CS5800 Assignment 01

Answers –

1 .

The logic of the improvement algorithm is as follows –

If there exist unstable pairs (m, w) and (m′, w′) such that m prefers w′ over w and w′ prefers m over m′, then replace the pairs by (m, w′) and (m′, w).

Preferences of men (U, V, W) and women (A, B, C) is as follows–

A	B	C	U	V	W
U	W	U	B	A	A
W	U	V	A	B	B
V	V	W	C	C	C

Initial Pairs –

{(A, U), (B, V), (C, W)}

To Prove –

i) Choosing wrong unstable pair leads to cycles.

ii) 4 step cycle for given preference list

Proof –

i) While choosing the pair if we choose correct unstable pairs like (B, V), (C, W) to swap the partners, then in the end we get following pairs -

{(A, U), (B, W), (C, V)}

which leads to stable matching pairing.

However, if we choose wrong unstable pair like (A, U), (B, V) then the swapping leads to a loop that never gives stable match.

ii) 4 Step Cycle is as follows –

Initial Pairs - {(A, U), (B, V), (C, W)}

a Step 1

Unstable pair chosen: (A, U), (B, V)

Explanation: U prefers B over his current assigned partner A
and B prefers U over her assigned partner V, so this is an unstable pair and we can swap them.
Pairs formed after swap: {(A, V), (B, U), (C, W)}

b Step 2
Unstable pair chosen: (B, U), (C, W)
Explanation: First preference of B is W,
Thus, she prefers W over her assigned partner U.
Also, C is last preference of W that's why W prefers B more than his assigned partner. Hence, we can swap these unstable pairs.
Pairs formed after swap: {(A, V), (B, W), (C, U)}

c Step 3
Unstable pair chosen: (A, V), (B, W))
Explanation: First preference of W is A,
Thus, he prefers A over his assigned partner B.
Also, A is currently paired with her last preference V and that's why she prefers W more as he is her second preference. Thus, we can swap these unstable pairs.
Pairs formed after swap: {(A, W), (B, V), (C, U)}

d Step 4
Unstable pair chosen: (A, W), (C, U))
Explanation: First preference of A is U,
Who is currently paired with her second preference W.
Also, U is currently paired with his last preference C and his second preference A. Thus, we have A(woman) and U(man) who prefers each other over their assigned partners and hence we can swap these unstable pairs.
Pairs formed after swap: {(A, U), (B, V), (C, W)}

We can see that pair formed after step 4 is same as that of the initial state.

2 .

(Collaborator: Jhalak Maheshwari, Prasad Tajane, Sourabh Punja)

Given:

Two television networks A & B.

Each network has *n* prime time slots and *n* TV shows.

All TV shows are unique and every show has got rating.

The highest rated TV show wins the time slot.

For simplicity will consider that each show has unique rating.

Stable Schedule:

Changes in the schedule does not result in the changes of time slot one by a TV network.

Prove:

Is there a stable pair of schedules?

Answer:

We cannot form a stable schedule until unless ratings of all TV shows of one network is greater than the other.

Will prove this by giving an example (part b).

Proof:

Let's assume that value of *n* is 3.

Network A has A1, A2, A3 TV shows.

Network B has B1, B2, B3 TV shows.

Ratings of the shows is as follows –

TV Show	***Rating***
A1	1
A2	4
A3	6
B1	2
B2	3
B3	5

Initial schedule of network A = S as {A1, A2, A3}

Initial schedule of network B = T as {B1, B2, B3}

Win Results of the initial schedule are as follows –

TV Slot	*Winner*
1	B1
2	A2
3	A3

Thus, at the end network A wins 2 slots while network B managed to win only 1 slot.

Now suppose if we change the schedule of network B to {B1, B3, B2} without changing schedule of network A, then the results would change.
Schedule of A = S is {A1, A2, A3}
Schedule of B = T is {B1, B3, B2}

Win Results of the new schedule are as follows –

TV Slot	*Winner*
1	B1
2	B2
3	A3

This results in network A winning only 1 slot whereas B wins 2.

Similarly, if we change slots of shows A2, A3 and keep the schedule B unchanged then again, the final results will be different as compared to the initial results.

