a) Shortest Duration does not work,

ex: 

} we only get 1 interval I_3 from shortest rule.

b) Minimum overlap shortest duration?
Does it work? → NO! →  → Infinite ex

c) Early finish → Choose & remove all overlaps
$(T(n) = O(n)) \times n \log n$ (This works)

ex; 

→ I_7, I_5, I_9, I_8, I_2 (Early fish rule)
→ I_7, I_5, I_9, I_3, I_2 (Another possibility)

⇒ Proof that its correct;

- Let A be our solution & O be optimal solution.

  ∴ |A| = |O|, for our solution to be valid

- By using earliest finish times we stay ahead of the ~~other optimal~~ solution O or equal to it.

⇒ Proof by induction; → finish time

⇒ We need to show $f(a_i) \leq f(o_i)$ for every $i = 1 \pm k$ where $O = \{o_1, o_2, \dots\}$, $A = \{a_1, a_2, \dots\}$

- Base case, $i=1$, we pick minimum, so $f(a_1) \leq f(o_1)$ always.

- Now let us assume $f(a_i) \leq f(o_i)$
- We need to show $f(a_{i+1}) \leq f(o_{i+1})$

⇒ $s(a_{i+1}) >= f(a_i)$
$s(o_{i+1}) >= f(o_i)$
} In any valid interval set.

⇒ $f(a_i) <= f(o_i)$ (Hypothesis)

∴ $f(o_i) >= f(a_i)$

∴ $s(o_{i+1}) >= f(a_i)$

⇒ Now this means we can choose o_{i+1} or a_{i+1}, but we choose a_{i+1} which means $f(a_{i+1}) \leq f(o_{i+1})$

∴ Our induction is correct.

Induction proof → If we assume $|A| = k$ & $|O| = m$ where $k < m$, then for k^{th} term,

$f(a_k) \leq f(o_k)$, so that means our algorithm would not stop at k & also pick a_{k+1}, until $|A| = m$.

$\therefore |A|$ must have been $\geq k$.

---

*Note; Algorithm Proof Design;

a) Algorithm stays ahead → Interval Scheduling

*b) Exchange argument → M.S.T (Changes on hypothetical MST gives us our MST)

c) Structure based proof

---

* These changes should not compromise on objective

---

2) Scheduling Problem (Duration and Deadline approach)

(Earliest deadline first)

(Lateness = $\max(0, f(i) - d(i))$

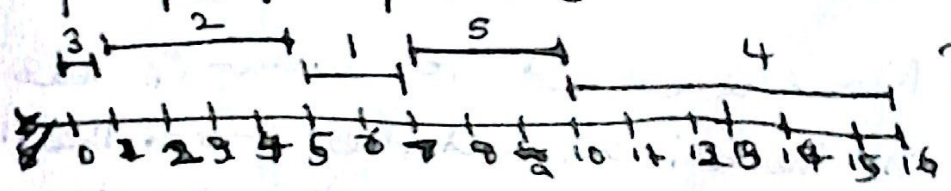
(We want maximum lateness to be minimum)

- Exchange argument approach.

---

ex; Draw a schedule for these jobs

Job	1	2	3	4	5	
t	2	4	1	6	3	→ Duration
d	5	5	3	7	6	→ Deadline



} Maximum lateness is 9

---

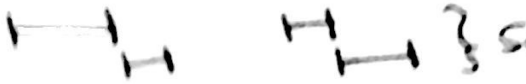
a) Shortest duration first → Nope!

✓ b) Later deadline last (or) greedy deadline first

- If jobs have identical deadlines then?

↓

It doesn't matter how we schedule them.



⇒ sort (a₁, a₂, ..., aₙ) According to deadline
print job numbers in order

•

Proof: A = Schedule generated by our algo.

O = Optimal schedule

- We will modify O to match A while not increasing max lateness of O.

⇒ Given any schedule, jobs(i,j) is an inversion if $d(i) > d(j)$ but $start(i) < start(j)$

Note: • Our algorithm produces a solution with no inversions. ↔ compare with O' to show our solution works

Lemma: All schedules that have no inversions have the same maximum lateness.

↓

This is why identical deadlines (no inversions) the same schedule is same.

- Now we need to show O has no inversions, and then from the lemma

⇒ We can say that O has no idle times (For best ie. smallest maximum lateness)

- We will now modify O to get O' such that O' has no inversions and maximum lateness of O' ≤ maximum lateness of O.

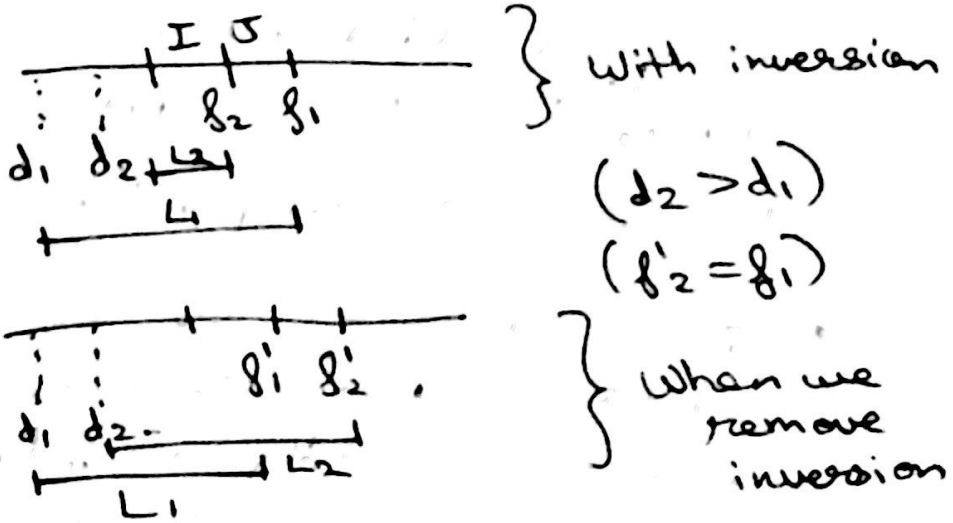
⇒ Proof to Modify; (O' → Optimal solution modified from O) (Exchange argument approach)

a) If O has an inversion, then there must be a pair of jobs i, j which are scheduled consecutively & $d_i < d_j$

⇒ (Note: If there is an inversion, then to make adjacent not under inversion, we would have to invert this with something else and so on...., so there must be 2 consecutive inverted jobs)

b) We swap i & j in O, so our result has one less inversion. (It won't create new inversion)

c) The new schedule has maximum lateness that is atmost that of O.



