24/08/2021

Algn Design by Kleinberg & Tardos (KT), Pearson - TextBook

– Assessment :-

• 60% HW, 20% (10% x2) Sep 25 & Nov 6 ← Midterms, 20% Final Exam, 10% Extra credit

• Weekly HW due on Monday (midnight)

• Submit a PDF file on canvas / code on GitHub.

26/08/2021

HW - Write pseudo code!

– Slides !!!

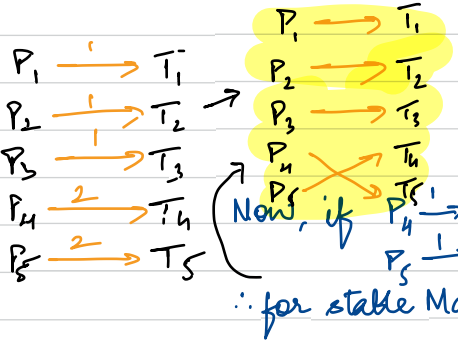
↳ How to solve a prblm

– Matching Pblm

• Given n TAs → m courses

Q. • Stable matching - no better outcomes

– Practise Pblm 1

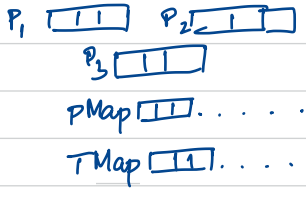


Discussion :-

• Scoring?

• Max Score!

• Gale-Shapley Matching - Book!!!



Prblm: eg,

– Top 10 most freq. bigram in search keyword.

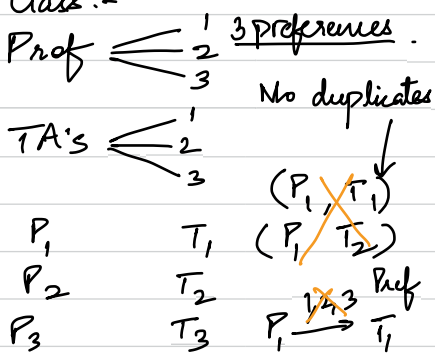
Eg. slow keyword restraunt

↳ Treated as a single bi-gram!!!

– Ask clarificat^n quest^n !!!

- What counts as sol^n?, where data comes from etc

Prblm Class :-

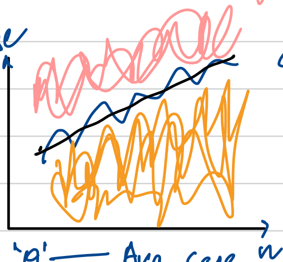


worst case

$f(n) \in \Omega(g(n))$

$\left[\begin{array}{l} O \\ \Omega \text{ best case} \end{array} \right.$

$f(n) \geq g(n)$



$g(n) \geq f(n)$

$O(g(n))$, $f(n) \in O(g(n))$

when 'O' & 'Ω' meets then 'θ' — Avg. case.

insertⁿ sort no 'θ'

- Best case & Worst case doesn't depend upon # of elements, but type of data.

5 6 7

3 -

$5 - 4 - 3 - 2 + 1 + 0$ '5'

$1 + 1 + 1 + 1 + 1 + 0$ $f(n) = n$

$f2 = (3 + 3 + 3) + (3 + 3 + 3) + ($

$3 \quad 3 \times 3 (3 + 3 + 3)$ $n^2 \times n^2 \rightarrow n^4$

$n \times n [n]$ (n^3)

Maths :-

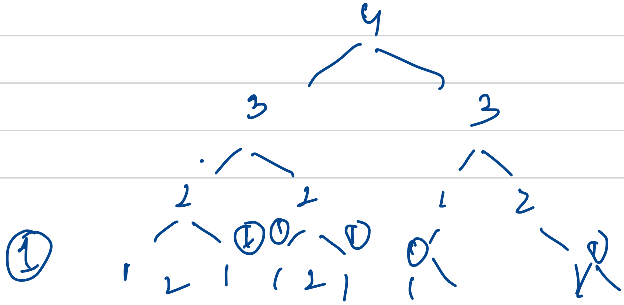
- $\Theta(n^2)$ | $\theta(n^2)$ | $T\_f(n) = 2^* T\_f'(n) + T\_f'(n/2)$

'O' & 'Ω' comes into picture if the pgm doesn't run for 'n' times. else 'θ'.

4

2

6



Optimum sol^n - Just 1 sol^n
Optimal sol^n - Many diff sol^n

31/08/2021 'KT 4.1'

Greedy Algorithm

Pseudo code for Earliest finish code:- Discuss time complexity

$a_i = \langle s_i, f_i \rangle$
- Sort a_i for ascending f_i → Depends which algo we use
  - ↳ quicksort $\theta(n \log n)$
  - ↳ heapsort $\theta(n \log n)$
  - ↳ countsort $(\theta(n)$
- Each a_j → $\theta(n)$
  - if compatible add to 'A'
    - ↳ f_i with the last job's finish time

→ Counting sort :-
1) All items come in range $(0, m-1)$, fairly small
2) n items, each item is $(0 \ldots m-1)$ k exchangeable
3) sort of 'n' items in the range of $(0, m-1)$ is $\theta(m+n)$
   ↳ if $f(n) = n+1$, $f(n) \in \theta(n)$
   but $f(n) = m+n \not\Rightarrow f(n) \notin \theta(n)$
   eg. $f(n) = n+n^2$ Necessarily
   if $m = n^2$ then; $f(n) \in \theta(n^2)$

L eg. of stephen curry v Jordan, dunk each 100$, but yet diff.

- How do we known Greedy algorithm is optimal?
  - Greedy algorithm is a decent algorithm & is always not to far from optimal sol^n.

CHECK SLIDE FOR HW

Induct^{n}

$A = \{P_1, P_2, \ldots, P_k\}$

$O = \{P_1, P_2, \ldots, P_m\}$

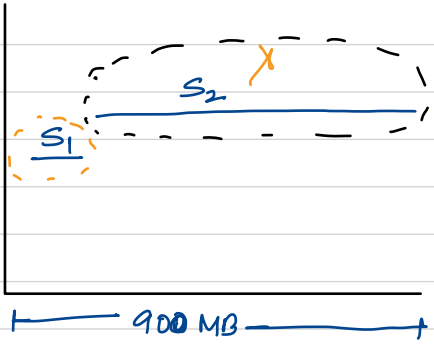
for optimal $m = k$.

for $m > k$, P_{m+1} that's in 'R'

$S_1 + S_2 + \ldots + S_m \leq$ 'D'

for 'S_{k+1}' not to be selected, $S_{k+1} > D - \sum_{i=1}^{m} S_i$

but since S_{k+1} is present in 'O' thus it is not possible not to select it.



S_1 : 10 MB

S_2 : 900 MB

M - 900MB.

Thus ↓ chooses the best

$900 > D - 10$ $\boxed{S_2 > D - 900}$

890

W

NY

Boston

$\Sigma w_i = N \times W$

$\Sigma w_i = 6 = 2(N)$

[1, 1, 1, 1, 1, 1]

[1] → ⊗

$T_A = \{i_1, i_2, \ldots, i_k\}$

$T_O = \{j_1, j_2, \ldots, j_m\}$

$m < k$

$w = [1]$, w $\quad A = [i_1, i_2, \ldots, i_k] \Leftrightarrow O = [j_1, j_2, \ldots, j_m]$

$i_{r-1} \le j_{r-1}$ $\quad i \& j$ # of trucks $\quad r \ge 1 \rightarrow i_r \le j_r$

$r \le k \Rightarrow w(i_r) \le w(j_r) \Rightarrow w$

for $r=1$, this is true as for 1 item atleast 1 truck would be sent

Let $r > 1$, assume $n(i_{r-1}) \le n(j_{r-1})$ i.e its true for 'r-1' & prove for 'r'

for accomodating next weight in same truck

$w(j_r) \le W - w(j_{r-1})$ $\quad$ GA $b_1, b_2, \ldots, b_j$ $\quad$ k trucks

OS $b_1, b_2, \ldots, b_i$ $\quad i \le j$

$i = 1$ $\quad j = 2$ $\quad$ for $k=1$, fit as many boxes as possible

Assmpt - $k-1$:- j' boxes k OS: $i' \le j'$

k^{th} :- OS packs $b_{i'+1}, \ldots, b_j$

$j' > i'$ ← GA $b_{i'+1} \ldots b_j$

Example i.e val max

S - not max pgms

O -

$f'(i) = f(j)$

$l_i = f_i - d_i$ $\quad l' > l_j > l'_j$

$d_i > d_j$

$l'_i = f_j - d_i$

$f_j - d_i < f_j - d_j$ (l'_j)

(l'_j)

- Watch Dykstra's !!!

$d_j < d_i$

$f(j) > f(i)$

$f(j) = f'(i)$ $\quad L_i = f'(i) - d_i$ $\quad L'_i > L_i$

$L_i = f(i) - d_i$ $\quad L'_i = f(j) - d_i$

$d_i > d_j$

$-d_i < -d_j$

$f(j) - d_i < f(j) - d_j$

$L'_i < L_j$ → proves ↓ in the max^m latency

OPT $\quad$ S

$\Rightarrow j_k \le i_k \quad k = 1 \quad j_0 \le i_0$ – farthest pt. $\quad \therefore j_{k+1}$

$j_k \le i_k \quad$ 'i_{k+1}' $\leftarrow i_k \quad$ If $j_k \le i_k \quad$ 'i_{k+1}'

$\downarrow \therefore$

$j_{k+1} \le i_{k+1}$

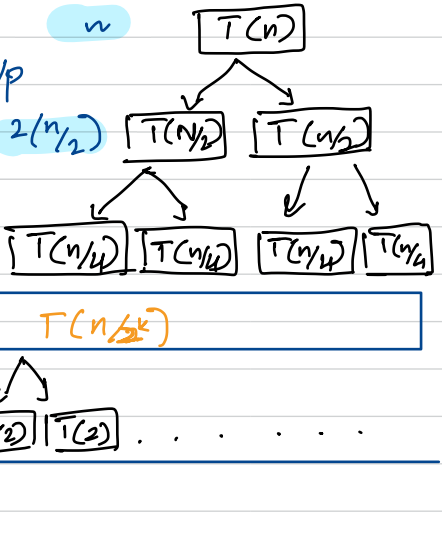
DIVIDE & CONQUER

07/09/21

- $T(n)$ = # of comparison to mergesort an i/p

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \end{cases}$$

for $n = 5$: $\lceil n/2 \rceil \hookrightarrow 3$, $\lfloor n/2 \rfloor \hookrightarrow 2$



Sol^n : $T(n) = O(n \log_2 n)$

Proof :-

Hypothesis

$T(n) = n \log_2 n$

To show :-

$T(2n) = 2n \log_2 2n$

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

$= 2[n \log_2 n + 2n]$

$= 2n[\log_2 n + 1] = 2n[\log_2 n + \log_2 2]$

$= 2n \log_2 2n$ — Hence proved!!!

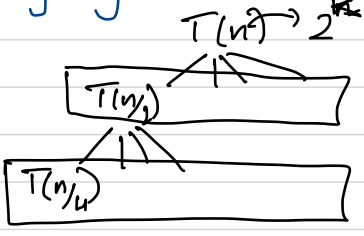
Matrix Solitare –

9/9/21

$k = \log n$ $\quad 2^k = n \Rightarrow k = \log n$ → Given

$4^k = 2^{2\log n} = (2^{\log n})^2 = n^2$ — (1)

Each layer gives – n^2 & there are k layers, ∴ $O(n^2 \log n)$



$T(n/2) \cdot 4$ $\quad 2n$ $\quad \frac{n}{2} \cdot 4$ $\quad \frac{n}{2} \times n$

$T(n/4) \cdot 16$ $\quad K = \log n$ $\quad \frac{n}{4} \cdot 16$ $\quad 4n$

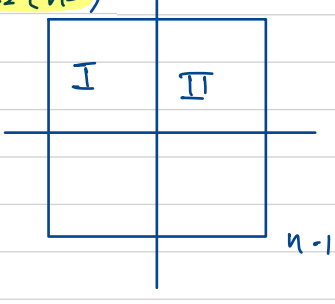
$n = 2^k$

$\therefore \log n = k$

$4^k = n^2$ (from 1)

Best case :- $\quad T(n)$ | C' $\quad \Omega(n^2)$

$$T(n) = \begin{cases} c, & n = 1 \\ \text{recursive} \begin{cases} c' \text{ (best case)} \\ 4 \cdot T(n/2) + f(n) \end{cases} \end{cases}$$



m, B[0:m-1][0:m-1]

m, B[0:m-1][m:]

m, B[m-1:n-1][0:m-1]

m, B[m-1:n-1][m-1:n-1]

Insert" sort :-

eg 5 4 3 2 1

```
while ( round n-1 & swap)
{
   for every pair (i, i+1)
       if ( A[i] > A[i+1]
           swap
}
```

4 5 3 2 1
4 3 5 2 1
4 3 2 5 1
4 3 2 1 5

sort on 'n-1'

$T(n) = (n-1) \cdot c$

- case 1 'swap' $T(n-1)$
- case 2 'no swap' c'
  - case 1, n=1 (base)

Ω – $T(n)$ - - - - 'n-1' <just once executed>

c'

O – $T(n)$ - - - - - 'n-1' comparisons. $T(n-1) \cdot c$

$T(n-1)$ 'n-2'

$T(1)$

n.

E.g. Proof by Induct"-

$T(n) \leq T(n/5) + T(7/10\, n) + 3n$ $\qquad T(n) \leq cn\log n$

$\leq c\frac{n}{5}\log\frac{n}{5} + c\frac{7n}{10}\log\left(\frac{7n}{10}\right) + 3n$

$= \frac{cn}{5}(\log n - \log 5) + \frac{7cn}{10}(\log 7n - \log 10) + 3n$

$= n\log n\left(\frac{c}{5} + \frac{7c}{10}\right) + \left(-\frac{c}{5}\log 5 + \frac{7}{10}c\log\frac{7}{10} + 3\right)n$

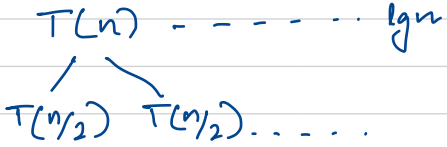
$= \frac{9}{10}c \cdot n\log n + \left(\quad (1) \quad\right)n$

$\leq$ 'cnlogn' $\quad (1) \leq \log n$ – Assumpt" $\Rightarrow$ ['catch']

↳ for very large 'n'

HW03

$T(n) = 2T(n/2) + \lg n$ $\quad T(n) \in \Theta(n)$ $\quad cn \le T(n) \le cn$



$T(1) = c \le cn$ ✓

P: $T(k) \le ck$

$T(k) \le 2T(k/2) + \lg k$

$\le 2 \cdot ck/2 + \lg k$ $\quad$ let $c = 1$

$\le cn + \lg k \le ck$

$cn + \lg n \ge ck$

$4T(n/4) + n$

$T(1) - 1 \lg 1 - 0 \le n \lg n$

P.T

$T(n) \le cn \log n$

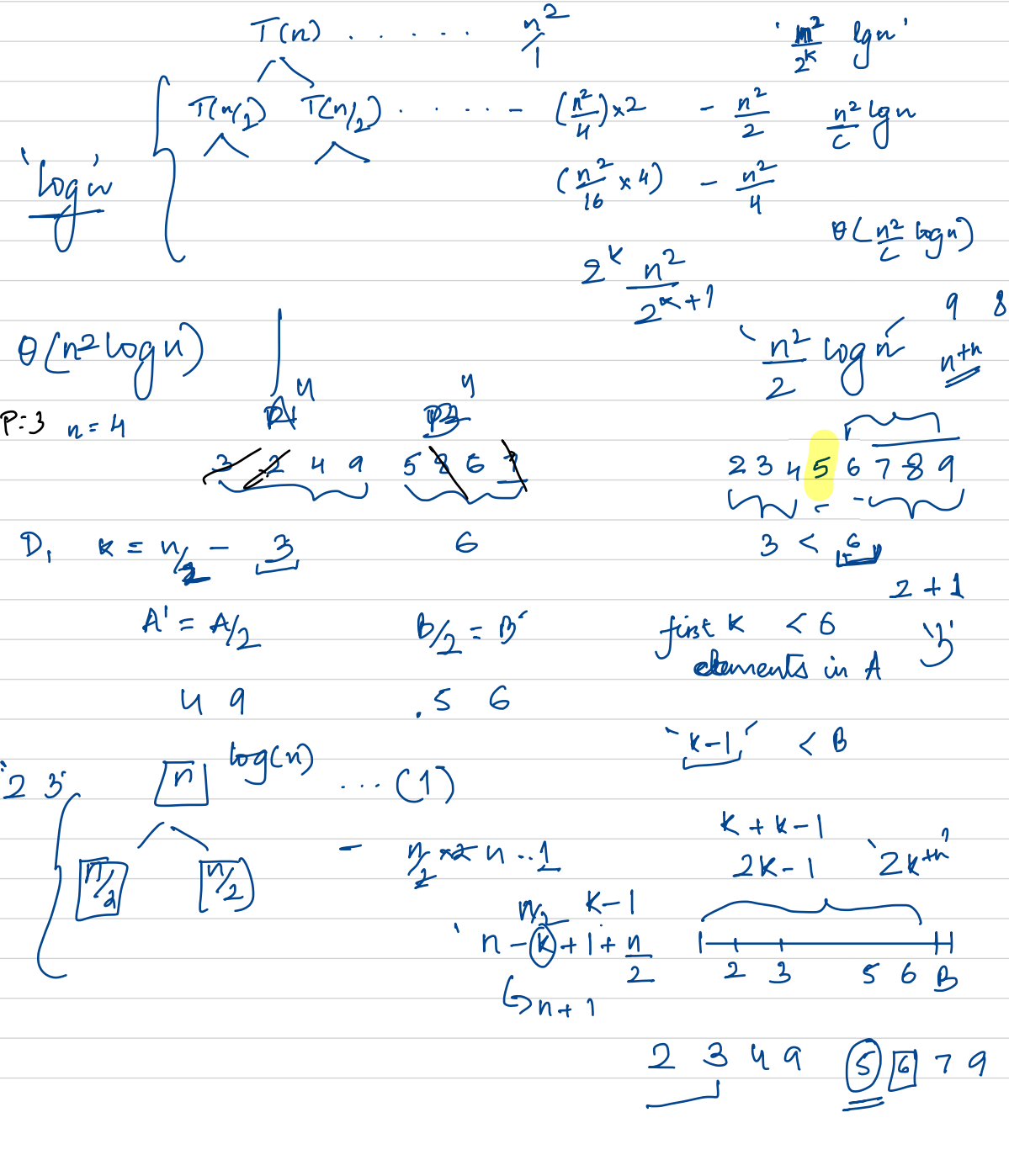
$T(n) \le 4T(n/4) + n \le 4 \frac{cn}{4} \log n/4 + n$