Since neither network can unilaterally change its own schedule without changing the number of time slots win by the network. Thus, this schedule is an unstable schedule.

Finally, from this example we can say that, analogues to the output of GS stable matching algorithm, generation of stable schedule is not possible.

3 .

Arrange the following functions in order from the slowest growing function to the fastest growing function.

$$\sqrt{n}, n\sqrt{logn}, 2^{\sqrt{\log n}}, (\log n)^2$$

Answer: -

$$(log(n))^2 \quad < \quad 2^{\sqrt{\log(n)}} \quad < \quad \sqrt{n} \quad < \quad n\sqrt{\log(n)}$$

Proof: -

a> Let's start with $\sqrt{n}, n\sqrt{logn}$ we know that $\sqrt{n} < c(n)$, as square root of any positive number will always be smaller than the number. Thus we can say

$$\sqrt{n} < c1(n\sqrt{\log(n)})$$

where there exists a $c1 > 0$ such that for all $n > n_o$, ($n_o > 2$) above condition evaluates to true.

(Finally, we can substitute the original values and say that)

$$\underline{\sqrt{n} \qquad < \qquad n\sqrt{\log(n)}}$$

b> Now let's compare $\sqrt{n}$, $2^{\sqrt{\log n}}$ (2 to the power square root of logn)

$$\sqrt{n} \qquad ? \qquad 2^{\sqrt{\log n}}$$

(For any variable $x = 2^k$ we can write the x as $2^k = 2^{log(x)}$, $\sqrt{n} = 2^{\log(\mathrm{sqrt(n)})}$)

$$2^{\log(\mathrm{sqrt(n)})} \qquad ? \qquad 2^{\sqrt{\log n}}$$

(we can do this comparison by comparing only the exponent part)

$$log(sqrt(n)) \qquad ? \qquad \sqrt{log(n)}$$

(using log(na) = a*log(n))

$$(1/2)*log(n) \qquad ? \qquad \sqrt{log(n)}$$

(now squaring on both the sides)

$$(1/4)*(log(n))^2 \qquad ? \qquad log(n)$$

(As square of any number greater than 1 would always be greater than the original number)

(There exists a constant c`> 0 such that for all n > no following value will hold True)

$$c`*(log(n))^2 \qquad > \qquad log(n)$$

(Finally, we can substitute the original values and say that)

$$\underline{\sqrt{n} \qquad > \qquad 2^{\sqrt{\log n}}}$$

c> Now let's compare $(log(n))^2$, $2^{\sqrt{\log n}}$

$$\sqrt{n} \quad ? \quad 2\sqrt{\log(n)}$$

(For any variable $x = 2^k$ we can write the x as $2^k = 2^{log(x)}$, $(\log(n))^2 = 2^{\mathbf{2log(log(n))}}$

$$2^{\mathbf{2log(log(n))}} \quad ? \quad 2\sqrt{\log(n)}$$

(we can do this comparison by comparing only the exponent part)

$$2*log(log(n)) \quad ? \quad \sqrt{log(n)}$$

(now squaring on both the sides)

$$4*[log(log(n))]^2 \quad ? \quad log(n)$$

(Taking lim $_{n\text{-> Infinity}}$ on both sides)

$$\lim_{n \to \text{Infinity}} 4*[log(log(n))]^2 \quad ? \quad \lim_{n \to \text{Infinity}} log(n)$$

(Taking Differentiation on both the sides, since it does not change the signage)

$$d/dx\{\lim_{n \to \text{Infinity}} 4*[log(log(n))]^2\} \quad ? \quad d/dx\ \{\lim_{n \to \text{Infinity}} log(n)\}$$

(after differentiation)

$$\lim_{n \to \text{Infinity}} 8*[log(log(n))]/(\log(n)*n) \quad ? \quad \lim_{n \to \text{Infinity}} (1/n)$$

(Multiply both sides by {n*log(n)})

$$\lim_{n \to \text{Infinity}} 8*[log(log(n))] \quad ? \quad \lim_{n \to \text{Infinity}} \log(n)$$