⇒ Our new schedule has maximum lateness ≤ max lateness of old schedule.

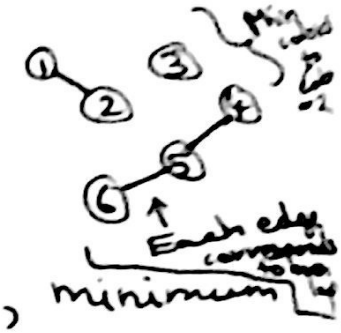
∴ We now have an optimal solution O' which has no inversions. #1

∴ From the lemma, our solution has the same maximum lateness as the optimal solution O'. #1

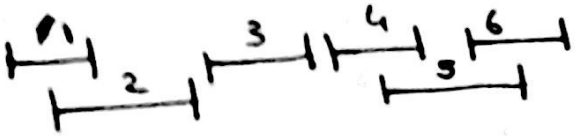
*3) Scheduling Problem-2 (Structure Based proof)

(Given start & finish times, min number of resources needed to address all the queries)

* (We can use interval graph coloring as well (Adjacent nodes of diff color)) ⇒



ex;



- Here we need 2 resources, to satisfy all queries.

⇒ Depth of interval I is the max number of intervals that overlap I, at any instant time x from its start to finish.

⇒ Depth/Answer of problem is the maximum depth of all intervals belonging to the input. (We need max depth + 1 many resources)

⇒ Resources = Max Depth + 1 //

- As we traverse our requests, we start a new resource if there are no currently available resources.

⇒ This is our greedy algorithm, we locally try to check if there are available resources or not and then assign [illegible] if needed.

Note: Clique number of G $\leq$ Chromatic number. (χ)
($\omega(G) = \chi(G)$ when G is Clique)

*Note: Morse code is not prefix free, so there can be ambiguities

---

*Note: Huffman encoding is prefix free, and also gives us the shortest possible encoding.

---

* Huffman Encoding: Greedy Algorithm

* • Any optimal encoding can be transformed to a full binary tree (All internal nodes have 2 leaves).

• $\sum f_i d_i$ should be minimum. } Cost

(f_i: Frequency, d_i: Depth (length of encoding of i^{th} character))

$\Rightarrow$ We can use exchange arguments to swap leaf nodes of our full tree to obtain optimal encoding.

• Suppose we have $f(a) \geq f(x)$ & $d(a) \geq d(x)$

$\Rightarrow$ We create a new tree (with exchange) where $d'(x) = d(a)$, $d'(a) = d(x)$

$\Rightarrow$ Decrease in cost = $f(a)d(a) + f(x)d(x) - f(a)d(x) - f(x)d(a)$

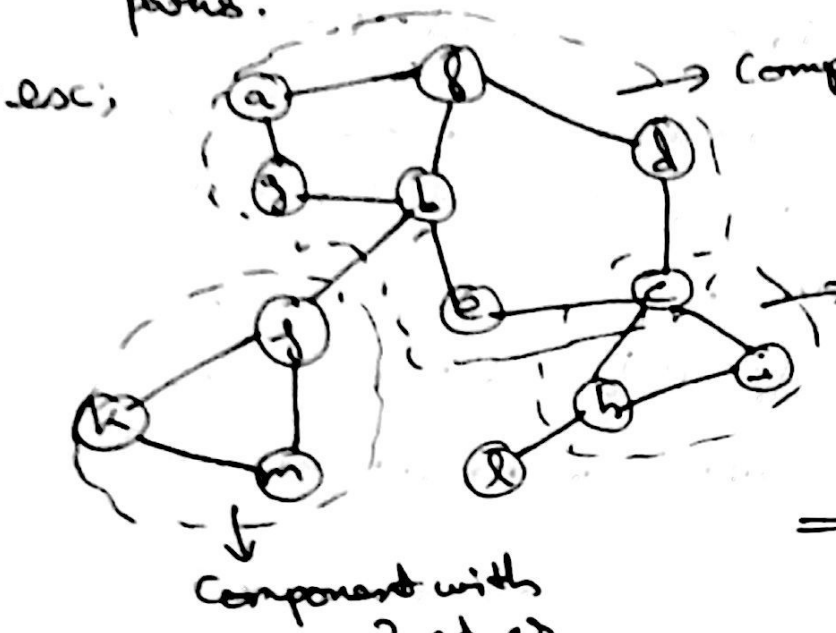
$= (f(a) - f(x))(d(a) - d(x))$

≥ 0

$\therefore$ We will see a cost decrease.

Biconnected Components
(Graphs) (videos)

* • Graph is biconnected if every pair of vertices have atleast two vertex disjoint paths.
  • Graph is 2-edge connected if every pair of vertices have atleast two edge disjoint paths.



⇒ We partition based on edges since vertices can be shared.

* * Articulation points;
  • A vertex is an articulation point if removing the vertex disconnects the graph.

* * Bridges;
  • An edge e is called a bridge if on removal of e, the graph gets disconnected.

— x —

* Note: When we perform a DFS from some node, if our selected root has more than 1 children, then this root is an articulation vertex.

- r is an articulation vertex since there can be no edges from T_1 to T_2 (else they would have merged T_1 & T_2)

---

*DFS for Articulation points / Bridges:

---

* Note:
- Ancestors/Decendants include the selected vertex as well.
- Proper Ancestor / Proper decendant exclude the selected vertex.
- Tree edges = Edges part of DFS tree
- Back edges = Edges connecting child to ~~parent~~ ancestor; not part of tree edges. (Same in undirected graphs)
- Forward edges = Edges connecting parent to child not part of tree edge (Same in undirected graphs)
- Cross edges = Edges connecting two subtrees. (Not present in undirected graphs)