$\le cn[\log n - 2] + n$

$\le cn \log n - n[2c - 1] \le cn \log n$

$2c - 1 \le 0 \Rightarrow c \le 1/2$

$c \ge 1/2$

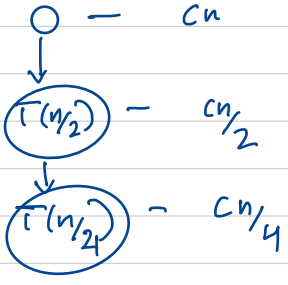

T(n) n²/1
T(n/2) T(n/2) . . . . - (n²/4)x2 - n²/2
(n²/16 x4) - n²/4
'log n'
'n² lgn'
n²/2^k lgn
n²/c lgn
θ(n² logn)
2^k n²/2^(k+1)
θ(n² logn)
P:3 n=4
3 2 4 9
5 8 6 7
D, k = n/2 - 3
6
A' = A/2
B/2 = B'
4 9
5 6
'n² logn
2
n+n
9 8
2 3 4 5 6 7 8 9
3 < 6
2+1
first k < 6 elements in A
'k-1' < B
'2 3'
n
log(n) ... (1)
n/2
n/2
n/2 x 4 .. 1
n - k + 1 + n/2
n/2 k-1
↳ n+1
k + k-1
2k-1
'2k^th'
1 2 3 5 6 B
2 3 4 9 5 6 7 9

Text Book 5.1 & 5.2

Any General – $T(n) \leq qT(n/2) + cn$

- Solving for $q = 1$

→ Creating recursion :-



$$T(n) \leq \sum_{j=0}^{\log n - 1} \frac{cn}{2^j}$$

$$\leq cn \sum_{j=0}^{\log n - 1} \left(\frac{1}{2^j}\right)$$

$T(n) \leq 2cn \Rightarrow T(n) = O(n)$

$\frac{cn}{2^j}$ → for any j^{th} level

$$S_n = \frac{a(r^n - 1)}{(1 - r)} \quad r = 1/2$$

$$= 1 \cdot \frac{\left[\left(\frac{1}{2}\right)^{\log_2 n - 1} - 1\right]}{1/2}$$

$$= \left(2^{-1}\right)^{\log_2 n - 1} - 1$$

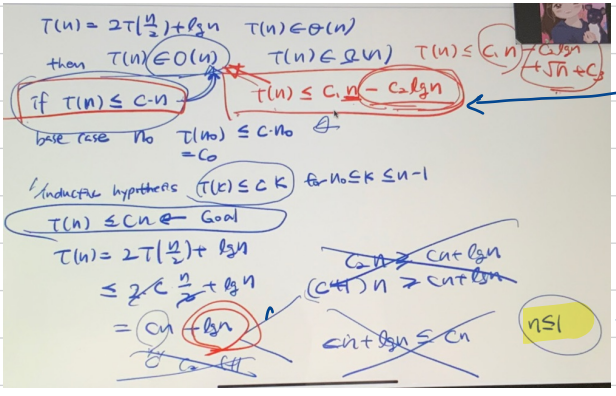
$$= 2^{1 - \log_2 n} - 1$$

$$= 2 \cdot 2^{-\log_2 n} - 1$$

$$= (2 \cdot -1)2$$

Classnotes

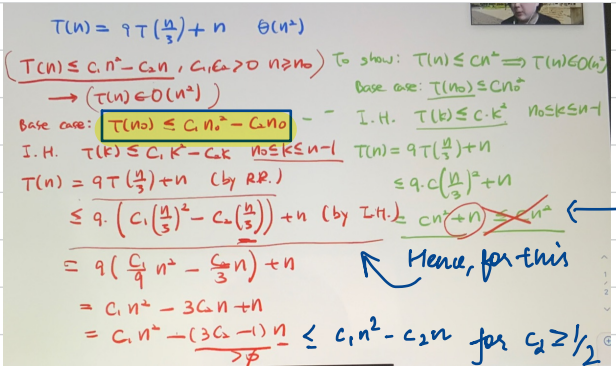
11/09/2021



Rather focus on proving this

Can't put condit^n on 'n'

Example for above case



can't be solved as the value of 'n' can't be threshold!!!

$T(m) \leq 2T(m/2) + \log m$

$\leq 2\left[\frac{c_1 m}{2} + c_2(\log m - 1)\right] + \log m$

$= c_1 m + 2c_2(\log m - 1) + \log m$

$= c_1 m + 2c_2 \log m - 2c_2 + \log m \leq c_1 m + c_2 \log m$

$c_2 \log m - 2c_2 + \log m \leq 0$

$c_2(\log m - 2) \leq -\log m$

$c_2 \leq -\left(\frac{\log m}{\log m - 2}\right)$

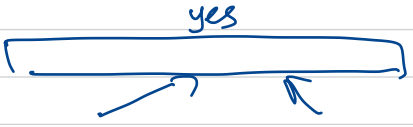
$\left\{c_2 \leq \frac{\log m^{-1}}{\log m - 2}\right\}$

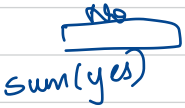
$\leq$

$c_1 m + \log m\left[2c_2 - \frac{1}{\log m} + 1\right]$

$2c_2 - \frac{1}{\log m} + 1 \leq c_2$

$c_2 \leq \frac{1 - \log m}{\log m}$





3 4 2 9 1 7 6 5 - 'n'

yes ↑ no

[3 4 2 9] [1 7 6 5] - $n/2 \times 2 - n$

yes / no

(3,4) (2,9) (1,7) (6,5) - $n/4 \times 4 - n$

yes:1 No:1 yes:1 No:1

2

yes : 1

no :

leftR (A[0:4]) < yes / no

RightR (A[4:8]) < yes / no

if len(A) == 2 ↻ c

equivalence(A[0:2])

else: leftR (A[0:2]) < yes / no

RightR (A[2:4])

if (L == no && R == no) return no

else [count(yes) return yes]

yes, count

R

0

yes

2, ␣ | 2, ␣

Merge Sort Algorithm

```
def mergesort(arr[], l, r):
    if r > l :
        middlem = l + (r-1)/2

            mergesort(arr, l, m)   # 1st half   } O(logn)
            mergesort(arr, m+1, r)
             merge(arr, l, m, r)  - O(n)
```

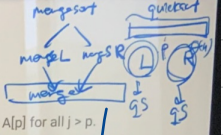
Classnotes

14/09/2021

→ Sorting algorithm using Divide & Conquer.

Quicksort

- Problem: Given n numbers (or strings), sort them.
- Algorithm: Divide-and-conquer again!
  - Input to recursive method: (A[], L, R) // To sort A[L .. R-1].
  - Pick a pivot (A[p]).
  - Rearrange the items so we have A[i] < A[p] for all i < p and A[j] > A[p] for all j > p.
  - Recursively sort A[1..p-1] and A[p+1..n].
- This is the famous (deterministic) Quicksort algorithm.
- What is its (worst-case) running time?



Actual sort happens later in Merge sort as compared to Quick sort

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

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

Quicksort - Worst-case running time

- $T(n) = T(n_1) + T(n_2) + O(n)$ where $n_1 + n_2 = n-1$
  - If we are lucky, we get $n_1 \sim= n_2$, which leads to O(n log n).
  - If we are unlucky, we get $n_1 = 1$ or $n_2 = 1$, which leads to $O(n^2)$.
- Indeed, the worst-case running time of Quicksort is $O(n^2)$.

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

$O(n \log n)$

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

$T(n-1) = T(n-2) + c(n-1)$

$T(2) = T(1) + c \cdot 2$

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

- Choose pivot a random unlikely to give worst-case scenario
- Another way is to find median & use it as a pivot.

- Given n (distinct) integers, find the k-th smallest integer.
  O(n log n) algorithm is easy -- sort and search! Can we do better? (Yes!)
- How?
  a. Divide n integers into (n/5) groups of 5 integers.
  b. Find the median in each group (call these g1, g2, ..., gx) where x = n/5.
  c. Recursively find the median of medians -- call it M (and say M is the j-th smallest number).
  d. Then 3 * (n/10) <= j <= 7 * (n/10). (Why? This is really critical!)
  e. Cases: k = j, k < j, k > j. We can recursively find k'-th smallest integer in 7*(n/10) elements.
  f. Running time Recurrence: T(n) <= T(n/5) + T(7n/10) + O(n). (Recall Slide 4 of L09?)

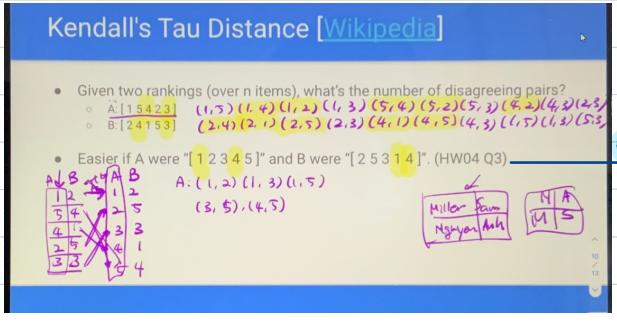
n is a multiple of 5 $\quad n/5 = n'$

$2 * \frac{n'}{2} + 2$ (n' is odd)

$3 * \frac{n'-1}{2} + 2$ (n' is even)

Classnotes 16/09/2021

- Kendall's Tau Distance :

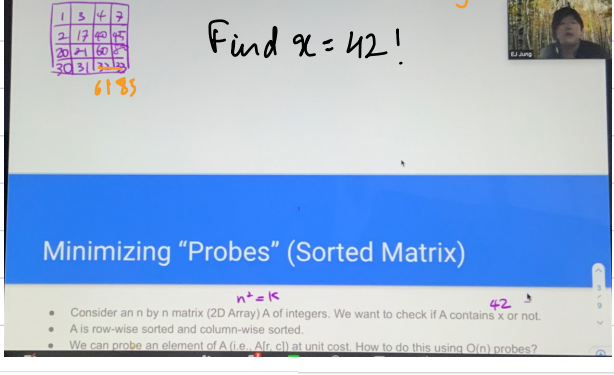


Def^n of disagreeing pair -
Say A = [1, 5, 4, 2, 3] &
B = [2 4 1 5 3], since (4,2) ∈ A
& (2,4) ∈ B
thus (2,4) is a disagreeing pair

Hint: Its all about locat^n!

- Problem :-

Binary Search Searching Matrix Pblm



BS (A[i] row, x) ← BS
BS (A[i] col, x)

if A[i][j] < x ← if this is true
=> A[i][<j] < x
=> A[<i][<j] < x
=> A[<i][<j] < x ← can conclude this

↳ MatrixSearch(top right)
MatrixSearch (bottom left)

if (A[i][j] > x)
=> A[i][>j] > x
=> A[>i][j] > x
=> A[>i][>j] > x

$T(n) = \log n + 2T(\frac{n}{2})$

Classnotes

→ Dynamic Programming –

Memoization (NOT Memorization)

```
1.  int computeMaxWeight(int j)
2.    if j == 0: return 0
3.    return max(computeMaxWeight(j - 1),
4.               computeMaxWeight(p[j]) + w[j])
```

```
1.  int computeMaxWeight(int j)
2.    if j == 0: return 0
3.    if opt[j] > 0: return opt[j]
4.    return opt[j] = max(computeMaxWeight(j - 1),
5.                        computeMaxWeight(p[j]) + w[j])
```

- makes the problem space smaller keeping the problem same.
- If subproblem has a repeated nature then we can see if we have repeated subroutine then apply dynamic problem.

• Problem statement :– Weighted Interval Scheduling

Weighted Interval Scheduling (KT 6.1)

- Recall the Interval Scheduling Problem from KT 4.1
  Now, each interval has a positive weight ($w_i > 0$), and we want to maximize the sum of weights (as opposed the number of intervals) of a compatible set.

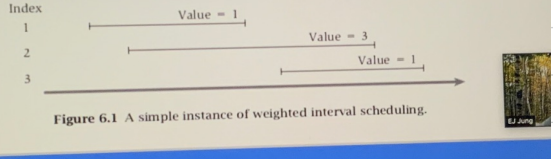


Figure 6.1 A simple instance of weighted interval scheduling.

HW04

P3 :- Kendall's Tau dist. counter example

→ consider A = [4, 2, 1, 3] (1 2 3 4)
B = [2, 4, 1, 3]

As per algm C = [[4,2], [2,4], [1,1], [3,3]]

'C' after sorting :-

C = [[1,2], [2,4], [3,1], [4,3]]

Now as per this algm Inversion pairs are (2,1) & (4,1), (4,3) but actual inv. pairs is '(2,4)' only.

P4 :-

[4,2,1,3] (4,2) (4,1) (4,3) (2,1) ① (2,4) (3,1) '2'

[2,4,3,1] (4,1) (4,3) (2,1) (3,1) '0' actually '2'.

P1 :- Counting Inversion 1, 686, 2020 0 2 3 'WTF - subhe'

2020, 686, 1, 3, 2, 0

4 2 1 3 5

(4,2) 686, 2020 1 0 1 2 3 0 686 > 2

0 0 1 2020, 686, 1 3, 2 0 5 1

LR [124] 686, 2020 0 1

[24] [5,3] 2020, 686 1 3, 2 0 ⑩

RR - [1] 3+3+4

2020 686 0, 2, 3

1, 686, 2020 1+1+1

0, 1, 2, 3, 686, 2020

1+

0 2 3, 686, 2020

For lidx < len(larr) ridx < len(rarr)

if arL[lidx] > arR[ridx]:

arr[k] = arR[ridx]

k+=1 count++

R++

else:

arr[k] = arL[lidx]

L++, k++

1+1+1

1+3+3

7+6 = 13

Classnotes 20/09/21

→ Dynamic Pblm :-

Eg. Wt. Interval Scheduling Pblm :-

of weights (as opposed the number of intervals) of a compatible set.

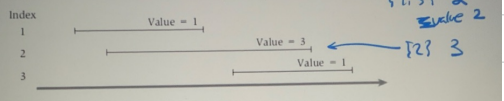


Figure 6.1 A simple instance of weighted interval scheduling.

Recursive Algorithm (KT 6.1) - Preliminaries

- We first sort the intervals by finishing time ($f_1 <= f_2 <= ... <= f_n$).

Recursive Algorithm (KT 6.1) - Preliminaries

- We first sort the intervals by finishing time ($f_1 <= f_2 <= ... <= f_n$).
- We'll say that a request i comes ***before*** a request j if i < j (hence $f_i <= f_j$).
- For convenience, define p(j) for an interval j to be the largest index i < j such that intervals i and j are disjoint. (i is the rightmost interval that ends before j begins.) p(j) = 0 if no request i < j is disjoint from j.

wrong in book !!

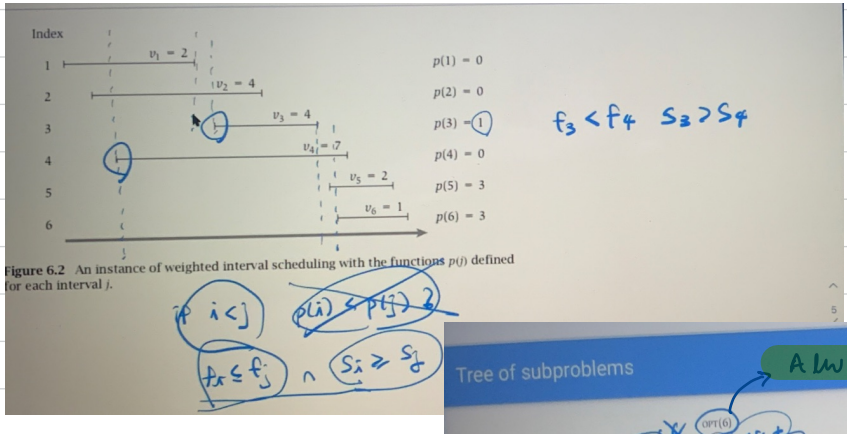


Figure 6.2 An instance of weighted interval scheduling with the functions p(j) defined for each interval j.

