CSE551: Homework 3

(1) Yes, the given process or algo will partition the set of activities to the fewest number of sets of mutually compatible activities

algo:-

~~begin~~: ~~while all activities aren't allocated resource~~

begin: Sort all activities in increasing order of start time

while all activities/events aren't allocated resource/auditorium:

allocate an activity to hall[k] ~~if hall[k]~~ if start time of current activity is greater than the finish time of previous activity in hall[k]

If it is not possible to allocate more activities.

change k → k+1 & Repeat this process for remaining activities that haven't been allocated yet.

The size of hall[] defines no. of sets required to partition activities.

We here sort based on start time so that no resource ~~is~~ + remains vacant & they are used efficiently.

~~We use a~~ We pick up a hall & use it to the maximum extent i.e. all the activities that can be fitted without overlap into the hall are considered. ~~No more~~ when no more activities ~~are~~ can fit, (since the hall will be busy for the times ~~when~~ of these activities), we allocate new hall only then.

Say, if some activity is wrongly fit into some hall A which means it belonged to other hall B which means activities of hall B aren't allocated optimally which contradicts since we allocate all activities/ events optimally to a particular hall before going to next hall.

This proves that the algo is correct.

Pseudo code for
S → set of all activities in sorted order of start time $\{(s_i, f_i) \mid s_i < f_i \ \forall i\}$

```
for i = 1 to n
    hall[i] = nil
k = 1
while (S not empty)
    hall(k) = SelectActivities()
    k++
```

SelectActivities():

```
temp = S[1].finish
for i = 2 to n:
    if S[i].start > temp
        A = A ∪ {i}
        temp = S[i].finish
        S.remove(i)
Return A;
```

SelectActivities() ensure all possible activities (ie max no of activities possible) are grouped together & assigned to a hall. Only then a new hall is selected.

Thus it ensures fewer no of halls / audotirua to be selected.

(2) (9) Given:

J_i → job id $1 \le i \le n$

d_i → deadline $1 \le i \le n$

t_i → exec^n time $1 \le i \le n$

To find: Best possible schedule so that fewer no. of jobs violate deadline.

Logic: While executing jobs based on deadline, we will execute a job as late as possible but before deadline, (instead of finishing it as soon as possible), because this will keep room (time) for other jobs that have prior deadline. ~~to execute.~~

~~Algo~~ Strategy: For this we will sort the jobs in descending order of deadline. We will create a priority queue that will store all the jobs that has deadline ≥ current deadline (job we are iterating) which means all such jobs can be executed ~~at the~~ at that deadline. Thus based on least exec^n time we will allocate for jobs.

When we look at i^{th} job, we will put that into queue, calculate the time (T) bet deadline of i^{th} job & deadline of $i+1^{th}$ job. For this time T, we will get job from queue & execute based on

last execution time. If only half of the job is able to complete with T time we will add the remaining part to the queue again (so that it can execute later).

Algo:-

Begin: Sort all jobs in descending order of deadline
Create a priority queue Q.

loop1: for all jobs do the foll.
Calculate T = job[i].deadline - job[i+1].deadline
add job i to queue.

loop2 while queue not empty && T != 0
Get job from queue.
~~T = T - job.exectime~~
if (T < job.exectime)
add this job to queue
with exec time =
job.execTime - T
else
T = T - job.exectime
add this job to the
list of completed jobs.
& remove it from queue

The final list of jobs will have maximum

no. of jobs that can be executed with given deadlines.

(iii) Complexity :- $O(n \log n)$

Here, we have priority queue as the data structure to store list of jobs, so that we can get the job with least execution time in $O(1)$.
But inserting a job in priority queue takes $O(\log n)$ time.
The outer loop iterates for n jobs.
The inner loop takes $(\log n)$ to insert & some ~~const~~ant c to loop over queue.
But looping over the queue for all jobs eventually sums up to n.
∴ we consider only $\log n$ term.
∴ for every insertion of job into queue
∴ Complexity comes out to be $O(n \log n)$

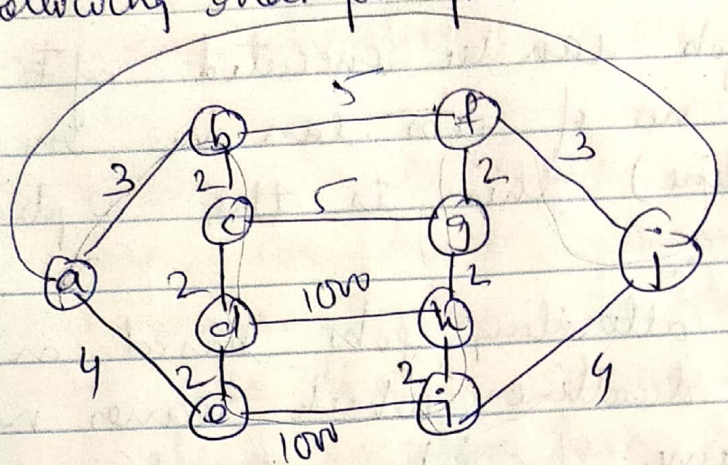
(ii) Correctness:

Since there is no penalty for executing job at any time before deadline we try to execute the job as late as possible but just before the deadline which ensures lot of time is left for other

jobs with lower deadlines.
Since no job can be executed after deadline (or fewer no of jobs can be executed after deadline) this is the optimal solution.
We are allocating jobs based on decreasing order of deadline which leaves ~~main~~ enough time for jobs with lower deadlines. If they can't execute in the maximum possible time given, they can't be executed at all. This proves that it is the optimal solution.