---

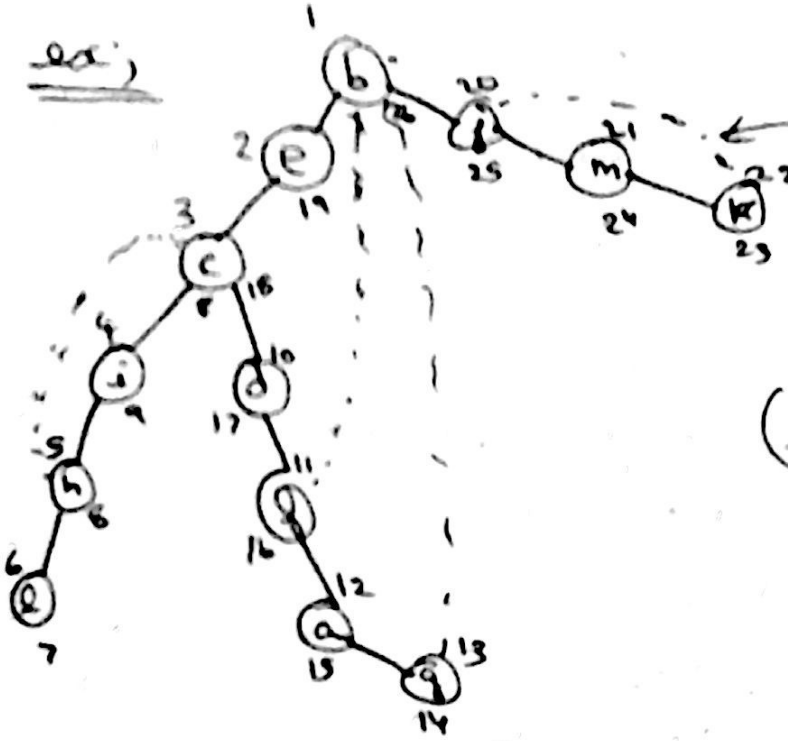
* Lemma: A non-root vertex is an articulation vertex if and only if v has a child w such that there is no back edge from w or any of its decendants to a proper ancestor of v.

⇒ We give each node a discovery time d(x).

- We define $Low(v) = \min\{d(v), d(x) \mid w_x$ is a back edge for some decendant w of v$\}$

  Low(v): Lowest back edge (Deepest back edge)

ex:



(Graph is the one we used in prev example)

	Low
a	1
b	1
c	1
d	1
e	1
g	1
h	1
i	3
f	3
d	20
k	20
l	6
m	20

⇒ A bad child v is one whose Low(v) is greater or equal to the Low(cur).

** ⇒ If a node has atleast one bad child then its an articulation vertex.

** ⇒ If a node has atleast one bad child whose Low(v) > Low(cur) then

We calculate these Low points & the articulation vertices & bridge edges in a bottom up manner. (Recursive).

Note: A bridge has atleast one endpoint as an articulation vertex.

---

* Parallel Algorithms: for these articulation points / bridges

(SiamJ. of Computing 1985, Vol 14, No 4, Pages 1-9)

* 1) Tarjan & Vishkin: (O(n+m)) → Asymptotic same runtime with a few more helper functions.

(Algo in next page)

- T is a rooted spanning tree
- Store preordering of vertices in T.
- Create a graph G' where nodes in G' are all edges in our original graph G. Two of these nodes uw & vw are connected if u is the parent of w & v-w is a non tree edge & pre(v) < pre(w)
- If T has edges uv, vw then if nontree edge from decendant to ancestor
- Once G' is created, any edges belonging to a cycle in G are a connected component in G'

⇒ So the biconnected components of G are connected components of G'.

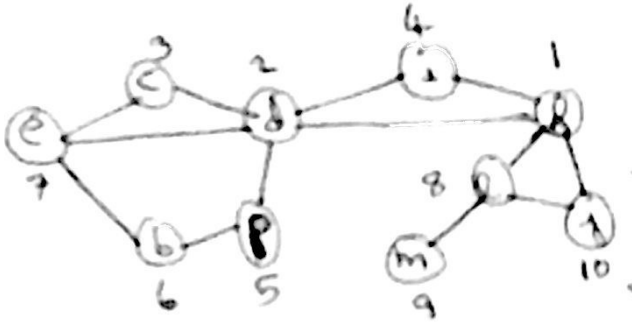
} Edge (Nontree from w to a non decendant of v) → Then we add (uv)—(vw) in G'

decendant of

→ the edge joining v & w in the tree is a bridge edge.

Tarjan & Vishkin example:

i)



→ G_1

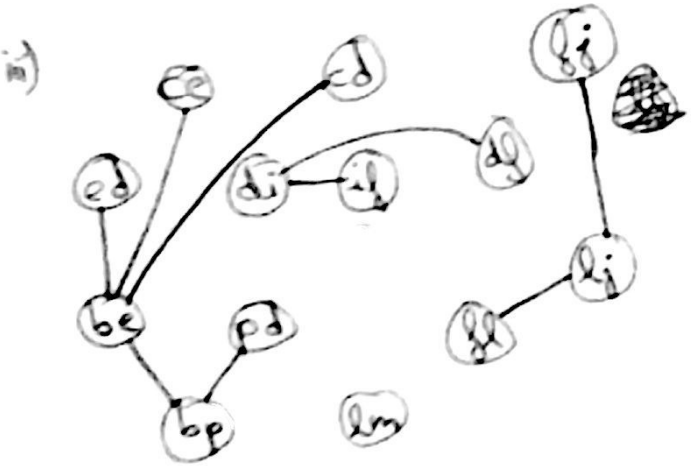
Preorder numbering for T.

ii) Take a spanning tree T & give preorder numbering

Non-tree edges:

⇒ ig
⇒ gj
⇒ ed
⇒ ce

iii)



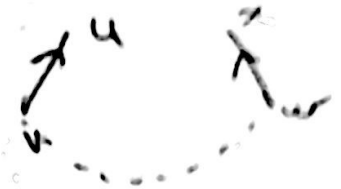
} Following our algorithm. (Since we have a small no. of tree edges, we just iterate over them & check which other nodes of G_1' are connected to this non tree node)

iv) We have 5 biconnected components in G_1.

Tarjan & Vichtin; (Algorithm)

- 3 cases to add edges to G'.

a) uw & vw are connected if u is the parent of w & $v-w$ is a non tree edge & $pre(v) < pre(w)$? → In our spanning tree.

b) 

- u is parent of v & x is parent of w in our tree T.

⇒ Tree edge $v \to u$ & $w \to x$ & non-tree edge $v \to w$, then we connect vu & wx

c) 

- $w \to v$ & is a tree edge (Parent in tree T)
- Non-tree edge joins descendant of w to non-descendant of V.