$f_3 < f_4$ $s_3 > s_4$

→ Considering all sets of possible sol^n :-

Equat^n for Job - not include

```
1.  int computeMaxWeight(int j)
2.    if j == 0: return 0
3.    return max(computeMaxWeight(j - 1),
4.               computeMaxWeight(p[j]) + w[j])
```

remaining Jobs ↑ include J

Tree of subproblems

Always start from last Job

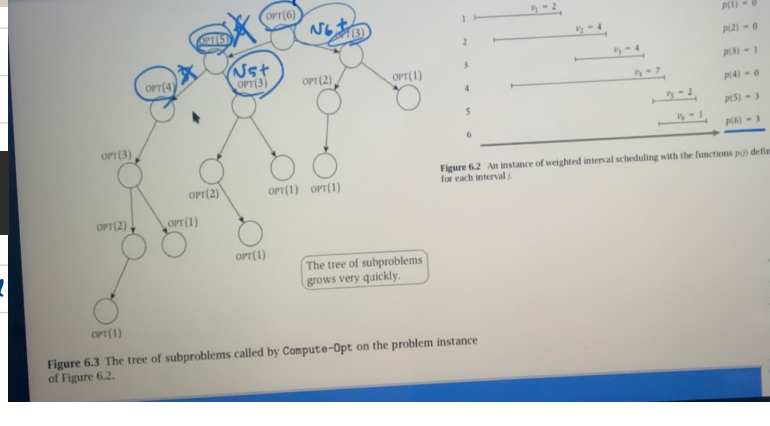


Figure 6.3 The tree of subproblems called by Compute-Opt on the problem instance of Figure 6.2.

```
1.  int computeMaxWeight(int j)
2.    if j == 0: return 0
3.    return max(computeMaxWeight(j - 1),
4.               computeMaxWeight(p[j]) + w[j])
```

Without Memoizat^n

Thus optimal sol^n {1,3,5} 8

Tree of subproblems :-

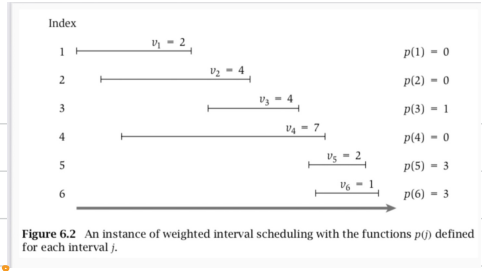
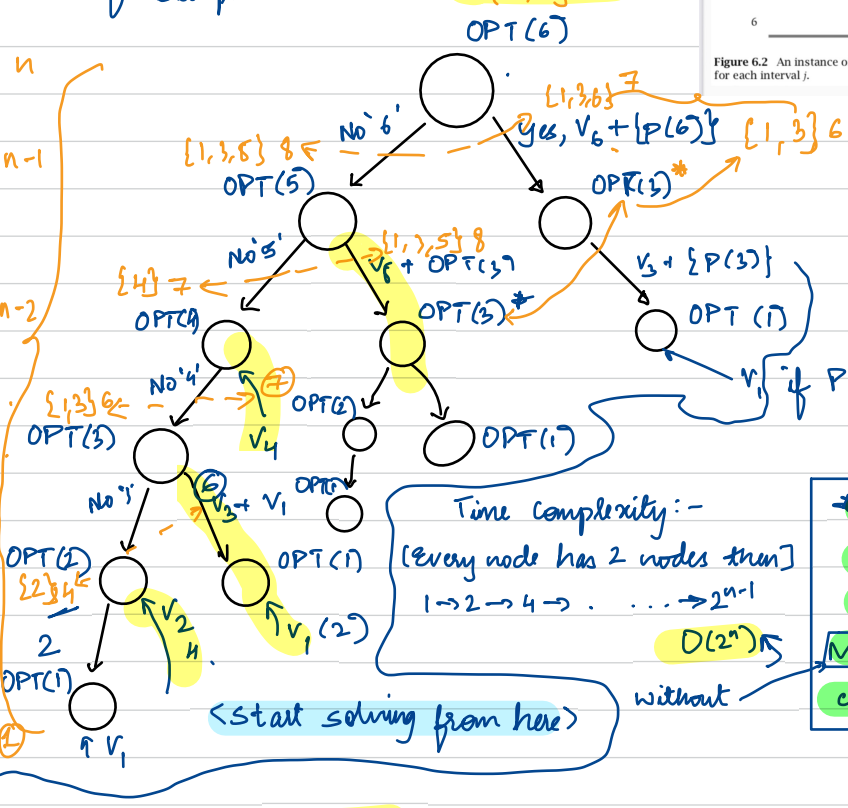
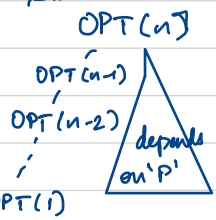


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.



General Tree



Time complexity :-

[Every node has 2 nodes then]

$1 \to 2 \to 4 \to \ldots \to 2^{n-1}$

$O(2^n)$

without

* OPT(3) since we already know the value of OPT(3) [via Memoizat^n], thus no computat^n.

<Start solving from here>

with Memoizat^n ($O(n)$)

```
1.  int computeMaxWeight(int j)
2.    if j == 0: return 0
3.    if opt[j] > 0: return opt[j]
4.    return opt[j] = max(computeMaxWeight(j - 1),
5.                        computeMaxWeight(p[j]) + w[j])
```

(just initialize 'opt' with -1 values)

without Memoizat^n ($O(2^n)$)

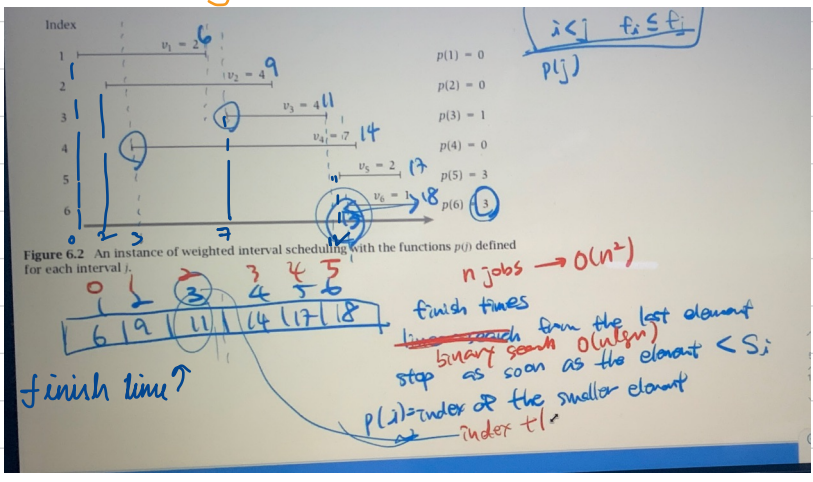
```
1.  int computeMaxWeight(int j)
2.    if j == 0: return 0
3.    return max(computeMaxWeight(j - 1),
4.               computeMaxWeight(p[j]) + w[j])
```

'opt' just contains values, whereas below code will give us optimal sol^n.

```
1.  void findSolution(Set<?> sol, int j)
2.    if j == 0: return
3.    if opt[j] == opt[j-1]: findSolution(sol, j-1) and return
4.    sol.add(j)
5.    findSolution(sol, p[j])
```

93:19

Now computing P(j) :-



finding P(j) can also be modified to run in O(n). Let's suppose start time for all jobs is sorted as well.

S = | 0 | 2 | 3 | 7 | 12 | 13 |

Thus compare F(last ele) with '13', till '11', then return p(6) = 3 & move to '12' [Si] again p(5) = 3 & move to '7', 7 < 9, move to 6 [fi] . . .

- sorting fi's & si's take O(n log n)
- finding P(i) takes O(n)
- computing takes O(n),

Thus, together gives O(n log n)

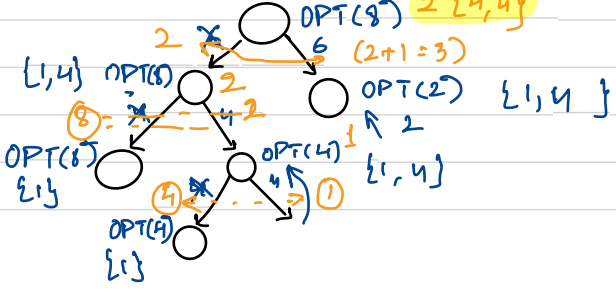
Pblm 2: finding change with the fewest coins possible

1) Greedy fails in this case :-

E.g. - {1, 4, 6} - Available denominat^n, obtain for 8¢

- Greedy - {6, 1, 1} but optimal sol^n {4, 4}

2) Using DP :-

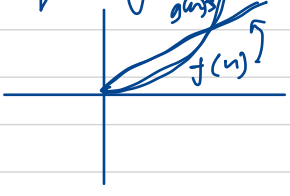


Time complexity Practise :-

1) $f(n) = O(g(n)) \Rightarrow g(n) = O(f(n))$

→ if $f(n) = O(g(n))$ then for some constant 'c', $f(n) \le c \cdot g(n)$

Now for $g(n) = n^2$ & $f(n) = n$, $n^2 \le c \cdot n$, true for some large value of 'c'. but for check $g(n) = O(f(n))$ ↑ large value of 'n', this is not true.



2) $f(n) + g(n) = \Theta(\min\{f(n), g(n)\})$

→ For $f(n) = n$ & $g(n) = 1$

$n + 1 \neq \Theta(1)$, thus disproved

But, $f(n) \ge \min\{f(n), g(n)\}$

$g(n) \ge \min\{f(n), g(n)\}$

$\Rightarrow f(n) + g(n) \ge 2 \min\{f(n), g(n)\}$

('c')

$\therefore f(n) + g(n) = \Omega(\min\{f(n), g(n)\})$

3) $\max(f(n), g(n)) = \Theta(f(n) + g(n))$

→ To prove this we want, $\max(f(n), g(n)) = \begin{cases} \Omega(f(n) + g(n)) \\ O(f(n) + g(n)) \end{cases}$

Now, $\max\{f(n), g(n)\} \ge g(n)$ or

\+ $\max\{f(n), g(n)\} \ge f(n)$

$2\max\{f(n), g(n)\} \ge f(n) + g(n) \Rightarrow \max\{f(n), g(n)\} \ge \frac{1}{2}(f(n) + g(n))$

$\therefore \max\{f(n), g(n)\} = \Omega(f(n) + g(n))$ - ①

Part 2 :-

$f(n) \leq f(n) + g(n)$; $\max\{f(n), g(n)\}$
$g(n) \leq f(n) + g(n)$

$\max\{f(n), g(n)\} \leq 1 \cdot [f(n) + g(n)]$

$\therefore \max\{f(n), g(n)\} = O(f(n) + g(n))$

4. $f(n) = O(g(n)) \Rightarrow \lg(f(n)) = O(\lg(g(n))$, $\lg(g(n)) \geq 1$ & $f(n) \geq 1$
↳ True

→ Now, if $f(n) = O(g(n))$
$f(n) \leq c \cdot g(n)$
$\log f(n) \leq \log(c) + \log(g(n)) \leq 2 \log(g(n))$
$\Rightarrow \log f(n) \leq 2 \cdot \log(g(n))$

$\therefore \log f(n) = O(\log g(n))$, Thus true.

5. $f(n) = O(g(n)) \Rightarrow 2^{f(n)} = O(2^{g(n)})$
→ $f(n) \leq c \cdot g(n)$; $2^{c \cdot g(n)} \leq K \cdot 2^{g(n)}$ → this is not always true
hence false!!!

6. $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$ - True
→ $\therefore f(n) \leq c \cdot g(n) \Rightarrow g(n) \geq \frac{1}{2} f(n)$

$\Rightarrow g(n) = \Omega f(n)$

7. $f(n) = \Theta(f(n/2))$
$f(n) \leq c \cdot f(n/2)$ Now for $f(n) = 2^n$, $2^n \leq c \cdot 2^{\frac{n}{2}}$ ⇒ Not always, ∴ false
Thus, false!!!

Read BS !

BINARY SEARCH :-

- Works on sorted list only !!!
- Algorithm :-

```
int BinarySearch(L, val)
{ if (len(L) >= 1)        -> idx
  { midVal = L[len(L)/2]

    if val == midVal :
        return L.indexof(midVal)
    else if val > midVal:
        return BS(L(idx+1 : ), val)
    else if val < midVal:
        return BS(L( : idx), val) }
  else {
        return -1 }
}
```

[1, 1, 4, 4]

4, 5

5

[1, 1, 4, 4]

W = 10 ∪ 2×10 - W/2 ∪ {W + t}

t = W/2

X' = X ∪ {

Classnotes

21/09/21

Dynamic Programming -

Pblm : -

- Segmented Least Squares

1. KT 6.3 Segmented Least Squares

- The Least Squares Problem
  - Given n points, we can "fit" a straight line that minimizes the sum of the squared differences.

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

(dep point − line)²

$p_i = (x_i, y_i)$

$ax_i + b = y_i$

y-coord

  - We can find (a, b) using a closed-form solution (often used in regression analysis; see this).

- Amt. of work = $\Theta(n)$

2. KT 6.3 Segmented Least Squares

Formulating the Problem As in the discussion above, we are given a set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, with $x_1 < x_2 < \cdots < x_n$. We will use p_i to denote the point (x_i, y_i). We must first partition P into some number of segments. Each *segment* is a subset of P that represents a contiguous set of x-coordinates; that is, it is a subset of the form $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for some indices $i \leq j$. Then, for each segment S in our partition of P, we compute the line minimizing the error with respect to the points in S, according to the formulas above.

The *penalty* of a partition is defined to be a sum of the following terms.

(i) The number of segments into which we partition P, times a fixed, given multiplier $C > 0$.

(ii) For each segment, the error value of the optimal line through that segment.

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

5 / 13

- Pt (i) makes sure minimum # of lines

Start from say P_7 & check for

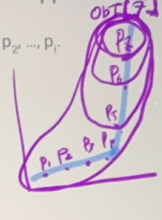
Solving This : -

OPT[i] = min(OPT[j-1] + C + E(j, i))

- Recall how we solved the WIS problem, and let's try to mimic the approach.
  - Subproblems:
    Let OPT[i] be the optimal solution for (the subproblem with) the points $p_1, p_2, \ldots, p_i$.
  - Recurrences:

$p_1 \cdots p_{i-1}$ p_i

OPT[i] :=
- OPT[i-1] + C + E(i-1, i) ← One segment to fit $\{p_i\}$
- OPT[i-2] + C + E(i-2, i) ← One segment to fit $\{p_{i-1}, p_i\}$
- ...
- OPT[1] + C + E(2, i) ← One segment to fit $\{p_2, \ldots, p_{i-1}, p_i\}$
- OPT[0] + C + E(1, i) ← One segment to fit $\{p_1, \ldots, p_{i-1}, p_i\}$

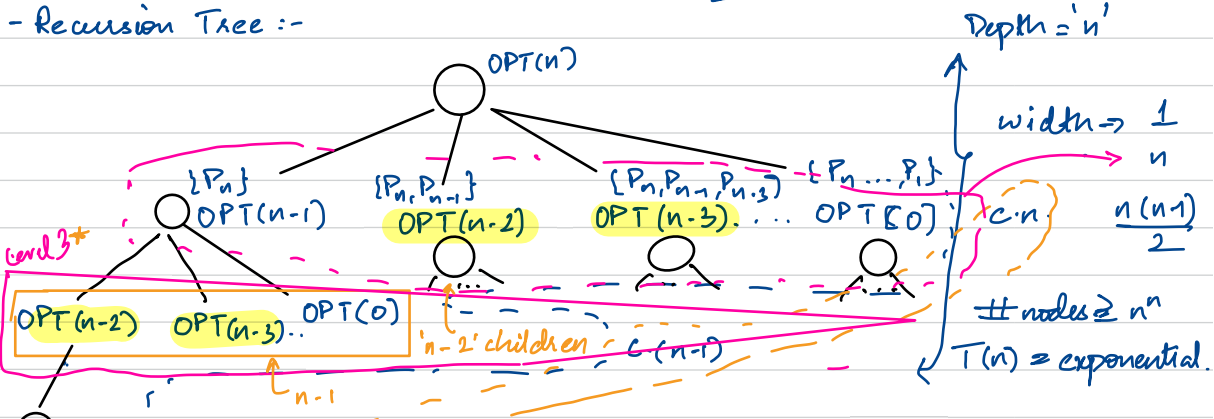


width of level 3 :-

$(n-1) + (n-2) + (n-3) + \ldots + 1$ → $(n-1) + n \cdot 2 + \ldots + 1$

↳ $\frac{n(n-1)}{2}$

- Recursion Tree :-



OPT(n-3)

OPT(0) c·1

Getting OPT(n-3) & adding $c + E(i,j)$ happens in 'O(n)' ⇐

- Benefit of Memoizat^n, no need to evaluate OPT(n-2) i.e.