(3) No, the given algo can't find the optimal tour for TSP.
This algo will tend to select local optimum value every time but in the end it doesn't lead to global optimum value.
Say, while travelling from node A to, ~~to~~ node B may seem nearest node & we go to node B, but the optimal solution may not follow A→B path instead it may follow some path say A→C→B that will give least add' since C→B may be a lower value.

following example proves this



If we use given algo, it will try to find nearest city every time & eventually we will get this path

abcdeihgfjja

Add^n of all edges = 3+2+2+2+1000+2+2+2+3+10

= 1028.

Whereas optimal path ~~can~~ the will be

abfjihgcdea

Add^n of all edges = 3+5+3+4+2+2+2+5+2+2+2+4

= 36

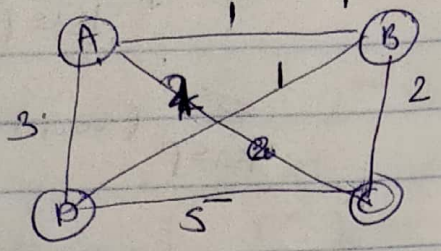
which seem way better than the above method.

Here we say that the greedy algo choose b→c since it is locally optimum but it couldn't see the e→j = 100 which lead to worst solution.

∴ we say it doesn't give optimal sol^n in

global level.

Another Example :-



Greedy sol^ :- A B D C A = 1+1+5+2 = 9

Optimal sol^ :- A C B D A = 2+2+1+3 = 8

(4) Given,

$n = 2p$, $V = (v_1 v_2 \cdots v_n)$

$W = (w_1 \cdots w_n)$

$$\sum_{1 \le i \le p} [(v_{2i-1} + w_{2i}) \times (v_{2i} + w_{2i-1})] - \sum_{1 \le i \le p} (v_{2i-1} \times v_{2i}) - \sum_{1 \le i \le p} (w_{2i-1} \times w_{2i})$$

This requires $\frac{3n}{2}$ multiplication

for $n \times n$ matrices, we have

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \begin{bmatrix} w_1 & v_2 & w_3 & w_n \end{bmatrix} = \begin{bmatrix} v_1 w_1 & v_1 w_2 & v_1 w_3 & \cdots & v_1 w_n \\ v_2 w_1 & & & & \\ \vdots & & & & \\ v_n w_1 & \cdots & & & v_n w_n \end{bmatrix}$$

From above multiplication we see that we compute a element $(v_i w_j)$ by multiplying vector v_i with vector w_j $\forall\, i \& j$

We can use above given formula to compute vector multiplication.

~~lets compute this~~

$$\sum_{1 \le i \le p} [(v_{2i-1} + w_{2i}) \times (v_{2i} + w_{2i})] - \sum_{1 \le i \le p} (v_{2i-1} \times v_{2i}) - \sum_{1 \le i \le p} (w_{2i-1} \times w_{2i}) \quad \text{-- (1)}$$

Here we see,
1st term includes v & w both
2nd term includes only v (independent of w)
& 3rd term includes only w (independent of v)
-- (2)

we will use this Result for our proof.

for v_i, w_i,
we will have to calculate 3p multiplications
-- (3)
from (1)
(since 3 times we calculate p multiplicat^n)

~~for~~
Now, for v_i, w_j ~~$i \Rightarrow 2 \le i \le n$~~
we have calculated ~~it already~~ 2nd term from (1) already since its only dependent on v_i & not w
∴ we need 2p multiplications now.
∴ for (n-1) elements of 1st row
we need $2p * (n-1)$ multiplications (4)
∴ from (3) & (4)
for 1st row, we need

$3p + 2p(n-1)$ multiplications .. (5)

Now,
Lets compute 2nd Row 1st element $V_2 W_1$,
here we see that we have already
Calculated 3rd term for W_1 in 1st Row
1st element. we store all these results
to avoid computation again & again.
∴ we need to Calculate $2p$ multiplications .. (6)

Now,
for Rest other terms,
Say $r_2 w_2$
we have Calculated V_2 & W_2 both
which means 2nd & 3rd term of (2) Result
have been Calculated. ∴ we just need
to do 1st term which is p multiplications
Same applies for rest element of 2nd .. (7)
Row

For 2nd Row
∴ $2p + p(n-1)$ .. from (6) & (7)
.. (8)

This goes for all $(n-1)$ Rows.
∴ $[2p + p(n-1)][n-1]$ → (8)
↳ for 2 to n^{th} Row.