⇒ Then we connect wv & vu

Note; Regions are depicted by triangulation (Point location problem)

Note. Voronoi Diagrams & Midl → Region nearest splitting (Convex Hull)

Cuts | Flows

- Value of flow from Source to Sink in graph G is $V(f) =$ Outgoing from source $=$ Incoming in Sink.

Assume this ← Flow function of G

(Ford-Fulkerson Algo)

1) Iterative approach to get $V(f)$;

Store

- Add all reverse edges with capacity of 0.
- We randomly pick a path & assign it max flow possible.

⇒ Sometimes we will end up choosing wrong paths, so we will need some way to send back flow. (use residual graphs)

⇒ Residual graph G_f; → For flow f

- $f(e) = \text{Min}(\text{flow of all edges})$.
- Nodes, $V(G_f) = V(G)$
- Forward edges: $e = (u,v) \in G$, $f(e) < c(e)$ → capacity than G_f has an edge $e' = (u,v)$ → $c_f(e) = c(e) - f(e)$

~~If e is a backward edge, $c_f(e) = c(e) + f(e)$.~~

~~If e is a forward edge~~ $c_f((u,v)) += f(e)$

- Backward edges; $e = (u,v) \in G$, $c_f((v,u)) +=$

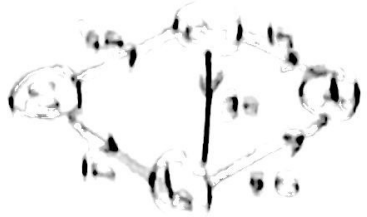
Flows on edges ← change inverse to capacities

$c_f((u,v)) -= f(e)$

$f(e) > 0$, Add (v,u) to G_f where

$c_f((v,u)) += f(e)$

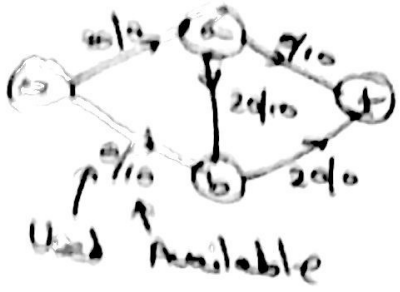
Ex: $G_1 \Rightarrow$



Capacities

- Consider path $s \to a \to b \to t$

$\Rightarrow$ We get



$\downarrow$

We get G_{f_1},

The values here are the new capacities

Residual graph.

These edges are what help us send the back flow.

Omit edges with capacity 0.

- Now consider another path,

$s \to b \to a \to t$ → Augmenting path.

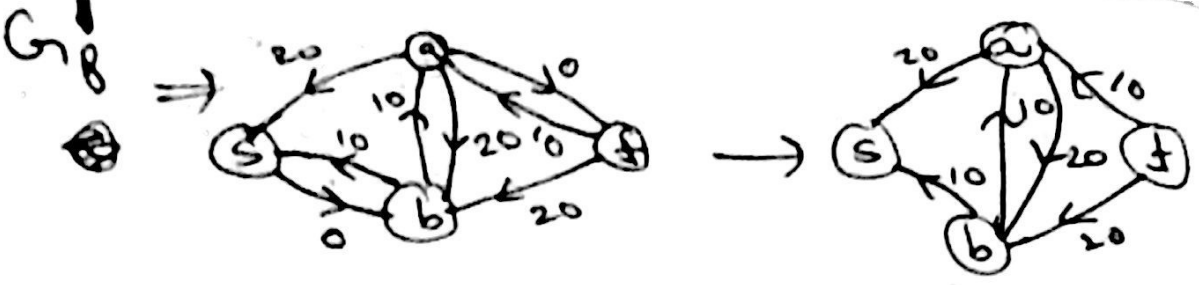
Note: We also need to store for 2 nodes u, v which edge $u \to v$ is a forward edge & which $v \to u$ is a backward edge.

In this path,

$\Rightarrow$ Here we see min of capacity on path $= 10$,

So, In our new residual graph, G_{f_2}

Gf

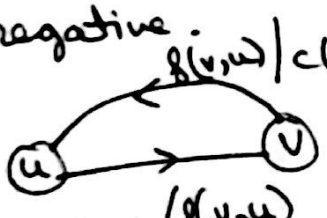


⇒ We can now stop our algorithm since there exists no other paths from ⓢ to ⓣ.

- The number of iterations is atmost max(f) (i.e. its a pseudo polynomial algo).

Proof;

i) ⇒ We can see that the flow on any edges at any time is non-negative.



} We can see that
$c_f(u,v) = f(v,u)$
$\therefore f'(v,u) = f(v,u) - x$
where $x \leq f(v,u)$

ii) We can also see that ~~capacity~~ flow never exceeds the capacity.

Note: When drawing G'_f & residual graphs, make sure you represent $f(e) / c(e)$ for each edge. (If $f(e) = 0$, then ignore edge)

---

iii) Flow conservation (we need to prove it)

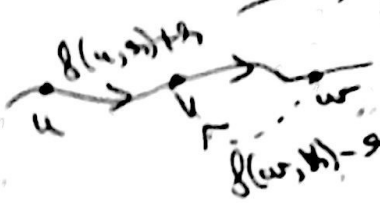
- Consider vertices v on path selected.

=> Let the two neighbours be u & w.

Case 1: If $u \to v$ & $v \to w$ are forward edges,

- η comes in & it leaves so flow is conserved.

Case 2: If $u \to v$ is forward & $v \to w$ is backward (So corresponding forward edge is $w \to v$)

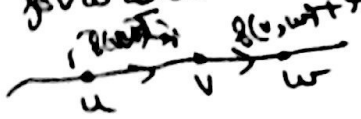


From our definition (if) (u,v) is forward $f'(u,v) = f(u,v) + \eta$ else $f'(u,v) = f(u,v) - \eta$

- So η comes in from (u,v) & η goes out via (w,v) (Back edge)

∴ Its conserved.

Case 3: If (u,v) is backward edge & (v,w) is forward edge.