↳ $O(n^2)$ <PTO for comp with wt-schedule>

```
minChange (int x)
{
    if x == c1 ... cn
        return 1;
    min = A[x - c1] + 1
    if A[x - c1] < 0
        minChange (x - c1)
}
```

KT 6.3 Segmented Least Squares

- Running Time Analysis
  - The number of subproblems: O(n)
  - Time to solve each subproblem: It depends on calculating "E(j, i)" in the recurrence.
    - If we calculate each "E(j, i)" in O(n) time, then the overall running time is O(n) * O(n²) = O(n³)
    - If we pre-calculate E(j, i) for all i, j (so we can look it up in O(1) time), then the overall running time is O(n²) See page 266 (KT Ch 6.3) for details of this optimization.
- Auxiliary Memory Space: O(n) for OPT (for the first approach above). For the second, if we store O(n²) values for all of E(j, i) in advance, we need O(n²) auxiliary memory space. (Can we do better?)

Proof by Induct^n

How to do DPI-

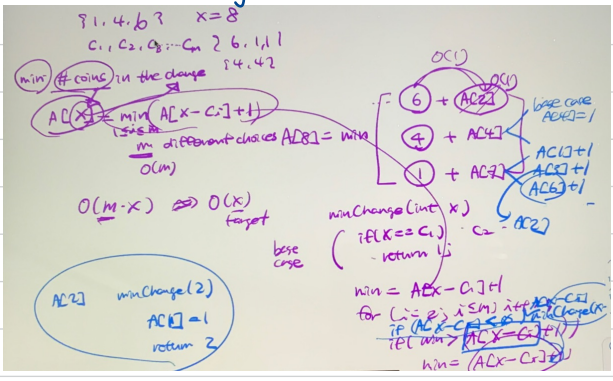
Step1: Write recursive algorithm

Step2: Goal in array/matrix (eg - min( ),

Step3: Re-write recursive Algorithm]

- More concisely: OPT[i] = Min(OPT[j-1] + C + E(j, i)) for all 1 <= j <= i.
- Proof of correctness (by induction):
  - (Base) When i = 1: The only solution is to use one segment to fit one point. Trivially correct.
  - For any i > 1:
    - Consider the last segment (of an optimal solution) that must fit a contiguous set of points $p_j, p_{j+1}, \ldots, p_i$ for some j <= i.
    - If we knew what this "j" is, then we can simply remove this segment and solve the smaller subproblem on $p_1, p_2, \ldots, p_{j-1}$ (whose solution is, by inductive hypothesis, captured by OPT[j-1]).
    - Since our recurrence finds the best way to choose the segment (containing p_i) by enumerating all valid choices for "j", OPT[i] agrees with the optimal solution.

- Min. coin change Pblm :-



[Note:
Time complexity for wt Int. schedule was O(n) as each level took constant amt of work x height, whereas in least sqs each level req. c·n amt of work for L1 then c·(n-1) for L2 . . . . c·(1) that makes it O(n²)]

Classnotes <Subset sum & knapsack Pblm>

'KT 6.4'

- Subset sum Pblm

- There are **n** items with integer non-negative, integer weights w[i].
- You have a bag which can withhold up to W units of weight.
- You want to put items into your bag (without exceeding the weight limit, W) while maximizing the sum of the weights.

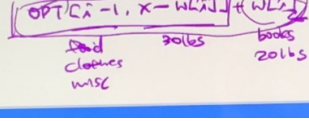
Travelling & fitting things within 50 lbs.

50 lbs bag: books 20 lbs, clothes 20 lbs, food 10 lbs, misc 5 lbs.

→ Greedy Algth counter Examples :-

- Put heavier {5} < {2,2,2} = (6), w = [2,1,2,5], W = 6
- Put lighter {3,2} < {2,3} = (5), w = [2,2,3], W = 5

- To describe a subproblem, simply stating "using items 1 through i" is no longer sufficient; we need to incorporate the weight limit as well.
  - OPT[i, x]: Optimal solution **using items 1, ..., i** and **with bound** (x). (weight)
    (Notice that OPT[n, W] is the main problem we're trying to solve)
  - OPT[i, x] := max(OPT[i-1, x], OPT[i-1, x - wi] + wi)



[2, 4, 8]

[1, 1, 4, 8]

[1, 1, 4, 4]

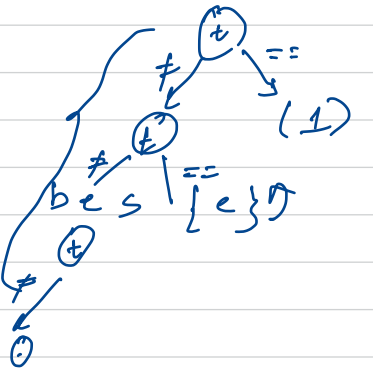
14/2 7

7

5

'4'

1



$n \times m$

$S \leftrightarrow t \qquad t \rightarrow s$ & else $LCS(x', y') + 1$

$\max(LCS(x', y), LCS(x, y'))$ (1)

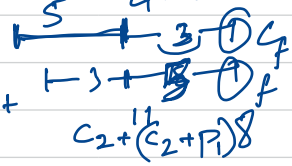
$M[i,j]$

if $M[m-1, n-1]$ {
  return $M[m-1, n-1]$;
} else {
  $M[m-1, n-1] = LCS(x', y')$
  return $M[m-1, n-1]$

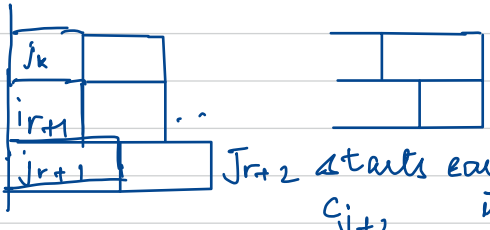
$\begin{cases} S\ c_1 + (c_1 + p_2) \\ c_1 - c_2 + p_1 \end{cases}$

$c_1 = 5$

$c_2 = 8$



$= p_1 +$



J_{r+2} starts earlier in S' than c_{j+2}

let $S_i = \{i_1, i_2, \ldots, i_m\}$

$O = \{j_1, j_2, \ldots, j_m\}$

$c'_k < c_{r+1} \qquad c'_{r+1} < c_{r+1}$

$S_{avg} > O_{avg}$

for max val of 'r'

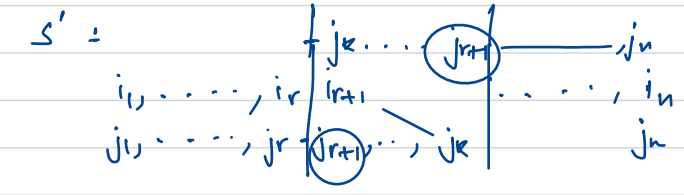
$a_j - p_j \quad c_j$

$p_{i_1} = p_{j_1} \ldots \ldots \quad p_{i_r} = p_{j_r}$

$p_{j_k} < p_{j_{r+1}}$

$c'_{r+1} = c'_k$

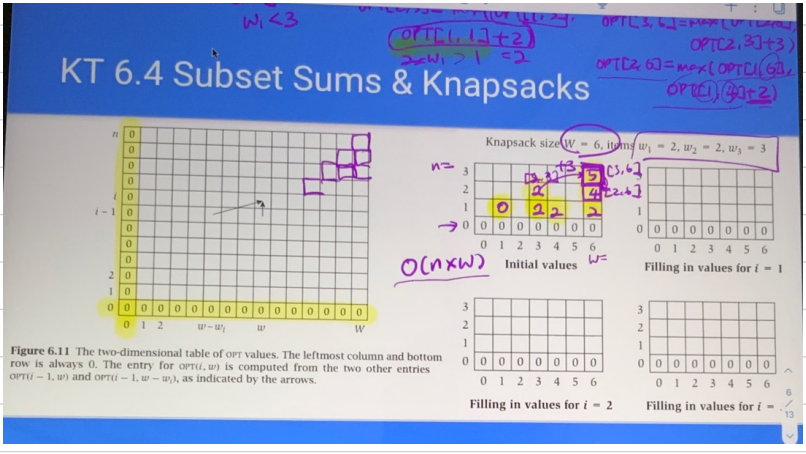
$S' =$



$S + p_{j_k} < S + p_{j_{r+1}} \quad c_{r+1} < c'_{r+1} \qquad c_{r+1}$

$c_k > c_{r+1}$

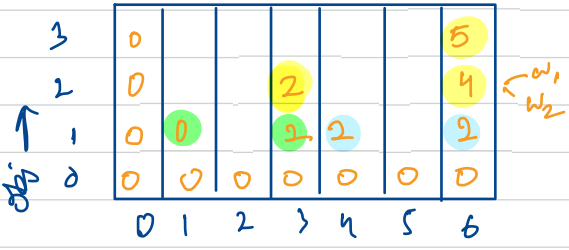
'0' items / '0' wt. limit, no items will be there



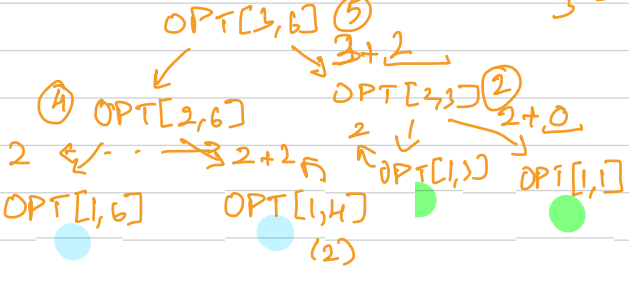
Though the upper bound is O(n x W), but we actually put very less items.
O(n x w), when all weights are 1

$OPT[i,x] := \max(OPT[i-1,W],\ OPT[i-1, W-w[i]] + w[i])$

Knapsack $W = 6$; $w_1 = 2$, $w_2 = 2$, $w_3 = 3$



Initial;
$OPT[3,6] = \max($
$OPT[2,6],$
$OPT[2,3]+3)$
$\uparrow w_3 = 3$



Classnote

30/09/21

- Proof by Induct^n

- Proof of Correctness (Induction, 2D):
  - (Base) When i = 1: OPT[i, x] = w[i] only if w[i] ≤ x
  - When i > 1:
    - If x < w[i], it's clear that we can't include item i, so OPT[i, x] = OPT[i-1, x].
    - Else: The optimal solution either excludes i or includes i.

otherwise 0

$w[i] \le x$ → max [include 'i' : $OPT[i-1, x-w[i]] + w[i]$; exclude 'i' : $OPT[i-1, x]$]

I.H: $OPT[j, y]$ contains max wt. possible from $i \le j \le i-1$, $y \le x$

$OPT[i, x] = OPT[i-1, x-w[i]] + w[i]$

→ K.T Ch 6.6 Sequence Alignment (2-D DP)

- We are given two strings X and Y ($X = x_1x_2...x_m$ and $Y = y_1y_2...y_n$).
- We want to measure similarity between two strings (which can be used to check for typos, auto-completion & suggestions, etc.). More motivating examples can be found in the textbook and here.
- We can try to "align" the given strings, but we may have a "gap" or "mismatch" between the two.

o-currance
occurrence

o-curr-ance
occurre-nce

1.

- We are given two strings X and Y ($X = x_1x_2...x_m$ and $Y = y_1y_2...y_n$).
- Consider the sets {1, 2, . . . , m} and {1, 2, . . . , n} as representing the different positions in the strings X and Y, and consider a *matching* of these sets.
- Recall that a matching is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching M of these two sets is an alignment if there are no "crossing" pairs: if (i, j), (i', j') ∈ M and i < i', then j < j'.
- Intuitively, an alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another. (page 280 of KT Ch 6.6)

2.

- Given any valid alignment M, we define the penalty as follows:
  - Penalty for Gap: For each position of X or Y that is not matched in M, we incur a cost of δ.
  - Penalty for Mismatch: For each pair of letters p, q in our alphabet (symbols), there is a mismatch cost of α(p,q) for lining up p with q. Thus, for each (i, j) ∈ M, we pay the appropriate mismatch cost α(xi, yj) for lining up x_i with y_j. For simplicity, we'll assume that α(p, p) = 0 for each letter p.

$\alpha(a,e) < \alpha(a,p)$

3.

eg. CAST [T, T] CAS / CA — α — case1

CAT [T, -] CAS / CAT — case2

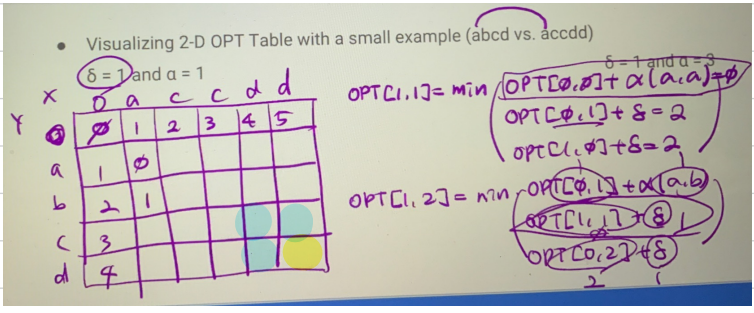
outcome [-, T] CAST / CA — δ+ — case3

{ C A S T / | | | | / C A − T } — δ+ (cost)

- Consider an optimal alignment M:
  - Case 1: x_m is matched with y_n (i.e., (m, n) ∈ M); or ($x_m = y_n$) $\langle x_1 \cdots x_{m-1} \rangle \langle y_1 \cdots y_{n-1} \rangle$
  - Case 2: x_m is not matched; or ($x_m \leftrightarrow -$) $\langle x_1 \cdots x_{m-1} \rangle \langle y_1 \cdots y_n \rangle$
  - Case 3: y_n is not matched. ($- \leftrightarrow y_n$) $\langle x_1 \cdots x_m \rangle \langle y_1 \cdots y_{n-1} \rangle$ — 2 δ penalties
- We can try to define our subproblems as: OPT[i, j] := the min-cost of an alignment between the substrings X[1:i] and Y[1:j].
  - Case 1: $\alpha(x_i, y_j) + OPT[i-1, j-1]$
  - Case 2: $\delta + OPT[i-1, j]$
  - Case 3: $\delta + OPT[i, j-1]$

min {

→ α(a,e) is still a match with cost α(a,e)



Eventually we need to find , which depends on , which in turn calculates the entire table i.e. $O(n \times m)$

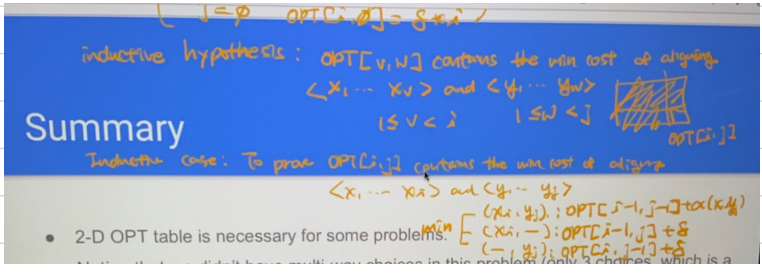
Mem –

Naive, by default we can store entire table, but at any time for row 'i' we require row 'i−1' only,

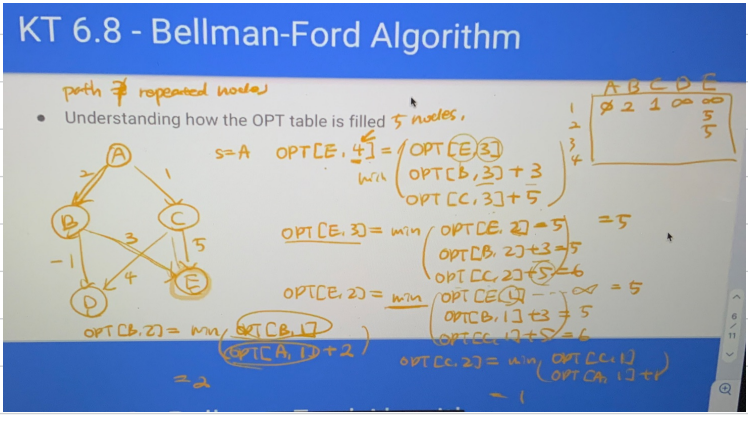
$O(2(n+1)) = O(n)$

– Proof of correctness –

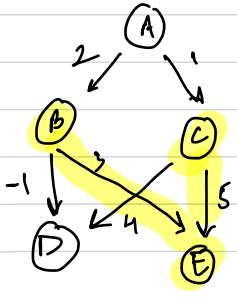
Base case – [if $i = 0$, $OPT[i,j] = j \times \delta$
if $j = 0$, $OPT[i,j] = i \times \delta$



Pblm : Shortest Path Problem (can handle -ve weights but no -ve cycle)



- Defining subproblem & recurrence :-
  - OPT[v,k] : Optimal soln from s → v using k-edges.
  - For any path P from s → v
    - 'P' has atmost k−1 edges [OPT(v, k−1)]
    - P's last edge is (w,v)



n nodes in graph, thus '4' is max nodes!

$$OPT[E,4] = \min \begin{cases} OPT[E,3] & 5 \\ OPT[B,3]+3 & \\ OPT[C,3]+5 & \end{cases}$$

$$OPT[E,3] = \min \begin{cases} OPT[E,2] & -5 \\ OPT[B,2]+3 & -5 \\ OPT[C,2]+5 & -1+5=6 \end{cases} \quad 5 \checkmark$$

$$OPT[E,2] = \min \begin{cases} OPT[E,1] & -\infty \\ OPT[B,1]+3 & -2+3=5 \\ OPT[C,1]+5 & -1+5=6 \end{cases}$$

$$OPT[C,2] = \min \begin{cases} OPT[C,1] & (1) \\ OPT[A,1]+1 & \end{cases}$$

$$OPT[B,2] = \min \begin{cases} OPT[B,1] & (2) \\ OPT[A,1]+2 & \end{cases}$$

- RUNNING TIME COMPLEXITY :-

$$OPT[E,K] = \min \begin{cases} OPT[E,K-1] \\ OPT[w,K-1] \\ \vdots \end{cases}$$

Now, in worst case 'E' can have 'n-1' neighbours & to find min. among 'n' values takes $O(n)$

And OPT table has n^2 values i.e. say OPT

	A	B	C	D	E
1	ϕ	2	1	∞	∞
2					
3					
4					

$n \downarrow$, $n \rightarrow$

$\therefore$ Overall $O(n^3)$

Q. Can we have -ve cycle in G?

- If we do then need to consider 2 factors apart from just the cost i.e. say the # of edges as well, so that we have a lower bound.