Now, we add ① & ②
which means we add Row 1 & (n-1) Rows to get n Rows multiplicat^n.

$$= 3p + 2p(n-1) + \{2p + p(n-1)\}(n-1)$$

$$= 3p + 2pn - 2p + 2pn - 2p + p(n-1)^2$$

$$= 3p + 2pn - 2p + 2pn - 2p + pn^2 - 2pn + p$$

$$= 4pn + pn^2 - 2pn$$

$$= 2pn + pn^2$$

$$= 2\left(\frac{n}{2}\right)n + \left(\frac{n}{2}\right)n^2 \qquad \because p = \frac{n}{2}$$

$$= n^2 + \frac{n^3}{2}$$

$$= \frac{n^3}{2} + n^2$$

∴ Hence Proved

(5) ## $A_1, A_2 \ldots A_n$ — given matrices.
$A_1 \Rightarrow d_0 \times d_1$ (dimensions)
$A_2 \rightarrow d_1 \times d_2$
$A_3 \rightarrow d_2 \times d_3$ $\quad D = \{d_0, d_1, d_2, d_3, d_4\}$
$A_4 \rightarrow d_4 \times d_5$

Algo says that we will pick the largest Remaining dimension from the array of dimensions & multiply two adjacent Matrices which share that dimension we create an array order[] whose i defines i^{th} multiplication
Dimension D = (do d1 .. dn) of n Matrices.
We can do this using 2 methods. -

(1) 1st method :-
Find index of max element from array, assign it to order[] & remove that index from array & Recursively iterate over Remaining array

```
Find Order(D):
    if length(D) ≥ 3:
        k = Index of Max(D)
        D = (do, d1 .. dk-1, dk+1 .. dn)
        i = i+1
        Order[i] = k
        Find Order(D)
    return order
```

(2) 2nd method :-

This will take $O(n^2)$ complexity since for $O(n)$ recursive calls $O(n)$ comparisons are performed

(2) 2nd method :-

We can sort the dimensions initially such that making a pair (index, dimension) & sorting on dimension OR we can create a priority queue with (index, dimension) as one element & sorted in descending order using dimension. Both will take $n \log n$ complexity

Algo:

Sort (index, dimension) as a pair by descending order of dimension

Call this as array or vector of pairs sorted in descending order of 2nd value of pair.

Let that array = matrices

Iterate over all elements of matrices from i = 0 to n-1 (we don't to index

~~Dont~~

of n^{th} dimensions because they have no shared dimension)

order[i] = matrices[i].index

i = i+1

Return orders

Here we just assign indices of of dimensions in decreasing order of dimensions to order & Return.

∴ Sorting has $O(n \log n)$ complexity & we are iterating $O(n)$ for assigning indices to order $= O(n \log n + n)$

∴ Total complexity = $O(n \log n)$

(ii) This algo doesn't give optimal solution we can show that using counter example.

~~Let A, B~~

Let A → a x b

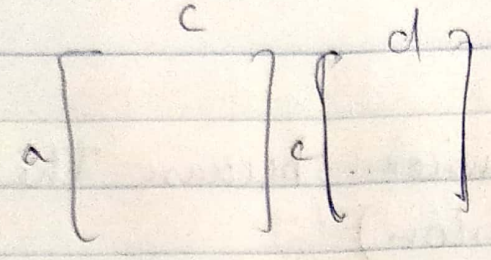
B → b x c

C → c x d

If we multiply (AB)C (in this order)

No. of multiplications = $\underbrace{a \cdot b \cdot c}_{(a \times c)}$ + $\underbrace{a \cdot c \cdot d}_{(a \times d)}$

abc

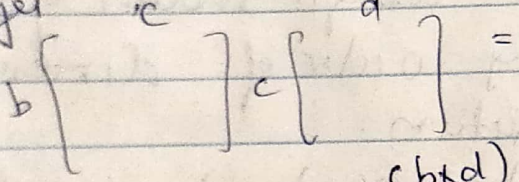


acd

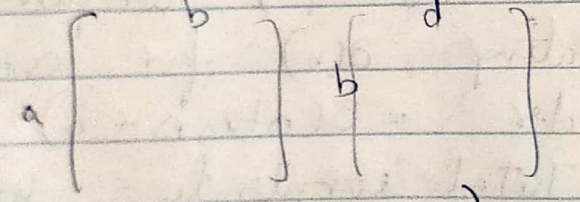
= abc + acd (1)

Similarly if we multiply A(BC)

we get



(bxd)



(axd)

= bcd + abd (2)

If we say b > c it means our order array will be

order[1] = 1

order[2] = 2

& order = (AB)C

which means since this is optimal abc + acd < bcd + abd from (2)

This should be true

Lets take values

$a = 4$
$d = 3$
$b = 5$
$c = 4$

for A(BC) we get

$abd + bcd = 60 + 60 = 120$

for (AB)C we get

$abc + acd = 80 + 48 = 128$

from above we see that the equality given in (2) doesn't hold

since,

$abd + bcd < abc + acd$.

∴ This doesn't give optimal sol^n & other sol^n exists that can be solved in less time

∴ The algo isn't efficient.