- So $-\eta$ leaves from (v,u) & $+\eta$ leaves from (v,w)

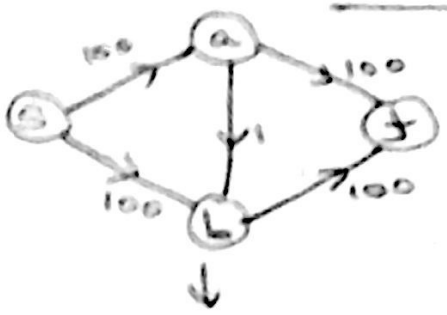
∴ Its conserved.

Case 4: Both are backward edges

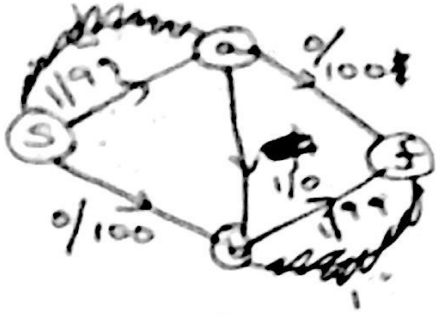
- So $-\eta$ enters from (w,v) & $-\eta$ leaves from (v,u).

Show that $f'(t) \Rightarrow f(t)$ as we iterate over f over again.

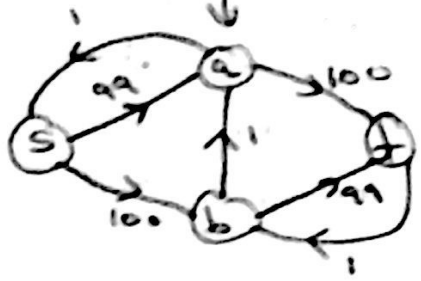
Since in each iteration we add $\min(f_e) > 0$ to $f(t)$. So $f'(t) > f(t)$. //



$\Rightarrow G_f$ (only capacities)
(Consider path s, a, b, t)

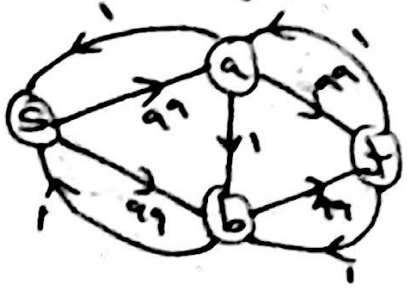


$\Rightarrow G'_f$ (Flow/Capacity)
(Don't show backward edges yet)



$\Rightarrow G_f'$ (Ignore 0 capacity edges)
(Capacities)
(This is the residual path.

$\Rightarrow G''_f$ (Flow/Capacity)
(Cap of forward + backward = c)
↑ Initial capacity



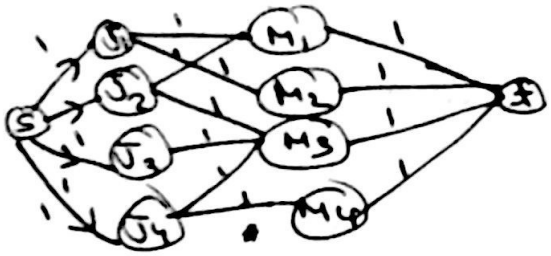
$\Rightarrow G''$ (The new residual path)

⇒ It was proved that selecting shortest path (Dijkstra) would make the algorithm effective.

———x———

* Note: Suppose there are n different jobs & m machines, where each machine is able to do one of k jobs its eligible to do. (Bipartite graph).

- We add a source node joining all jobs & a sink node joining all machines.
- Then we check if max-flow is n or not.



} Check max flow.

———x———

* Cuts:

- For a graph G, an ST cut is a partition of vertices of G into S & T where $s \in S$ & $t \in T$.

$$f(S,T) = \sum_{u \in S,\ v \in T} f(u,v) - \sum_{v \in T,\ u \in S} f(v,u)$$ } Flow from S to T — Across cut

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$ } Total capacity from S to T — Across cut.

$$\Rightarrow f(S,T) = v(f) \quad // \quad \left| \left(f(S,T) \leq c(S,T) \right) \right.$$

Capacity of Capacity of edges,
$\Rightarrow$ ~~Minimum~~ cut = ~~Flow~~ across cut (S,T)
$(C(S,T))$

- We can show mincut = maxflow for any flow/cut.

Maxflow = Mincut;

Proof: f is a maximum flow in G_f, we can show we get it from the ~~2 possible~~ minimum cuts which ~~include only source of G_f/s or only sink of G_f/t~~.

$f(s) = f(t) =$ Mincut in this case.

** Maxflow = Mincut;

- The maximum flow, in our graph, is $f(s) = f(t) =$ Mincut. (i.e. the minimum $C(S,T)$ for all cuts of graph G_f).

Note: Edmond's Karp algorithm (Out of syllabus).

Reduction Among Problems (Similarity/Usability of Algorithms)

- Using already known algorithms to analyse newer algorithms.

1) **Lower Bounds**; of algorithms which use known algorithms.

ex. Convex-Hull ↔ Sorting algorithm (excluding pseudo polynomial.)
(So we can say lower bound is nlogn for convex-hull since it has sorting) (whose lower bound is nlogn)

Reduction;

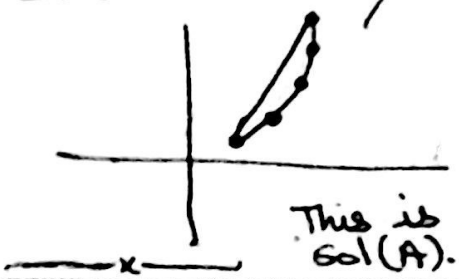
⇒ Consider problem A & problem B.

- If we have a function f which maps Input(A) $\xrightarrow{f}$ Input(B), and on solving B with f(Input(A)) if we can get the solution of A from sol(B) then we can reduce problem A to B.
- If A = Ω(g(n)), then B cannot be solved in o(g(n)).

ex; A = Sorting Problem, B = Convex Hull

- We make f such that for input $i = a_i$ in input of A.
- We make (a_i, a_i^2) as input for B (∵ f is O(N)) (Parabolic convex hull.)

⇒ Then we solve these points with convex hull → sol(B)



⇒ Now from this solution we can just read vertices in increasing x order by traversing

So we reduced A to B. ← This is sol(A).

P2) We can solve new problems via known solutions to other problems.

a)

ex: Consider matrix inversion & Gaussian elimination

- Given input A is a square matrix which has an inverse.