- How to detect?

* Need to check for 'n' edges & not just 'n-1' i.e. $OPT[V, n]$, thus if there is a -ve wt. adding it would further ↓ cost.

- Floyd - Warshall -

- Computes pair-wise shortest path distance. i.e. $\{V_1, \ldots, V_n\}$ all possible pairs i.e. $(V_1, V_2), (V_1, V_3), (V_2, V_3) \ldots$
- $OPT_k[i,j]$ - shortest path distance from V_i, V_j only using $\{V_1, \ldots, V_k\}$ as intermediate nodes.

Base Case - $K = \phi$ $\quad OPT_\phi[i,j]$ = weight of edge from (V_i, V_j) ∞ otherwise

$OPT_2[i,j] = \min \begin{cases} OPT_1[i,j] \\ OPT_1[i,2] + OPT_1[2,j] \end{cases}$ $\quad OPT_1[i,j] = \min \begin{cases} OPT_0[i,j] \\ \text{cost of } V_i \to V_1 \to V_j \text{ (i.e. } OPT_0[i,1] + OPT_0[1,j]) \end{cases}$

Thus Generalizing -

- Base case ($K = 0$) :- $OPT_0[i,j] = c(i,j)$
- Main case :- $OPT_k[i,j] = \min\left(\boxed{OPT_{k-1}[i,j]}, \boxed{OPT_{k-1}[i,k] + OPT_{k-1}[k,j]} \right)$

Not using 'k' ↑ $\quad$ using k

⇒ Run Time Complexity :-

Now, we have 3 indices to take care of i.e. $OPT[i,j,k]$ & filling this would make it $O(n^3)$

& each operat^n takes constant amt. of time, thus $O(n^3)$

[Note: How do I know I don't visit the same node again in $OPT_{k-1}[i,k]$ & $OPT_{k-1}[k,j]$

- Say $V_i \to V_3 \leftrightarrows V_5$, $V_3 \to V_6$ The cost computat^n makes sure we don't choose it as $V_i \to V_3 \to V_5 \to V_3 \to V_6$ is expensive]

Inductive Hypothesis :-

Proof of Correctness

Base case : $k = \phi$ (no intermediate node) shortest path $OPT_0[i,j]$ = direct edge from v_i to v_j

Inductive Hypothesis: Assume that $OPT_k[i,j]$ has the shortest path distance from v_i to v_j using $(v_0, \dots, v_k)$ as intermediate nodes

$OPT_0[i,j]$ = cost of edge (v_i, v_j)

$c(v_i, v_i) = 0$

if no edge $v_i \to v_j$, $c(i,j) = \infty$

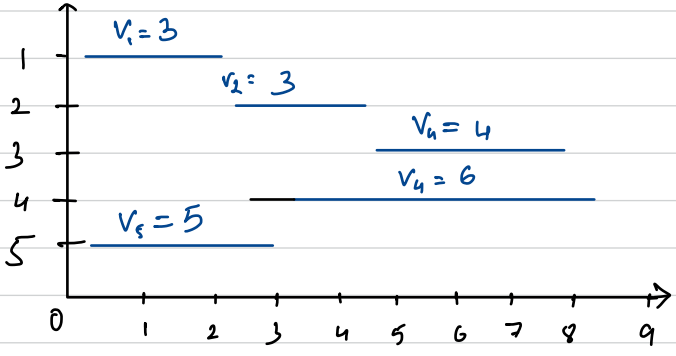
$\phi \le k \le n-1$ by I.H.

Inductive case : $OPT[n, i, j] = \min$ [$OPT[n-1, i, j]$, $OPT[n-1, i, n] + OPT[n-1, n, j]$]

HOW TO ACTUALLY FIND PATHS :-

How to Obtain Shortest Paths?

- Computing shortest path distances (numbers) vs. constructing paths
- In all three shortest-path algorithms, we only need to store the "predecessor node" per destination.
  - Dijkstra's Algorithm: Each time d[v] is improved (i.e., decreased), store pre[v] := w.
  - Bellman-Ford Algorithm: Each time OPT[v, k] is improved, store pre[v, k] := w. Each time OPT[v] is improved, store pre[v] := w.
  - Floyd-Warshall Algorithm: Each time OPT[i, j, k] is improved, store pre[i, j] := k. Each time OPT[i, i] is improved,



Greedy Algorithm would choose -

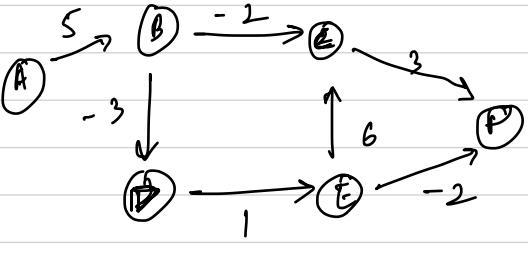
$\{3, 2, 1\} \rightarrow 10$ } sum = 16

$\{4\} \rightarrow 6$

Optimal solution -

$\{4, 1\} \rightarrow 9$ & $\{5, 3\} \rightarrow 9$

thus, sum = 18



dist	A	B	C	D	E	F
1.	0	∞	∞	∞	∞	∞
2.	0	5	3	2	3	
3.						
4.						
5						

d[F] = d[C]

3 - 2 - (1)

Classnotes

- Dijkstra's Algorithm :-

Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)

Implementation #1

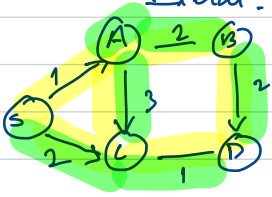
```
1  S <- {}, d[*] <- infinity, d[s] = 0
2    while (S != V):
3      pi[*] <- infinity
4      for v in (V \ S):
5        for e = (u, v) with u in S:
6          pi[v] <- min(pi[v], d[u] + l[e])
7      v = arg min(pi[v])
8      S.add(v), d[v] <- pi[v]
9    output d
```

|V|=n

- for n times
- initialize Π = best known distance from S
- for every unexplored node v
- for every incoming edge of v
- update pi[v]
- find 'min' out pi
- add that node to S (explored)

'while' Iteration: 1

Iteration: 2



'Line-4-6' start with exploring all nodes as everything is unexplored.

2nd - Iteration: all edges apart from 'S→A'

Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$$

add v to S, and set d(v) = π(v).

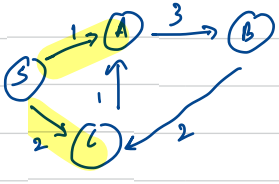
shortest path to some u in explored part, followed by a single edge (u, v)

Implementation #2

```
1   S <- {}, d[*] <- infinity
2   pi[*] <- infinity, pi[s] <- 0
3   while (S != V):
4     v = arg min(pi[v]) with v in (V \ S)
5     S.add(v), d[v] <- pi[v]
6     for e = (v, x) with x in (V \ S):
7       pi[x] <- min(pi[x], d[v] + l[e])
8
9   output d
```

- for n times
- ~~pick min~~
- add v to S
- update Π using outgoing edges of v

Line 4 - find min out of diff. n, (n-1), (n-2), ..., 1



π			
S	0	0	0
A	∞	1	1
B	∞	∞	4
C	∞	2	2

Thus 'for' loop will only be executed not more than once for a visited nodes

CN

Implementat^n #3, using Min. Heap (Binary tree

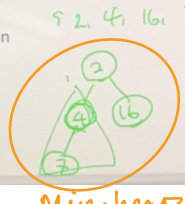
```
1  S <- {}, d[*] <- infinity
2  pi[*] <- infinity, pi[s] <- 0
3
4  while (S != V):
5    v = arg min(pi[v]), v in (V \ S)
6    S.add(v), d[v] <- pi[v]
7    for e = (v, x) with x in (V \ S):
8      if pi[x] > d[v] + l[e]:
9        pi[x] <- d[v] + l[e]
10 output d
```

O(n²)

```
1  S <- {}, d[*] <- infinity
2  pi[*] <- infinity, pi[s] <- 0
3  PQ <- Priority Queue over pi[*]
4  while (S != V): // Or (!PQ.isEmpty())
5    v = PQ.extractMin()
6    S.add(v), d[v] <- pi[v]
7    for e = (v, x) with x in (V \ S):
8      if pi[x] > d[v] + l[e]:
9        PQ.updateKey(x, pi[x] = d[v] + l[e])
10 output d
```

O(n·log n)

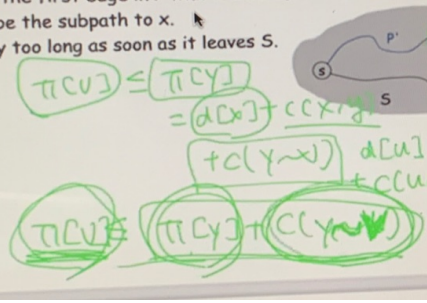
Line 5:	O(|V|) per iteration	vs.	O(log |V|) per iteration
Line 6:	O(1) per iteration	vs.	O(1) per iteration
Line 9:	O(1) per update	vs.	O(log |V|) per update
Overall:	O(|V|² + |V| + |E|)	vs.	O(|V| log |V| + |V| + |E| log |V|)



Thus changing the value results in reorganising the value

Min-heap e.g. [smallest element at top]

Invariant. For each node u ∈ S, d(u) is the length of the shortest s-u path.
Pf. (by induction on |S|)
Base case: |S| = 1 is trivial.
Inductive hypothesis: Assume true for |S| = k ≥ 1.
- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length π(v).
- Consider any s-v path P. We'll see that it's no shorter than π(v).
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



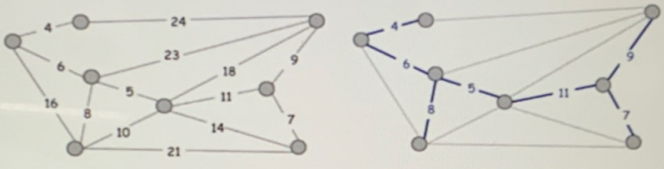
→ $\bar{\pi}(v) \leq \bar{\pi}(y)$
↳ by our defⁿ
& $\bar{\pi}(y) = d(x) + c(x,y)$
& for going to 'v' from 'y'
∴ $d(x) + c(x,y)$
$+ c(y,v)$
∴ $\bar{\pi}(v) < \bar{\pi}(v)$
↳ through y

CN

-Minimum Spanning Trees

Minimum Spanning Tree

Minimum spanning tree. Given a connected graph G = (V, E) with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



G = (V, E)

T, $\Sigma_{e \in T} c_e = 50$

Cayley's Theorem. There are n^{n-2} spanning trees of K_n.

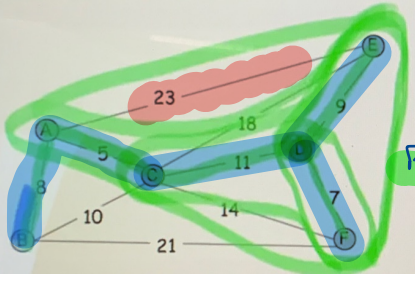
↑ can't solve by brute force

Kruskal's algorithm. Start with T = ϕ. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

→ Diff Approaches



Prim's Algorithm
↳ Start from 'D' & find min. values & select

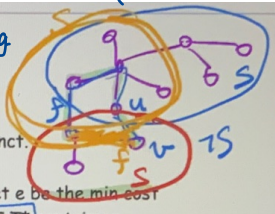
CLASS NOTES

'S' can be any subset

07/10/21

Greedy Algorithms

e.g.

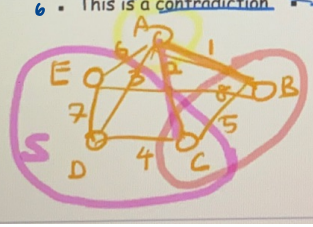


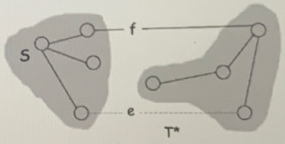
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min-cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf. (exchange argument) proof by contradiction

1. Suppose e does not belong to T*, and let's see what happens.
2. Adding e to T* creates a cycle C in T*.
3. Edge e is both in the cycle C and in the cutset D corresponding to S (D is a set of edges connecting S and ~S) ⇒ there exists another edge, say f, that is in both C and D.
4. T' = T* ∪ {e} − {f} is also a spanning tree.
5. Since $c_e < c_f$, cost(T') < cost(T*).
6. This is a contradiction. ▪





9

1) Adding any edge to a MST creates a cycle.
3) And if there is a cycle then for sure there'll be another edge say 'f' that also is present in S & S' & in cutsets 'D'.

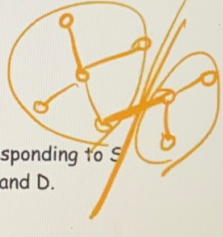
Greedy Algorithms

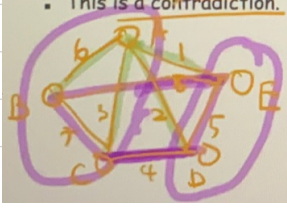
Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

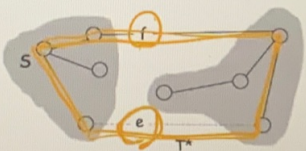
Pf. (exchange argument) proof by contradiction

- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* creates a cut S in T*.
- Edge f is both in the cycle C and in the cutset D corresponding to S ⇒ there exists another edge, say e, that is in both C and D.
- T' = T* ∪ {e} − {f} is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪





f=(B,E)
c_f=8



10

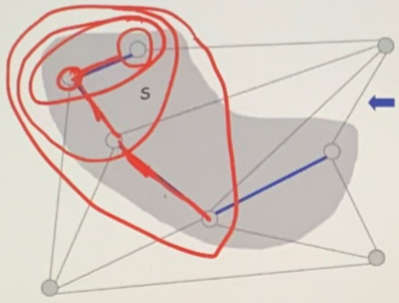
start with the opp. of the claim.

Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize S = any node.
- Apply cut property to S.
- Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.

e.g. start from 's'.



11

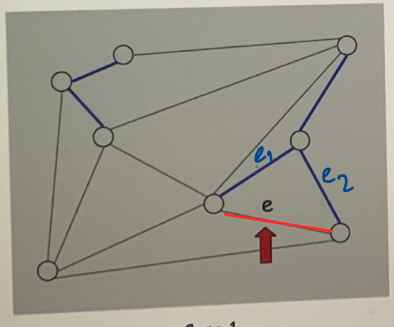
- Prim's algorithm works by repeatedly applying 'cut-property' i.e. choosing smallest edge at each pt

- Each time we apply cut-prop. our 'S' ↑ by 1 and we do it until 'S = V'.

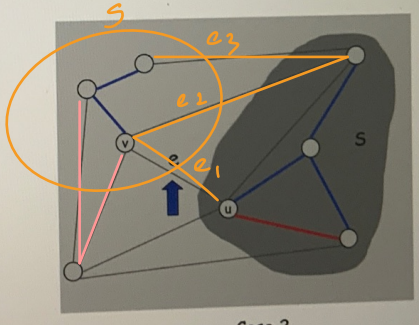
Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

- Kruskal makes use of both the cut & the cycle property.
- Unlike prim's algorithm it doesn't start from any root node but rather considers diff. edges.

- As in case 1 it applies cycle property not to include edge 'e'.

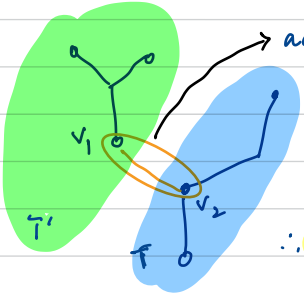
- For case 2 either we can merge the trees (—) or grow the trees keeping (—) ## same.

cost(e1) < cost(e)
cost(e2)

cost(e1) < cost(e2)
cost(e3)

[as per Kruskal's def"]

- Checking if adding 'c' makes a cycle:-



→ adding this won't create a cycle! Thus following the tech. to check if node was visited / not is a bad implementation

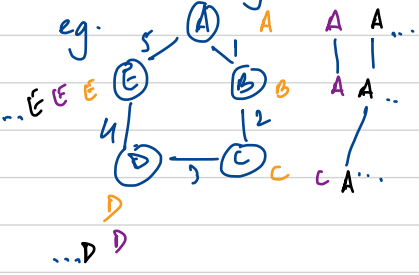
∴ does e(u,v) to T create a cycle
⇒ are u,v in same connected component T.