(Taking Differentiation on both the sides, since it does not change the signage)

$$\lim_{n \to \text{Infinity}} 8*[1/n(log(n))] \quad ? \quad \lim_{n \to \text{Infinity}} (1/n)$$

(Multiply both sides by {n*log(n)})

$$\lim_{n \to \text{Infinity}} 8*[1] \quad ? \quad \lim_{n \to \text{Infinity}} (log(n))$$

(at infinity, rhs will go to infinity)

$$1 \quad ? \quad \infty$$

(Thus we can say that)

$$\lim_{n \to \text{Infinity}} 8*[1] \quad < \quad \lim_{n \to \text{Infinity}} [(log(n))]$$

(Finally, we can substitute the original values and say that)

$$\underline{(log(n))^2 \quad < \quad 2\sqrt{\log n}}$$

d> Now, from a, b, c we can say that,

$$(log(n))^2 \quad < \quad 2\sqrt{\log n}$$

$$\sqrt{n} \quad > \quad 2\sqrt{\log n}$$

$$\sqrt{n} \quad < \quad c1(n\sqrt{logn})$$

$$(log(n))^2 \quad < \quad 2\sqrt{\log(n)} \quad < \quad \sqrt{n} \quad < \quad n\sqrt{\log(n)}$$

4.

Given: Algorithm to solve 2D array creation problem.

```
For i = 1, 2, ..., n
  For j = i + 1, i + 2, ..., n
    Add up array entries A[i] through A[j]
    Store the result in B[i, j]
  Endfor
Endfor
```

(a) Running Time of the given algorithm.

As algorithm does not mention anything about how to add array entries (line 3), will consider that this is done using another for loop. The storing solution can be done in constant time, So, in the general worst case (B[i][n]) the innermost for loop will be executed *n* times. Whereas, for complete calculation of B[1][j] (B[1][1]…B[1][n]) the second for loop will be executed *n-1* times. And it is pretty obvious that outer for loop will be executed *n* times in general (B[1][n]…B[n][k]).

Therefore, by the time last value is calculated, all the for loops would be executed *n* times. Hence, running time of this algorithm is

$$n * (n-1) * n = n^2*(n-1) = n^3 - n^2$$

Finally, we can say that the running time curve/function of the algorithm is $O(n^3)$, since from the above proof we know that, there exists a constant $c > 0$ such that for all $n > n_o$, the runtime time function is less than or equal to $c* n^3$.

(b) Asymptotically tighter bound.

We know that outer for loop will be executed *n* times however to calculate the time taken by the innermost loop, let's divide the array into 8 small chunks (*part a1 through a8*).

i) When i is in a1 and j is in a8.

In this case, innermost loop will be executed at least *3n/4* times while calculating the addition of all elements from I to j. And there will be $(n/8)*(n/8)=n^2/64$ such cases.

Thus total executions will be at most
$(n^2/64)*(3n/4)=3n^3/256$

ii) When i is in a2 and j is in a7.

In this case, innermost loop will be executed at least $n/2$ times while calculating the addition of all elements from I to j. And there will be $(n/8)*(n/8)=n^2/64$ such cases.

Thus total executions will be at most $(n^2/64)*(n/2)=n^3/128$

iii) When i is in a3 and j is in a6.

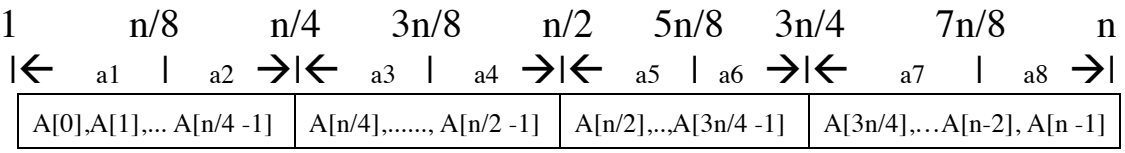
In this case, innermost loop will be executed at least $n/4$ times while calculating the addition of all elements from I to j. And there will be $(n/8)*(n/8)=n^2/64$ such cases.

Thus total executions will be at most $(n^2/64)*(n/4)=n^3/256$

iv) When i is in a4 and j is in a5.

In this case, innermost loop will be executed at least $n/8$ times while calculating the addition of all elements from I to j. And there will be $(n/8)*(n/8)=n^2/64$ such cases.