$\Rightarrow$ Assume $B = A^{-1}$,

$\therefore B \cdot A =$ Unit matrix (Identity).

$B \cdot A = [A \times b_1 \quad A \times b_2 \ldots A \times b_n]$ where

$B = [b_1 | b_2 | \ldots | b_n]$

↓

Column vectors.

- Let $e_i = A \times b_i$

$\Rightarrow$ The i^{th} place in e_i should be 1 & rest should be 0.

- Using gaussian elimination, we can get B.

$\therefore$ We have reduced matrix inversion to gaussian elimination.

———x———

b)

x2: Consider a clique of size k (Input) on graph & finding a set of k vertices in G which are mutually disjoint.

G, k → Input of Clique | G', k' → Input to independent set problem

⇒ Solve independent set problem as Sol(B) then get sol(A) from this.

- G_1' = Complement of G_1 & $k' = k$.

⇒ Then on solving independent set problem on G_1' & k', we get clique on G_1 & k.

—×—

Note: These are mutually reducible

—×—

- Since in the complement graph, if we select k' vertices, then it means they are not connected in G_1', which means they are connected in G_1, so its a clique of size k

⇒ ∴ We can reduce A to B (or vice versa)

—×—

c) ex 3: Vertex cover & Set cover problems

↓

- Pick smallest subset of vertices such that every edge in G_1 has atleast one endpoint in subset of vertices.

↓

- Pick smallest number of given n sets such that the union of selected sets is the set U.

⇒ Select set $U = E(G_1)$

- Now there are n sets (each vertex has a set) and each of these sets are all adjacent edges of vertex v_i.
- So we can reduce vertex cover to set cover problem. (A to B)

—×—

*d) Bin packing problem & Partition problem (
ex 4i (B)

- We reduce A to B

⇒ Bin packing problem

↓

- Given n objects each of size < 1, we need to find how many bins of size 1 are required (minimum).

↓

- Given n numbers, we need to find a partitio such that sum of elements in is equal to su of elements in

⇒ Now the function f to convert is each $x_i' = \frac{2 \times x_i}{\sum x_i}$, so that our new $\sum x_i' = 2$. Now we given the packing problem & we see if we can pack them into 2 bins. If we can then the only possibility is 1 in bin 1 & 1 in bin

∴ We have partitioned our given n numbers.

— ∝ —

Problem Classes

Note: Turing-Machine: Infinite memory (Tapes)
Multiple Heads for cores (Tape heads)

---

* 1) P = Class of problems which can be solved in polynomial time assuming we have as memory. (Easy to solve problems)

---

* There exists no tester (program) which can take an input & tell if an input program will halt or not (infinite loop).

↑

Halting Problem

---

* Non-determinism: (In programming)
* • Suppose our computer has a 'guess' instruction.
  • If there exists a right guess, then this additional instruction we assume will give us the correct guess.
  • If there exist no right guess, then this cannot give us a correct guess.

ex: Pick a perfect square b/w 1 to 100,

- Random probability to be correct $= \frac{10}{100} = 10\%$
- Non deterministic probability $= 100\%$.

Note: We can solve problems like clique, vertex cover etc with non-deterministic computers

Eg: Solve clique problem with non-determinism;

⇒ Pick vertex v_1

⇒ Pick vertex v_2 adjacent to v_1 according to non-deterministic of forming a clique.

⇒ Pick vertex v_3 which is adjacent to both v_1 & v_2.

⇒ Continue till we have found k vertices, if found return "YES" else "NO".

Note: We could also just say we guess a single non-deterministic choice of pick k vertices which form a clique.

(Let us assume we can only guess 1 number/unit of time (So we ignore this & use above method))

---

* 2) NP → Problems which can be solved by using a non-deterministic computer in polynomial time.

⇒ $P \subseteq NP$ (∵ Any P problem can be solved on a non deterministic computer in polynomial time).

⇒ $NP \subseteq P$? } No proof (Answer is not known)

Note: $^nC_k \approx \left(\frac{n}{k}\right)^k$ (Stivling's Approximation) (Large n)

After Mid 2

Problem Classes

Note: Turing - Machine: Infinite memory (Tapes)
Multiple heads for cores (Tape heads)

---

* 1) **P** = Class of problems which can be solved in polynomial time assuming we have ∞ memory. (Easy to solve problems)

---

* There exists no tester (program) which can take an input & tell if an input program will halt or not (infinite loop).

↑

Halting Problem

---

* Non-determinism (In programming)

* • Suppose our computer has a 'guess' instruction.
• If there exists a right guess, then this additional instruction we assume will give us the correct guess.
• If there exist no right guess, then this cannot give us a correct guess.

ex: Pick a perfect square b/w 1 to 100,

- Random probability to be correct $= \frac{10}{100} = 10\%$
- Non deterministic probability $= 100\%$.

Note: We can solve problems like clique, vertex cover etc with non-deterministic computers in small amounts of time.

---

* Note: Clique, Vertex cover, Set cover etc are NP complete problems.

—— x ——

Note: Identify small D.N.F which has large C.N.F.

—— x ——

* Deterministic vs Non-Deterministic:

- We can solve all Non-deterministic problems on deterministic computers by modifying the algorithm a little. (Trying all possibilities)
- But this would take a lot of time.

⇒ So all non-deterministic algorithms are compu on present computers.

* ⇒ If a non-deterministic algo runs in $p(n)$,
* then deterministic requires $k \times q^{p(n)}$ where q is number of choices each non deterministic instruction has at each point of time. (So in deterministic, we n to check all these possibilities to get a

—— x ——
—— x ——

* Halting Problem:

* ⇒ Cardinalities of infinite sets: (Cantors rule)

- For finite sets $|A| = |B|$ iff all elements of A can be put in a 1:1 map with elements of B. (we can also apply this for ∞ sets).

—— x ——

⇒ Cardinality of infinite sets;

- The above rule applies here as well.