~~if(!exp[v1] || !exp[v2])~~
~~{ add e to T;~~
~~exp[v1] = true;~~
~~exp[v2] = true;~~
~~}~~

Soln :- < Union find Data struct >

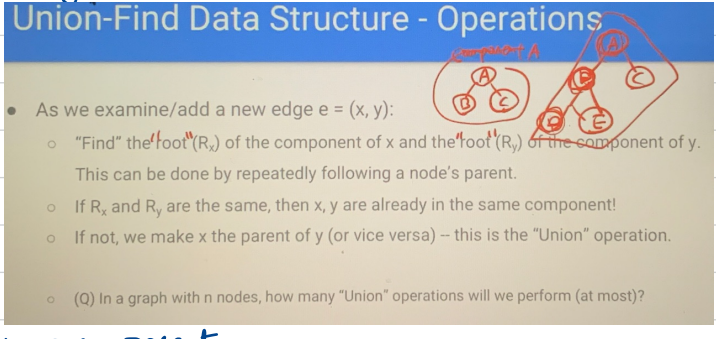
1. - 'Find' the root (R_x) of component 'x' & R_y for 'y'.
   - If $R_x = R_y$ then cycle

→ Understanding Time Complexity



Find(u) → O(n)

↳ Initially everyone are their own parent



$e \leq n^2$ [from Floyd-Warshall] → $O(e \log e)$ → $O(n)$ →

2. $e \log n \Rightarrow O(e \log n)$

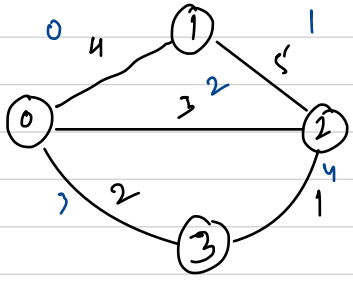
$O(e \log n) < O(e \cdot n)$ → $O(e \cdot n)$ { $O(e)$ → $O(n)$ →

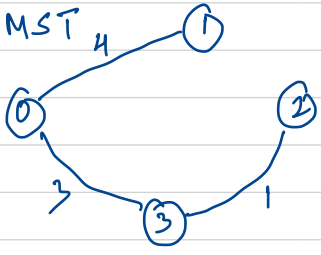
Kruskal's algm (G) {
sort the edges in E in ascending order
T = { } → prepare p[] MakeUnionFind
for each edge e = (u,v) in E {
if adding e doesn't create cycle in T ~~find(u) == find(v)~~
add e to T;
Union(find(u), find(v))
}
}

→ OPTIMIZATION 1 : UNION BY RANK :-

Rank - decided by size of component., bigger connected component has higher rank.

TC03





⇒ **IMP :**

I) S[i] :- set containing i (depends on 'n')

↳ CreateSets(n)

S = [0 , 1 , 2 , 3]

2, 3 → 0 , 1 , 2 , 2

(0, 3) → 0 , 1 , 0 , 0

sets[0 → 0 , 0 , 0 , 0

sets[2

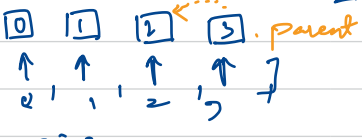
0 , 2 0 , 1

tmp. element

'2' '3'

II) Using Tree :-

- Each vertex is a Node in tree

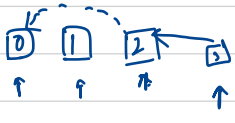


sets[rooty].parent = sets[rootx]

S[3].par S[2] (0,3)

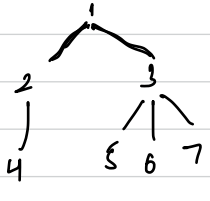
rx 0 2

ry



sets[2].p = S[0]

⇒

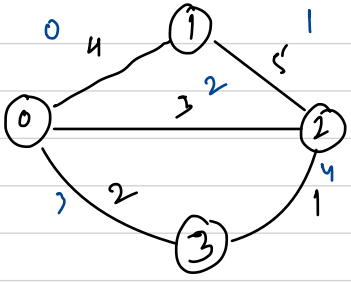


S = [-1, -1, -1, -1, -1, -1, -1, -1]

0 1 2 3 4 5 6 7

[-1, -1,

· Parent



(2,3) -
(0,3) ✓
(0,2) ✗
(0,1)
(1,2)

3, 3, 3, -1
(0,3) 3, -1, 3, -1
(2,3) -1, -1, 3, -1
Parent = [-1, -1, -1, -1]
0 1 2 3

R_x : F(0) 3
R_y : F(1) 1
L(3,1)

||||||||||

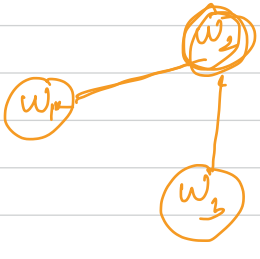
0, 0, 0, 1
Rank = [0, 0, 0, 0]
0 1 2 3

a a b a a b a b a
a b a b d b d

a b a b a
a

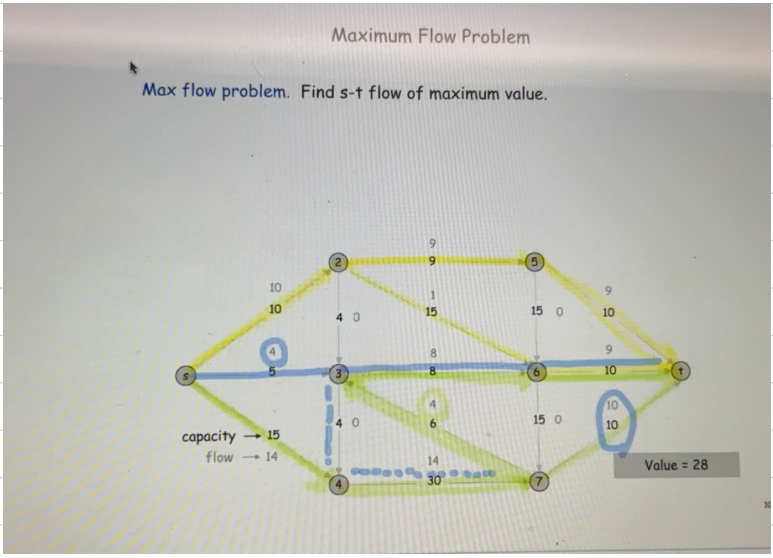
n = 5
f = n - 2

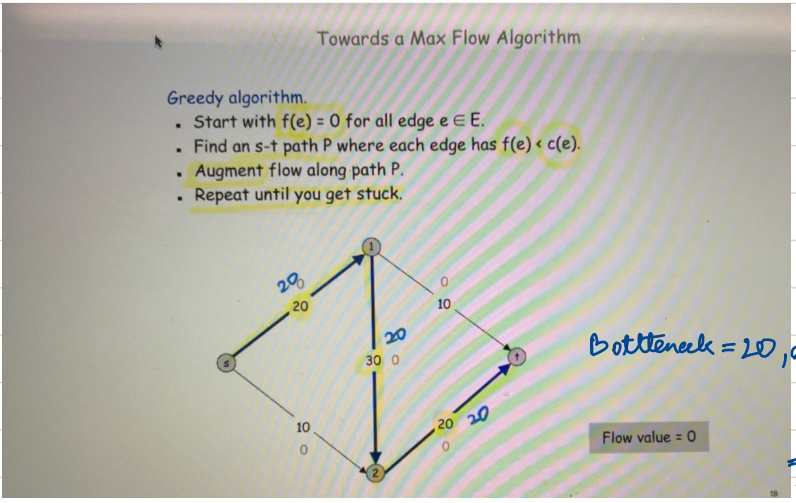
f = 3



CLASSNOTES

→ Max-Flow Algorithm:



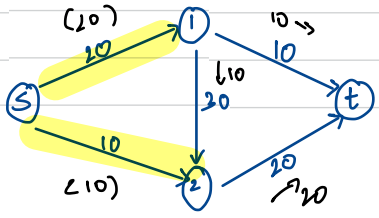


f(e) → Amt of flow used.

Bottleneck = 20, even though middle can carry 30!

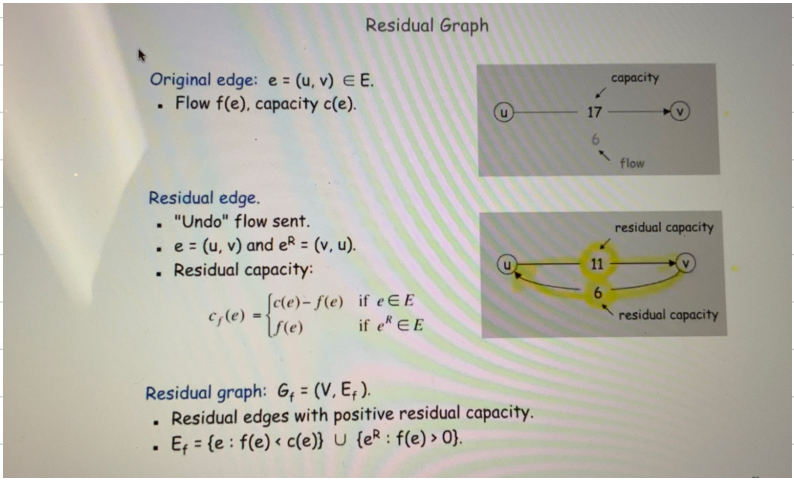
⇒ Flow value = 20

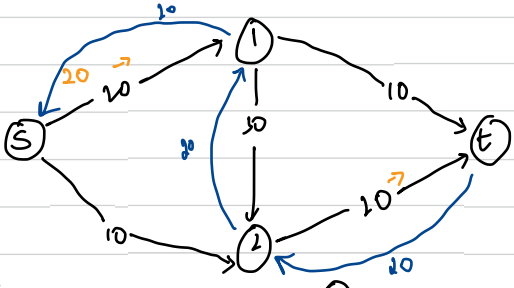
But this is not optimal!



optimal val = 30, Thus how do we go from Greedy to optimal?,
<s-2> even if we send 10, <2-t> is already occupied!!!

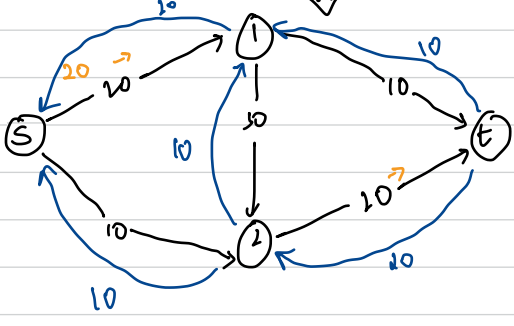
→ Residual Graph :





- Residual Graph After 1 flow.
- For 2nd flow we can use S→2→1→t (using backward edge)
  - ↳ bottleneck = '10'
    As min is '10'.

Flow 2:

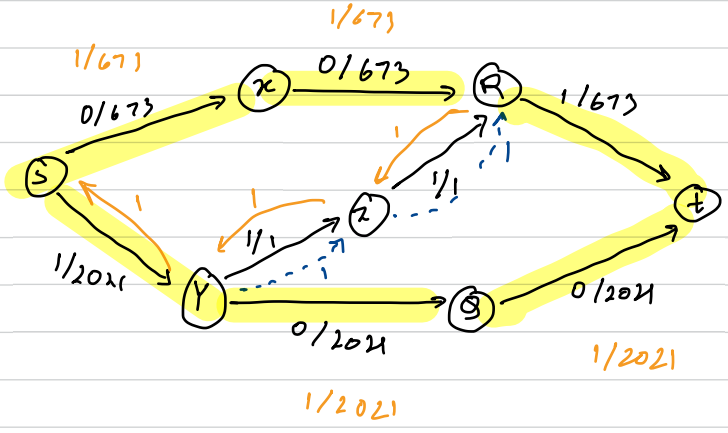


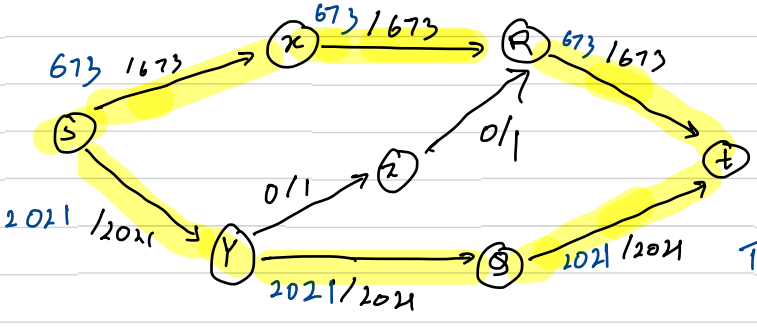
→ No flow to 't' as all edges are used up.
- There's actually no backward edge but what we mean is that earlier we had 20 through ①—②,
  ▷ Backward edge of '10' means 20-10='10'

32.39

→ Ford-Fulkerson Algorithm :-

1. find path
2. Compute Bottleneck
3. augment path





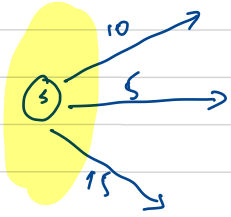
Max-flow:

$s \to x \to R \to t$

$s \to Y \to Q \to t$

Thus max-flow Value = 2694

- Cuts



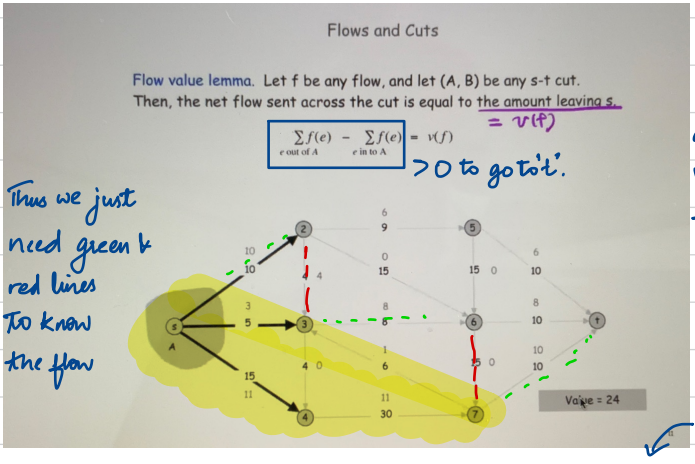
A s-t cut is a partition

Capacity = 10 + 5 + 15 = 30

$cap(A,B) = \sum_{e \text{ out of } A} c(e)$ → cap. of a cut.

Min. Cut Pblm :- <Find an s-t cut of min. capacity>

- After 'cut' you can't have more than 2 partitions!



>0 to go to 't'.

Thus we just need green & red lines To know the flow

$\sum_{e \text{ into } v,\ v \in A,\ v \notin s} f(e) - \sum_{e \text{ out of } v} f(e) = 0$ (let's say for 's' vi)

+

$\sum_{e \text{ out of } s} f(e) - \sum_{e \text{ into } s} f(e) = v(f)$

↳ 0

$$\sum_{e \text{ out of } v,\ v \in A} f(e) - \sum_{e \text{ into } v,\ v \in A} f(e) = v(f)$$

Flow value lemma ↑

Value = 10 − 4 + 8 − 0 + 10

= 24

(s,2) (s,3) (3,4) (3,6) (4,t) (7,t)

−

(s,2) (7,3) (2,3) (3,4) (s,4) (4,7) (6,7)

$e = (u,v)$

if $u, v \in A$

$e' \notin (u', v')$

either u'/v' is not in 'A'.

(Just flow Value)

10 + 8 + 10 − (4)

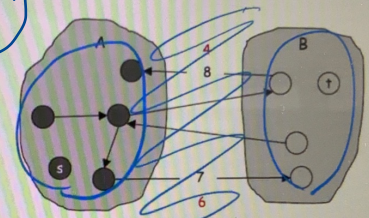
= 24

Weak duality. Let f be any flow. Then, for any s-t cut (A, B) we have $v(f) \le cap(A, B)$.

$x - y \le x$, if $y \ge 0$

Pf.

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \le \sum_{e \text{ out of } A} f(e) \le \sum_{e \text{ out of } A} c(e) = cap(A, B)$$



$v(f)$ = Σ outgoing flow − Σ incoming flow

value of flow

Certificate of Optimality :-

Max-Flow Min-Cut Theorem

Augmenting path theorem. Flow f is a max flow iff there are no augmenting paths.

Max-flow min-cut theorem. [Elias-Feinstein-Shannon 1956, Ford-Fulkerson 1956]
The value of the max flow is equal to the value of the min cut.

Pf. We prove both simultaneously by showing TFAE: (the following are equivalent)

(i) There exists a cut (A, B) such that $v(f) = cap(A, B)$. ($v(f) \le cap(A,B)$)
(ii) Flow f is a max flow.
(iii) There is no augmenting path relative to f.

(i) ⇒ (ii) This was the corollary to weak duality lemma.