Thus total executions will be at most $(n^2/64)*(n/8)=n^3/512$



Thus, the total execution will be at most addition of all of them, which would be $3n^3/256 + n^3/128 + n^3/256 + n^3/512 = 13n^3/512$. Hence, running time of the given algorithm will be at most $13n^3/512$. And hence we can say that there exists a constant $c > 0$ (c should be in between 0 and 13/512) such that for all values of $n>n_o$, $13n^3/512 >= c*n^3$

Finally, we can say that $13n^3/512 = \Omega(n^3)$

From the proof of 5(a) and 5(b) we can say that runtime function of the algorithm is $\theta(n3)$.

(c) Improvement in the algorithm.

Running time of the algorithm can be improved by improvising the technique used for the addition of the elements 1 through n for calculation of term B[1][j] (line 3 of the algorithm).

Instead of using a for loop we can save the addition of the previous term and then use that in the calculation of next term. Please refer below code.

```
For i=1, 2,…n
        sum = A[i]
        For j=i+1,i+2,…n
                sum = sum + A[j]
                Store the result in B[i][j]
        Endfor
Endfor
```

Running Time of the improved algorithm:

Like the original algorithm first for loop of the algorithm will run *n* times while calculating each row of the 2D matrix (B[1][j]…B[n][j]).

And the second for loop will be executed *n-1* times when calculating values of first row (B[1][1]…B[1][n]).

Finally, the addition can be done in constant time thus, in the worst-case algorithm would take $n * n * 1 = n^2$ time.

Hence, we can say that the running time curve/function of the algorithm is $O(n^2)$, since from the above proof we know that, there exists a constant $c` > 0$ such that for all $n` > n_o$, the runtime time function is less than or equal to $c` * n`^2$.

5.

(a) Prove If $f(n) = \Omega(h(n))$ and $g(n) = O(h(n))$, then $f(n) = \Omega(g(n))$.

Proof:

$f(n) = \Omega(h(n))$ thus we can say $f(n) >= c_1 * h(n)$ $(c_1>1)$ …(1)

$g(n) = O(h(n))$ thus we can say $g(n) =< c_2 * h(n)$ $(c_2>1)$ …(2)

$f(n) >= c_1 * h(n)$ and $g(n) =< c_2 * h(n)$

multiply equ1 by $(1/ c_1)$ on both sides and by multiply equ2 by $(1/c_2)$ on both sides

$(1/ c_1) * f(n) >= h(n)$ and $(1/ c_2) * g(n) =< h(n)$

from above state we can come up with following state

$$(1/ c_1) * f(n) >= h(n) >= (1/ c_2) * g(n)$$

now we can remove the middle h(n) term to come up with following equation

$$(1/ c_1) * f(n) >= (1/ c_2) * g(n)$$

multiply above equ by (c_1) on both the sides

$$f(n) >= (c_1 / c_2) * g(n) \quad \ldots(3)$$

From the equation 3 we can say that there exists a $c` > 0$ such that for all values of $n > n_o$, $f(n) >= c` * h(n)$ holds true.

Thus, we can say that, $f(n) = \Omega(g(n))$

(b) Prove If $f(n) = O(g(n))$, then $f(n)^2$ is $O(g(n)^2)$.

Proof:

$f(n) = O(g(n))$ thus we can say $f(n) =< c_1 * g(n)$ $(c_1>1)$ …(1)

As f(n) and g(n) are runtime functions they cannot be negative

Therefore, on squaring them will not change the equality

$$[f(n)]^2 \quad <= \quad [c_1 * g(n)]^2$$

$$f(n)^2 \quad <= \quad c_1^2 * g(n)^2$$

As C1 is a constant whose value is greater than zero,

its square will also be a constant which would be greater than zero.

Thus, we can replace it by another constant C2 > 1

$$f(n)^2 \quad <= \quad c_2 * g(n)^2 \quad \ldots(2)$$

Here we can say that there exists a constant $C_2 > 0$ such that for all values of $n > n_o$ above equation holds true.

Thus, from equation 2, we can say that, $f(n)^2 = O(g(n)^2)$