1) ⇒ We will prove $\boxed{|\mathbb{Q}| = |\mathbb{N}|}$

- $\mathbb{Q} \Rightarrow p/q$ where $\gcd(p,q)=1$ (Rational numbers)

$\therefore |\mathbb{Q}|$ is atmost $|\mathbb{N}\times\mathbb{N}|$

- We will show $|\mathbb{N}\times\mathbb{N}| = |\mathbb{N}|$

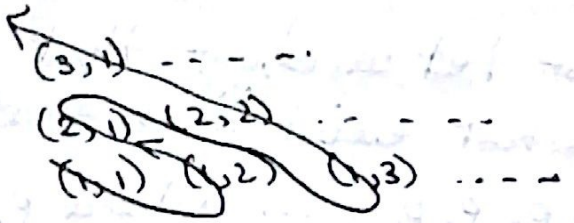
$\therefore |\mathbb{Q}|$ is atmost $|\mathbb{N}|$

Similarly we ~~know~~ know $|\mathbb{N}|$ is atmost $|\mathbb{Q}|$ Since every $n \in \mathbb{N}$ also belongs to $\mathbb{Q}$.

$\therefore |\mathbb{N}| \leq |\mathbb{Q}|$ & $|\mathbb{Q}| \leq |\mathbb{N}|$

$\therefore |\mathbb{Q}| = |\mathbb{N}|$ //

- ⇒ Now lets show $|\mathbb{N}\times\mathbb{N}| = |\mathbb{N}|$
  - Let us consider a mapping as such



N:	1	2	3	4	5	6	7	8	9
N×N:	(1,1)	(1,2)	(2,1)	(1,3)	(2,2)	(3,1)	. . . .		

- For a given (x,y) we can find the mapped element → $\frac{k(k+1)}{2} + p$ where $k = x+y-1$ (All elements whose sum is smaller), p is elements before this whose sum is $x+y$.
- For a given value x in $\mathbb{N}$, we can say, find the k such that $\frac{k(k+1)}{2} \leq x$, then sum of $(a,b) = k+1$ ⇒ $a+b = k+1$, checking which one we get after $x - \frac{k(k+1)}{2}$ iterations

⇒ ∴ We get a mapping, so $|\mathbb{N}| = |\mathbb{N}\times\mathbb{N}|$

Note: In our proof above we only considered Q+
but we can use similar proofs showing
$|\mathbb{Z}| = |\mathbb{N}|$, then $|\mathbb{Z}\times\mathbb{Z}| = |\mathbb{N}\times\mathbb{N}|$ and ex
o/o, we can show $|\mathbb{N}| = |\mathbb{Q}|$.

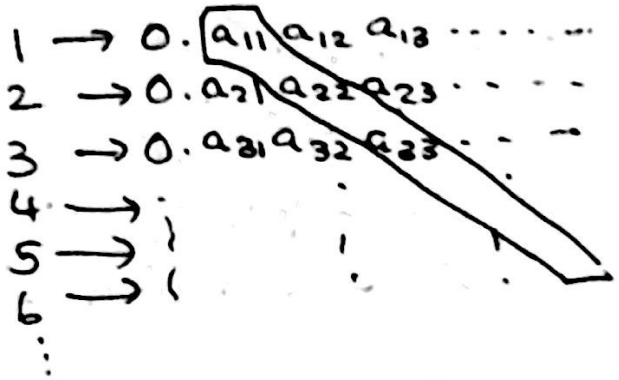
— x —

*2) We will show $\boxed{|\mathbb{R}| > |\mathbb{N}|}$

- We will show |subset (0,1)| itself is bigger than $|\mathbb{N}|$.

* ⇒ Proof by diagonalization (Contradiction)

- Assume we have a mapping, (bijection)



⇒ Now let us show that a map for Q does not exist such that,

$Q \to 0.p_1p_2p_3p_4\ldots$ where $p_1 \neq a_{11}, p_2 \neq a_{22}, p_3 \neq a_{33}, \ldots$

⇒ This does not belong to the mapping.

∴ We can see that our assumption is false, so we cannot form a bijection

∴ $|\mathbb{R}| \neq |\mathbb{N}|$ //

∴ $|\mathbb{R}| > |\mathbb{N}|$

— x —

Characteristic Function:

- A string of 0's and 1's where each element is an output for some input of a problem.

—— x ——

⇒ Lets consider a problem A, its characteristic functions that it can be either of the possibilities where there are n-inputs

2^n

ex: Inputs: 0, 1, 00, 01, 10, 11,

ex: Characteristic function: "Yes" for {0, 1, 00, 11} → 111001 } string
or
"Yes" for {01, 10, 11} etc .... → 000111 } string

Each problem corresponds to a characteristic function.

Proof: Why Non computable problems exist?

- Suppose every problem has a corresponding program, → problem's characteristic function.

Program 1 → 1 0 1 0 1 1 0 0 .....
Program 2 → 0 1 0 0 0 1 1 .....
Program 3 → 0 0 1 0 0 0 0 0 .....
⋮ ⋮

} Find a problem whose output is not listed here.

⇒ Now we can always find a characteristic function $b_1 b_2 b_3 \ldots$ where $b_1 \neq$ 1st elem of first characteristic function, $b_2 \neq 2^{nd}$ elem of second " " etc ....., so our list of every problem having a corresp. program is incomplete.

(Proof by diagonalization)

—— x ——

*Halting Problem: (It is non-computable) (Video?)

- Consider we have a program which tells us of the halting problem.
  Takes input → Outputs "Yes" or "No"
  ↑ Halts ↑ Doesn't halt

*- If we have HALT we can also have NEGHALT (Consider, (make it loop) NEGHALT loops forever if its input program halts

⇒ Now what if we give NEGHALT as input to NEGHALT?

⇒ If input NEGHALT gives "Yes", then outer NEGHALT loops forever

⇒ If input NEGHALT gives "No", then outer NEGHALT gives "Yes".

- This is meaningless since if inner NEGHALT gives "Yes", then input does not halt, so outer NEGHALT should give "Yes" as well, but we get "No".

If input gives "Yes" then it means the input program halts so outer NEGHALT gives us "No".

*- This is a contradiction since, if NEGHALT loops forever, neghalt gives "YES" (so looping) or vice versa

Note: A problem P which does not gives "YES" output for any input is called an empty problem.

—x—

1) Emptiness Problem; (Non computable problem)

- A program that tells us whether a given problem is an empty problem or not.

2) Acceptance Problem (Non computable problem)

3)

—x—

Approximation

1) Maximization Problems:
   ex: Find largest clique in a given graph

2) Minimization Problems:
   ex: Find/Minimize the number of bins to pack all given items.