(ii) ⇒ (iii) We show contrapositive. (contrapositive: $p \to q \iff \neg q \to \neg p$; ↳ contra-positive)

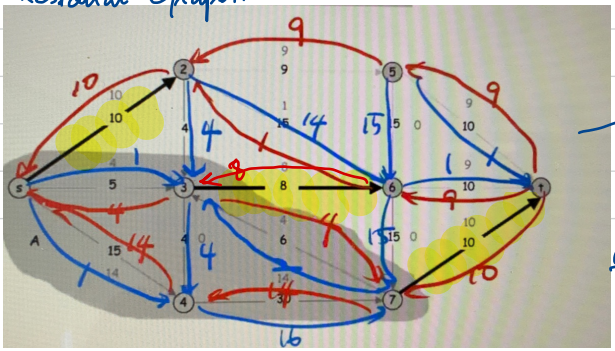
- Let f be a flow. If there exists an augmenting path, then we can improve f by sending flow along path. ↖ in residual graph

↖ then 'f' is not max flow!

No augmenting path → there is a cut (A, S) s.t.

$$v(f) = \left(\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in } V-A} f(e)\right) \le \sum c(e) = cap(A,B)$$

Residual Graph:-



1. for every e into A, $f(e) = 0$
2. for every edge 'e' out of A, $f(e) = c(e)$

Actual edge leaving 'A', G_A = 's − 3 − 4 − 7'

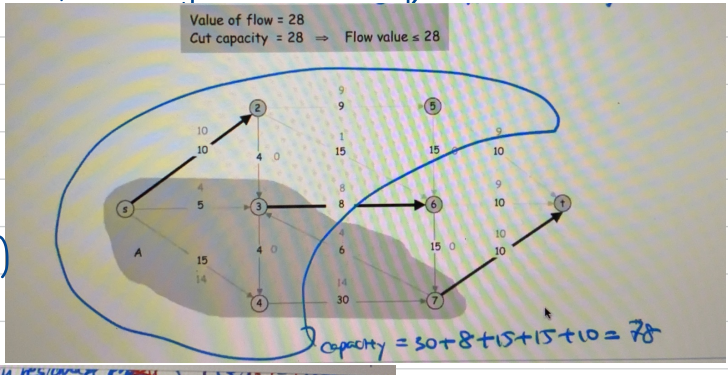
Leaving A: (s-2), (3,6), (7,t)

↳ $f(e) = c(e)$

Corollary: let 'f' be any flow, & let (A,B) be any cut. If v(f) = cap(A,B), then 'f' is a max flow & (A,B) is a min. cut.

Capacity :- 28 - 78

flow :- '1 - 28'



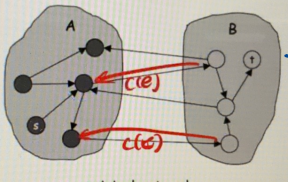
found in residual graph

↑

no augmenting path → $v(f) = cap(A,B)$

there is a cut (A,B) s.t. ↑

- Let f be a flow with no augmenting paths.
- Let A be set of vertices reachable from s in residual graph.
- By definition of A, $s \in A$.
- By definition of f, $t \notin A$.

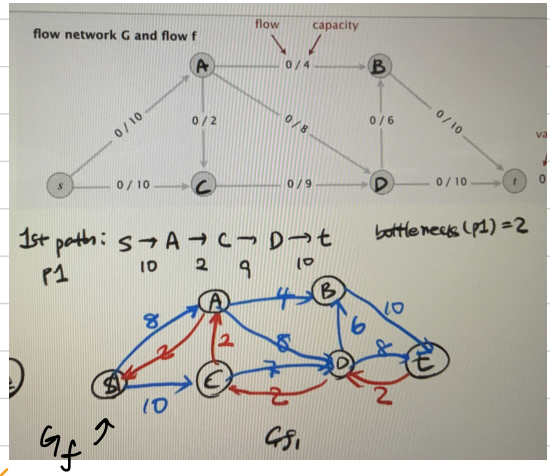


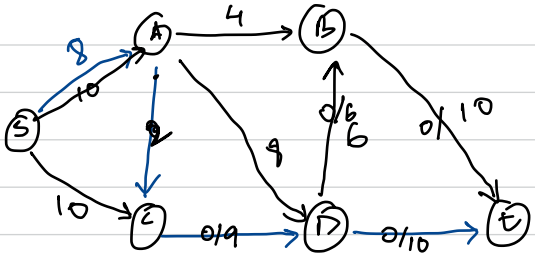
- Outgoing edge fully used.
- Incoming edge not used

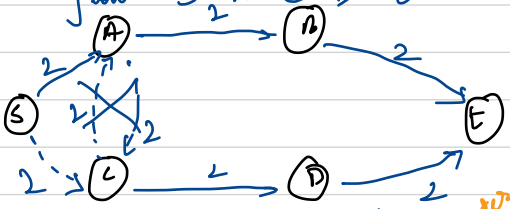
<check prev slide for proof>

'HW8 : 21/10'

Algo Classnotes :







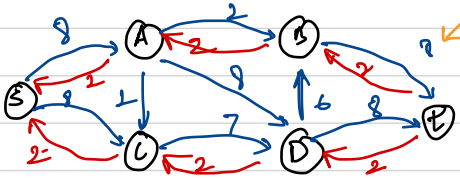
after f_2

2nd path: S → C → A → B → t

Residual Graph:

forward edge: remaining capacity

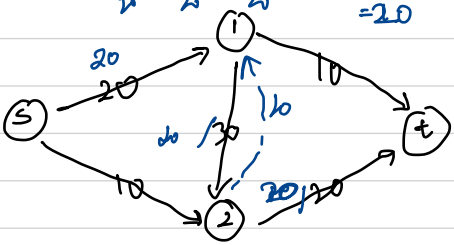
backward edge: what we can substract from existing.

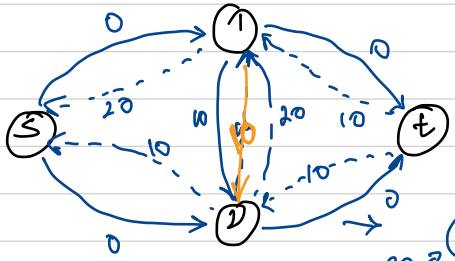


And actually there will not be any edge btw 'A & C', thus we can send S → A & S → C thus total flow is '4'.
2 2

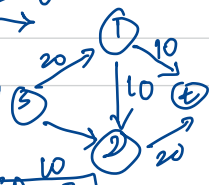
S → 2 → 1 → t
10 10 10

E.g. P1: S → 1 → 2 → t, bottleneck = 20
20 20 20





2 → 1 can send '20'



Electives AI , ML , DSD , NP

24 22 20 28 capacity

Q1. 1 std. has to take 1 elective before grad"

$24 + 22 + 20 + 28$

Q2. 1 " " " " 2 " " "

$\frac{24 + 22 + 20 + 28}{2}$

How to find max-flow!

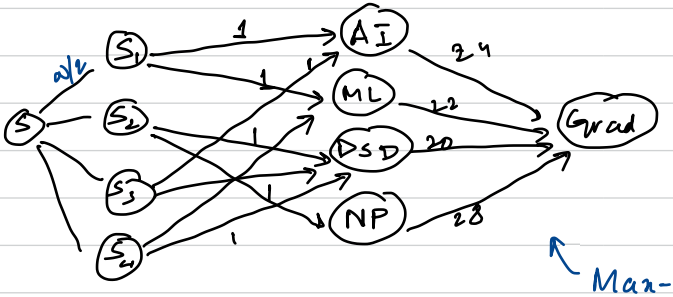
Q3. " " " " 2 " " 1 preference

$s1 = \{AI, ML\}$

DSD, NP

AI, DSD

ML, DSD

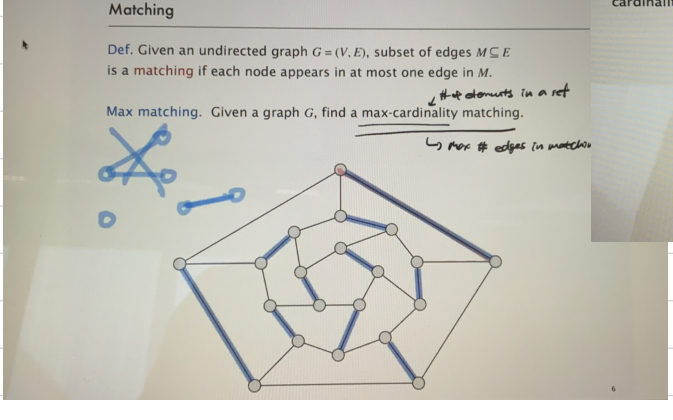


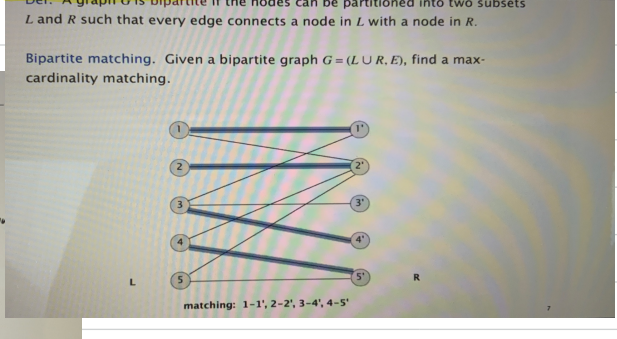
Max-flow

Max students getting desired courses!

Bipartite graph—

Bipartite Matching





	Course	1	2	3	4	
$n=4$, $m=3$	classroom	a_1	a_2	a_3	a_4	# stds
		A_1	A_2	A_3	A_4	hrs
a_i – course i		10	11	12	1	

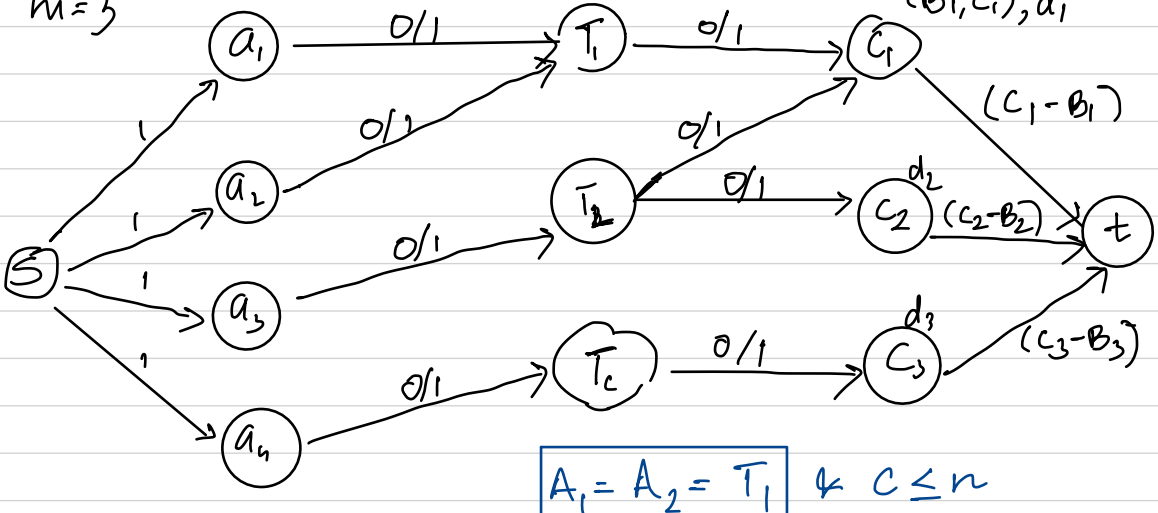
C_1 – $\{d_1, B_1, C_1\}$

C_2 – $\{d_2, B_2, C_2\}$

C_3 – $\{d_3, B_3, C_3\}$

$n=4$
$m=3$

Courses (Distinct A_n) → Time $\{A_n\}$ Classroom



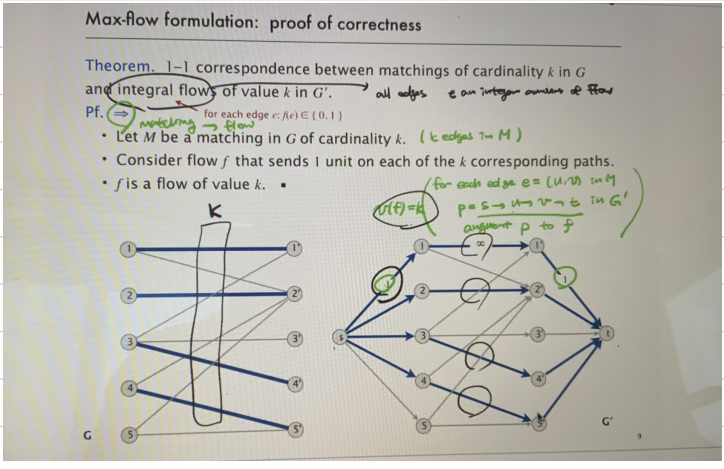
$A_1 = A_2 = T_1$ & $C \leq n$

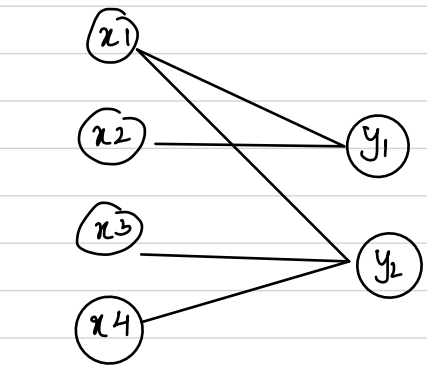
Max-flow formulation: proof of correctness

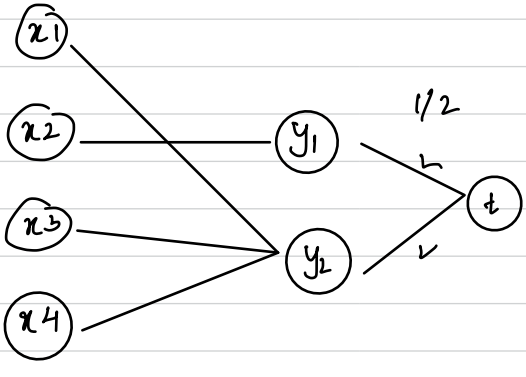
Theorem. 1–1 correspondence between matchings of cardinality k in G and integral flows of value k in G'.

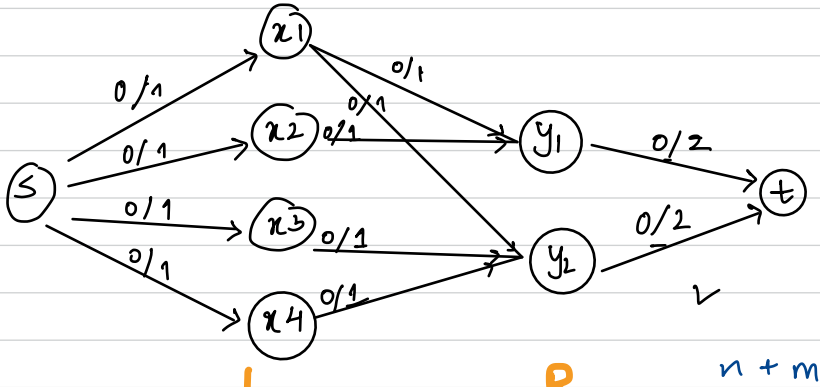
Pf. ($\Rightarrow$) matching $\rightarrow$ flow

- Let M be a matching in G of cardinality k.
- Consider flow f that sends 1 unit on each of the k corresponding paths.
- f is a flow of value k. ▪









n

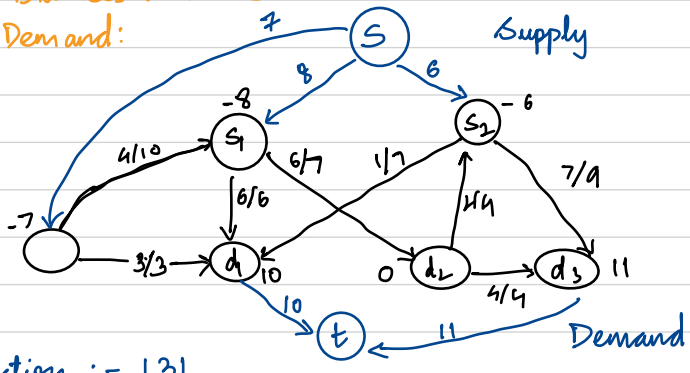
$n + m + nm$

$(nm \times n)$

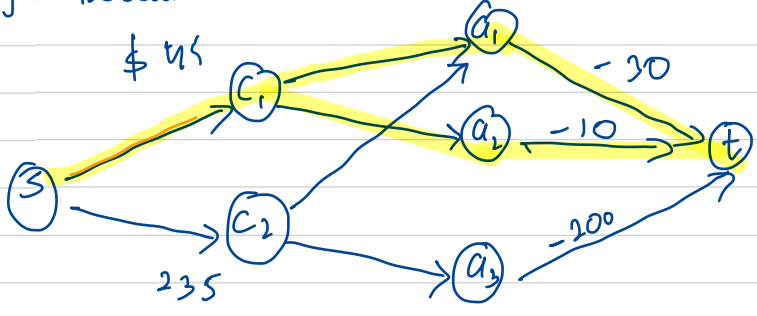
Class Notes

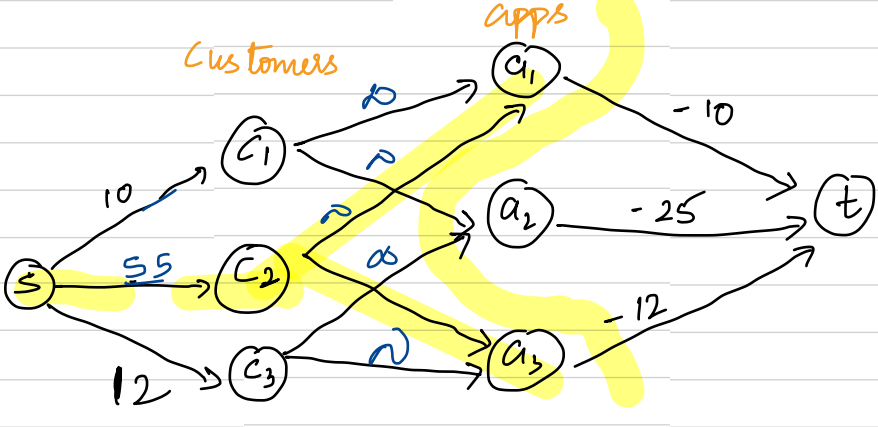
23/10/21

- Multiple sources & sinks :-
- Supply & Demand:



Project selection :- L31

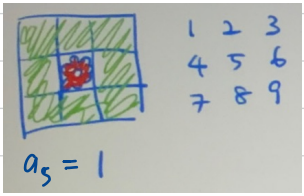




Classnotes

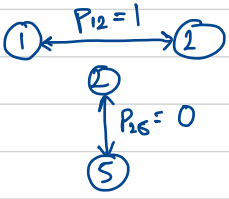
Image segmentat^n: <Refer Class slide>

a_i - likelihood of



$a_5 = 1$, $a_1, a_2, a_3, a_4, a_6, a_7, a_8, a_9 = 0$, $b_5 = 0$
rest all 1

P_{12} → penalty between 1 & 2 (penalty)
i.e



P_{25} !