—— x ——

* r- Approximate algorithms:
* • Suppose our algorithm A is an r-approx algorithm
  • Let us consider a polynomial time algorithm A giving us outputs A(I) & optimal NP algorithm gives us OPT(I).
  a) If its a maximization problem, $\frac{OPT(I)}{A(I)} \leq r$
  b) If its a minimization problem, $\frac{A(I)}{OPT(I)} \leq r$

$\Rightarrow r = \max\left(\frac{OPT(I)}{A(I)}, \frac{A(I)}{OPT(I)}\right)$

⇒ The closer r is to 1, the better our approximate algorithm A is.

—— x ——

* 1) Let's consider vertex cover problem.
  (Slader's Algorithm) ($r \leq 2$) (Minimization problem)

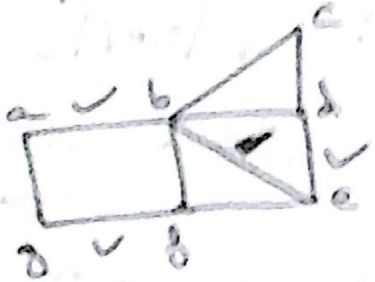
⇒ Let M be a subset of E which is a maximal matching of G.
   ⇒ No 2 edges in M share the same endpoint.
   ⇒ No superset of M exists with same conditions.

⇒ Once we have M, we need to include both endpoints of each edge in our vertex cover.

- We try to find the smallest vertex cover.

⇒ So from this algorithm we might not get the smallest vertex cover since we include both endpoints (to ensure its a proper vertex cover).

ex:



Suppose, $M = \{ab, fg, de\}$

So vertices $= \{a, b, g, f, d, e\}$

Optimal $= \{b, g, d, e\}$

⇒ We can see that our vertex cover covers all edges, ∴ Its a valid vertex cover.

———— x ————

*Proof: For a maximal matching M, the optimal vertex cover $|OPT| \geq |M|$ since we need to pick atleast one vertex from each edge in M. (Since we need to cover all edges)

⇒ $|U| = 2|M|$ where U is the vertex cover produced by our algo. (Stated)

∴ We can see that '2' here is atmost 2.

———— x ————

*2) Bin Packing Problem: (Minimization problem)

a) First fit: Check for each item, all the bins if it fits anywhere then put it in otherwise make a new bin.

Proof: Firstfit uses no more than twice the number of bins the optimal algorithm uses.

⇒ Suppose first fit uses k bins, we can see that sum of items in bin i & bin i+1 is > 1, since if it was ≤ 1, we would have put them in the same bin i.

∴ The total weight of all items is atleast $k/2$.

$\Rightarrow |OPT| > \sum_i w_i$, $|OPT| \geq k/2$

∴ $k \leq 2|OPT|$ where our algorithm uses k bins.

∴ $\gamma \leq 2$ (For this algorithm)

——— x ———

We can show that $\gamma \leq 1.8$ approx but proof is difficult

——— x ———

b) Sorted Order:

- Use first fit with items in sorted order.

Proof: Sorted first fit uses no more than 1.5 bins the optimal algo uses.

⇒ Suppose sorted first fit uses k-bins consider bin $2k/3 = t$

Case 1: Bin t has an item of size $\geq 1/2$.

- This means that all t-bins have an item of size $> 1/2$.
- So optimal solution needs t-bins $= 2k/3$ bins.

Case 2: Bin t does not have an item of size $\geq 1/2$.

- This means that items in bins $t+1$ to k have sizes $< 1/2$.
- None of these fit in first $t-1$ bins.

- Each of these bins from $t+1$ to k has atleast 2 items except possibly the last bin.
- So there are $2(k-t)+1$ items which did not fit in bins 1 to $t-1$.

Total weight exceeds $t-1$ since the $t-1$ bins overflow

If we add these small items.

This is why we chose $t = 2k/3$.

$\Rightarrow$ This means that our total weight of items exceeds $\min(2(k-t)-1, t-1)$

$\therefore$ We have $|OPT| >= \sum w_i >= 2k/3 - 1$

$\therefore k \leq 3(|OPT|+1)/2$

---

3) Load Balancing; (Minimization problem)

- There are m machines & n jobs with times t_i.
- Makespan of machine M_i is $T_i = \sum_{j \in A(i)} t_j$ where $A(i)$ = Jobs assigned to machine i.
- Makespan of an assignment $T = \max_j(T_j)$

$\Rightarrow$ We want to minimize T.

a) Greedy Algorithm

- Let T^* be the best possible makespan

$\therefore T^* \geq \left(\frac{1}{m}\right) \sum_n t_i$

- $T^* \geq \max_j t_j$ } Since all jobs must be assigned some machine.

$\Rightarrow$ Suppose we consider the last job we assigned t_j to machine M_i, and T_i is the largest now.

$\Rightarrow$ Before assigning t_j, $T_i \Rightarrow T_i - t_j$

- Since we are assigning this job to this machine it means every other machine has atleast $T_i - t_j$ load time.

$$\therefore \sum_{k=1}^{m} t_k \geq m(T_i - t_j)$$

$$\Rightarrow T_i - t_j \leq \left(\frac{1}{m}\right) \sum_{k=1}^{m} t_k \leq T^*$$

$\Rightarrow$ We also know $t_j \leq T^*$

$$\therefore T_i - t_j \leq T^*$$

$$T_i \leq T^* + t_j$$

$$T_i \leq 2T^*$$

$\therefore \alpha = 2$ here

---

b) Sorted greedy assignment.

- If there are less than m jobs, our algo is same as the optimal one.

$\Rightarrow$ Consider t_{m+1} in the sorted order of jobs (Decending order).

$\Rightarrow$ So t_{m+1} is given to a machine which already has a job $\geq t_{m+1}$

$$\therefore T^* \geq 2t_{m+1} \rightarrow (1)$$

$\Rightarrow$ Now consider the previous proof: if t_j is the last job assigned to M_i then $t_j \leq t_{m+1}$ (Sorted order) (As ... atleast ...)

$$\Rightarrow t_{m+1} \leq \frac{T^*}{2} \text{ from } (1)$$

$\therefore t_j \leq \frac{T^*}{2}$

$\therefore T_i \leq T^* + t_j$

$T_i \leq \frac{3T^*}{2}$

$\therefore \eta = \frac{3}{2}$ here

———x———