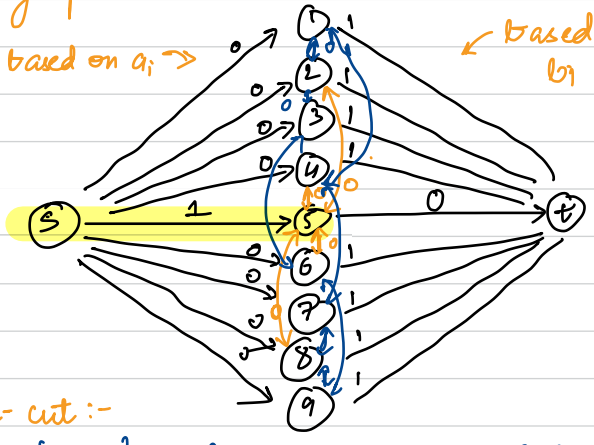
$P_{12}, P_{23}, P_{14}, P_{36}, P_{47}, P_{78}, P_{89}, P_{69} = 1$ (same region thus separat^n penalty is high)

$P_{25}, P_{45}, P_{56}, P_{58} = 0$ (very diff. color)
(separat^n penalty is low as 'ij' are in diff. region)

$P_{ij} \leftrightarrow : 1 \leftrightarrow : 0$ P_{ij}

→ Creating a graph:

based on a_i → based on b_i



Formulat^n of min-cut :-

A = {s, 5} B = {1, 2, 3, 4, 6, 7, 8, 9, t} → cut (A, B)

By def^n: Minimizing →

$$cap(A,B) = \sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{(i,j) \in E \atop i \in A, j \in B} P_{ij}$$

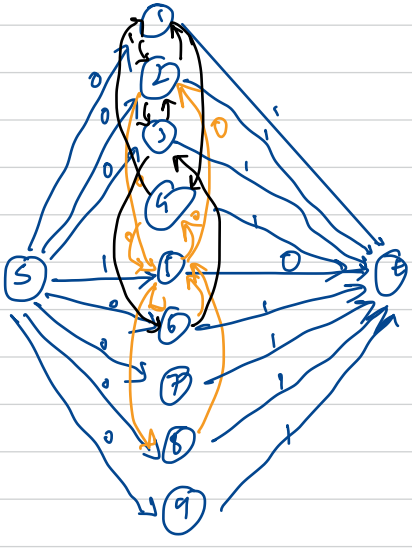
$s \not\to 5$

$s \to 1$
$s \to 2$
⋮
$s \to 9$

$5 \to t$

$2 \to 5$ ⇔ $5 \to 2$
$4 \to 5$ ⇔ $5 \to 4$
$5 \to 6$
$5 \to 8$

Black edges – val 1 (Not complete)



$A = \{S, 5\}$ $B = \{1, 2, 3, 4, 6, 7, 8, 9, t\}$

$cap(A,B) =$ $S \xrightarrow{a_1} 1$, $S \xrightarrow{a_2} 2$, $S \xrightarrow{a_3} 3$, $S \xrightarrow{a_4} 4$, $S \xrightarrow{a_6} 6$, $S \xrightarrow{a_7} 7$, $S \xrightarrow{a_8} 8$, $S \xrightarrow{a_9} 9$ | $S \to t$ b_5 | $5 \xrightarrow{p_{25}} 2$, $5 \xrightarrow{p_{45}} 4$, $5 \xrightarrow{p_{65}} 6$, $5 \to 8$ p_{58} $= 0$

$\sum_{j \in B} a_j$ → 0 | $\sum_{i \in A} b_i$ → 0 | $\sum_{(i,j) \in E, i \in A} p_{ij}$

$|A \cap \{i,j\}| = 1$

⇒ there is only 1 element in $A \cap \{i,j\}$

1. construction of flow graph $O(mn)$
2. $cap(A,B) = \sum_{i \in B} a_i + \sum_{j \in A} b_j + \sum_{(i,j) \in E, |A \cap \{i,j\}| = 1} p_{ij}$
3. min-cost (A^*, B^*) will minimize.

Time complexity

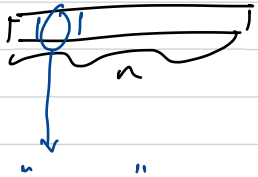
- $S + t + m \times n$:- # of nodes

$O(2mn + 2|E|) = O(mn)$ ($|E| < 4mn$)

($2|E|$: # edges)

- Conversion :-
  - $O(mn)$
  - $O(1)$ - only if we have special DS

1. use array

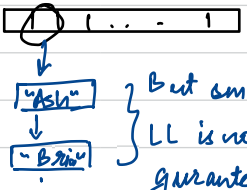


put (key, val)
→ index = H(key) mod n
→ array[index] = value;

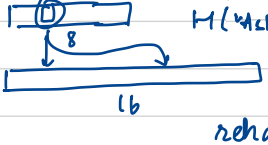
- put(Ash, '.....')
  - H(Ash) = 010
  - H(Ash) mod 8 = 2

P1: more than 'n' pair to insert?
P2: collission H("Brian") mod n = H("Ash") mod n ?
→ Dynamic / Extensible HashTable.

2. use LL, addF - O(1), lookup - O(n)
3. use SortLL, still finding right spot O(n)



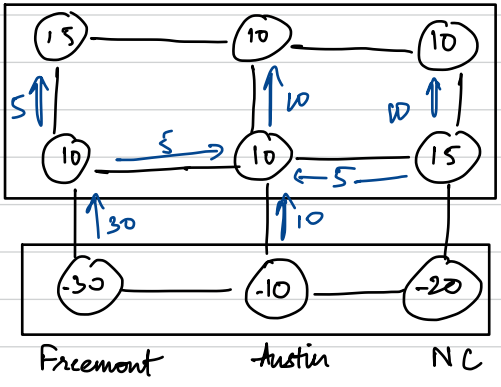
Double array when it's full / no collision:-
'Java doubles the Hashmap when the array is 75% filled 'Load factor'.



rehash - reinsert all items!

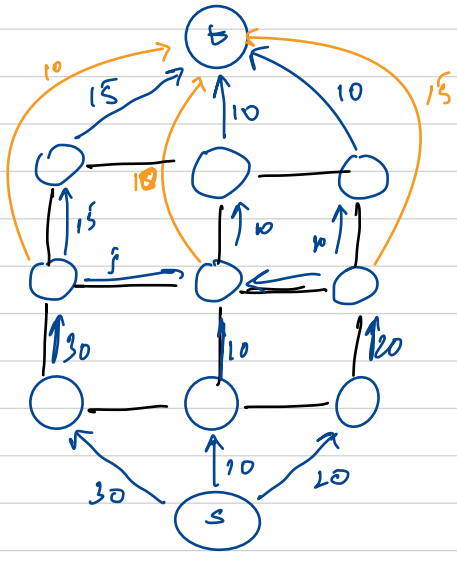
Circulat°:

supply / demand + flow ntwk



Demand

Supply



Practise Test :

4.

1. For every 'c' add (s, c) → c[1, 3]
2. menu item m, e (m, t) → cap[3, ∞]
3. For every 'c' ordered m, add e (c, m) → cap[0, 1]

} - max-flow

$O(c + m + cm) = O[cm]$

Range - Capacity :
- Use lower-bound as the priority!

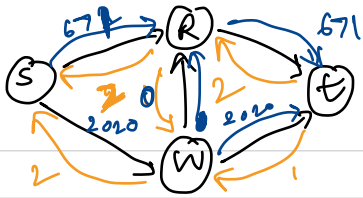
5. 20

6

Subset Sum problem

- $OPT[i, W] = \min(OPT(i-1, w], \; OPT(i-1, w - w(i)) + w(i))$

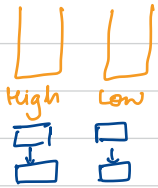
as the wt. is required!



7.

- For Kruzkal's don't consider direct^{n} for directed edge.
- add(R,Z) : comp[R] = S | add(S,W), comp[W] = S
  add(W,y) comp[y] = S | add(Z,X), comp[X] = S

8.



push(ele, priority) : O(1)

pop if(high.head != null)
  pop from high
else
  pop from low

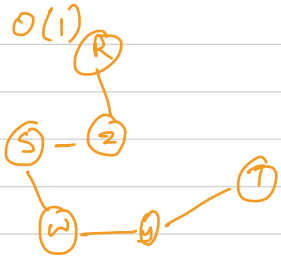
— O(1)

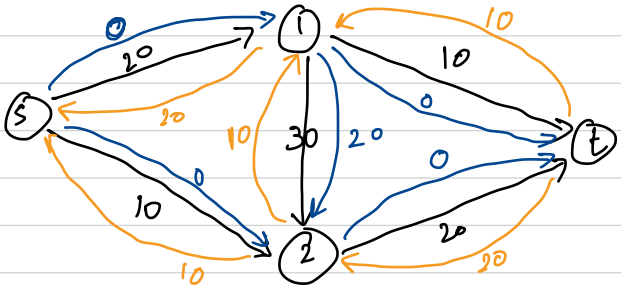
Aggr my reasoning!

multipop(k)

O(k)

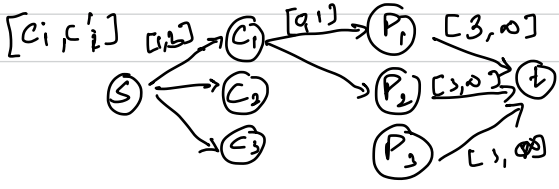
while(high.head != null & count < k)
  pop count++
while(count < k)
  pop from low



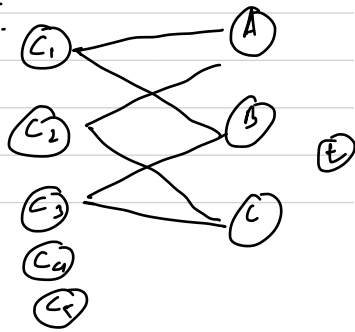


S→1 : V(f) = 20
S→2 : V(f) = 10
8 + 12

4.
[1,3]



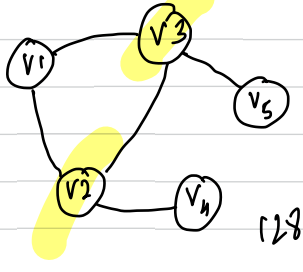
5.



Classnotes : 09/11/21

- Password cracking : very difficult to solve
  ↓ expensive
  exponential time complexity
  ↳ no efficient algorithm to solve

Polynomial
⇒ efficient!



28

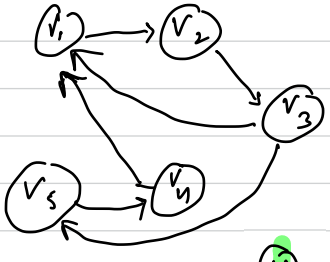
161

512

161

160

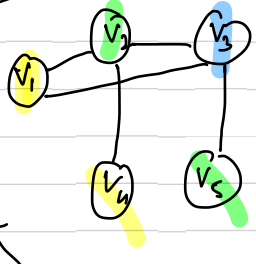
k=2



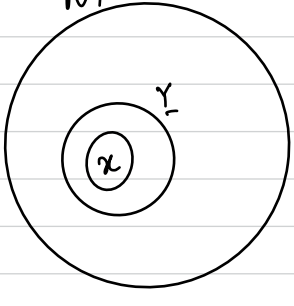
128+

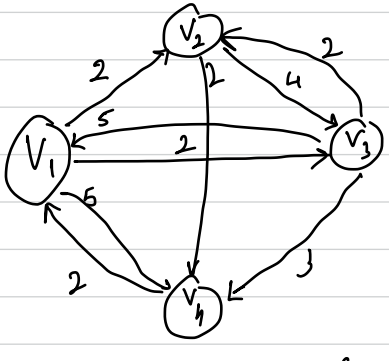
117

{V2, V3}



NP





D = 8

6 = 8

2 + 2 + 2 + 2

$V_2 \xrightarrow{2} V_4 \xrightarrow{2} V_1 \xrightarrow{2} V_3$

$V_3 \xrightarrow{2} V_2$

8

- KMP pattern Match:

Text - a b c x a b c d a b x a b c d a b c d a b c y - n

Pattern - a b c d a b c y - m

1.

a b c x
a b c d

'any prefix same as suffix? - No! start from 'a' again in pat

2.

a b c d a b x
a b c d a b c

(ab) is a prefix that's also a suffix, thus we start from 'c'

3.

x a b
a b c

- Again no suffix that's same as prefix, thus again i++

4.

a b c d a b c d a b
a b c d a b c y

'abc' matches both suffix & prefix, thus start from 'd'.

5.

a b c d a b c y
a b c d a b c y

Match, O(n+m)

- Building Longest Proper suffix (LPS):

e.g.:

lps = [0 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 2]

0 1 2 3 4 5 6 7 8

i = 1 & j = 0

step 2, i ≠ j

thus value of j! when i != j, decrement j till i = j & value = 'j+1'

↳ O(n)

Initial: j i

step 2 j != i, thus prev val. in lps i.e. 0 & again i != j [i.e. a ≠ b], thus 0, a != b, thus jumps to lps[j-1] = 2

a a b a a b a a a

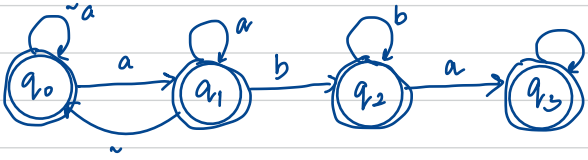
0 1 0 1 2 3 4 5

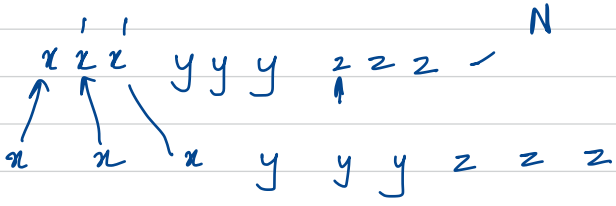
j j i

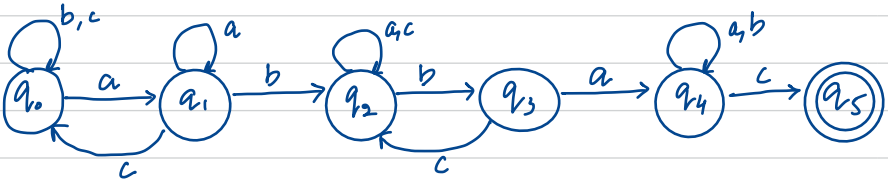
- again i != j, prev lps[j-1] = 1

- DFA :-

ab<>ba<> c


~a
a
b
a
a
b
a
~
q_0
q_1
q_2
q_3


N
x x x y y y z z z
x x x y y y z z z


b, c
a
a, c
a, b
a
b
b
a
c
c
q_0
q_1
q_2
q_3
q_4
q_5

Leet code:

→ Recursion :

I) Reverse String :

```
def reverseStr(start, end, ls):
    if (start >= end)
        return
    swap(ls[start], ls[end])
    reverseStr(start+1, end-1, ls)
```

II swap Node pairs:

e.g. 1 → 2 → 3 → 4

2 → 1 → 4 → 3

```
def swapPairs(head):
    if head == none or head.next
        return head
    fnode = head
    snode = head.next
    fnode.next = swapPair(sn.next)
    snode.next = fnode
    return snode.
```