HW-3

1.

(a)$T(n) = 4T(n/2)+n^2\log n$

We start with computing $c = \log_b a$
$= \log_2 4$
$= 2$

Here $f(n) = n^2 \log n$ and $n^c = n^2$
We have $f(n) = \theta (n^c \log n)$. k=1
This is case 2. so applying the generalized Masters theorem, $T(n) = \theta (f(n^c \log^{k+1} n)) =$ **$\theta (n^2 \log^2 n)$** for some $\varepsilon > 0$.

(b)T(n)=8T(n/6)+nlogn.
Here, $n^c = n^{\log_6 8}$ and $f(n) = n\log n = O(n^{\log_6 8 - \varepsilon})$ for some $\varepsilon > 0$.
This is case 1.So, by applying Masters Theorem, we get $T(n) = \theta (n^c) =$ **$\theta (n^{\log_6 8})$** for some $\varepsilon > 0$.

(c) $T(n) = \sqrt{6006}T(n/2)+ n^{\sqrt{6006}}$

We have $c = \log_2\sqrt{6006} = 0.5 \log_2 6006 = 6.28$ (approx..)
We have $n^c = n^{6.28}$
$f(n) = n^{\sqrt{6006}} = n^{77}$ (approx.) = **$\Omega(n^{6.28+\varepsilon})$** for some $\varepsilon > 0$.

This is case 3.
Check the regularity condition:

$a\, f(n/b) \leq k\, f(n)$
- ⇨ $\sqrt{6006}\, f(n/2) \leq k\, f(n)$ for some k<1.
- ⇨ $\sqrt{6006}\ (n/2)^{\sqrt{6006}} \leq k\ n^{\sqrt{6006}}$
- ⇨ $\sqrt{6006}\ (\frac{n^{\sqrt{6006}}}{2^{\sqrt{6006}}\, n^{\sqrt{6006}}}) \leq k$
- ⇨ For $k = \frac{\sqrt{6006}}{2^{\sqrt{6006}}}$, $k < 1$. This satisfies our regularity condition.

Applying Master's Theorem, we get $T(n)= \theta (f(n))=$ ***$\theta (n^{\sqrt{6006}})$*** for some $\varepsilon > 0$.

(d) $T(n)=10T(n/2) + 2^n$

We have $c = \log_b a = \log_2 10 = 3.32$
$f(n) = 2^n = \Omega(n^{3.32+\varepsilon})$, for some $\varepsilon > 0$.
This is case 3.
Check for regularity condition.
$a\, f(n/b) \leq k\, f(n)$ for some k<1
- ⇨ $10\, f(n/2) \leq k\, f(n)$
- ⇨ $10\, 2^{n/2} \leq k\, 2^n$
- ⇨ $10\, (2^n/2^{n/2}) \leq k\, 2^n$

We have, $k = 10/2^{n/2}$ which for larger n will be <1. Regularity condition is satisfied.

By applying master theorem, we get $T(n)= \theta (f(n))=$ **$\theta (2^n)$** for some $\varepsilon > 0$..

(e) $T(n)=2T(\sqrt{n})+\log_2 n$

Consider $n = 2^m$, then $\sqrt{n} = \sqrt{2^m} = 2^{m/2}$
$T(n) = 2T2^{m/2} + \log_2 2^m$
$= 2T2^{m/2} + m$
$S(m) = T(2^m) = 2S(m/2) + m$
Here , $f(m) = m$ and $m^c = m$
This is case 2. Applying master theorem, we get $T(n) = m \log m = \boldsymbol{\theta(\log_2 n \log \log_2 n)}$

(f) $T(n)=T(n/2)-n+10$

Here, f(n) is -ve for n>10. Hence we **can not apply** master theorem.

(g) $T(n) = 2^n T(n/2) + n$

A is a changing function, it is not constant. Hence we **can not apply** master theorem.

(h) $T(n) = 2T(n/4) + n^{0.51}$

$C = \log_4 2 = 0.5$
We have, $n^c = n^{0.5}$ and $f(n) = n^{0.51}$
$f(n) = \Omega(n^{0.5+\varepsilon})$ for some $\varepsilon > 0$. This is case 3.
Check for the regularity condition:

$a\, f(n/b) \leq k\, f(n)$ for some $k<1$
⇨ $2\, f(n/4) \leq k\, n^{0.51}$
⇨ $2 . (n/4)^{0.51} \leq k\, n^{0.51}$
For $k = 2/ 4^{0.51} = 0.98$.
We have k<1. Therefore regularity condition is satisfied.
By applying master theorem, we get $T(n) = \theta(f(n)) = \boldsymbol{\theta(n^{0.51})}$

(i) $T(n) = 0.5T(n/2) + 1/n$

Here a<1. Therefore, we can not apply master theorem.

(j) $T(n) = 16T(n/4) + n!$
$C = 2$ and $n^c = n^2$. We have $f(n) = n!$.
$f(n) = \Omega(n^{0.5+\varepsilon})$ for some $\varepsilon > 0$. This is case 3.
Check for the regularity condition:
$a\, f(n/b) \leq k\, f(n)$ for some $k<1$
⇨ $16\, f(n/4) \leq k . f(n)$
⇨ $16\, (n/4)! \leq k . n!$
⇨ $16\, (n!/4!) \leq k. n!$
For $k = 16/4! = 0.667$, $k < 1$. Satisfy the regularity condition.
This is case 3. By applying master theorem, we get $T(n) = \theta(f(n)) = \boldsymbol{\theta(n!)}$

2.
We prove the existence of a local minimum by induction.

Base case : For n = 3, we have A[1] ≥ A[2] and A[3] ≥ A[2] as stated in the question. Hence, A[2] is a local minimum.

Inductive Hypothesis: Let A[1 : n] has a local minimum for n = k. We will have to prove that A[1 : k + 1] also has a local minimum.

Inductive Step: If we assume that A[2] < A[3] then A[2] is a local minimum for A[1 : k + 1] since A[1] ≥ A[2] by the question.

Now let's assume that A[2] > A[3]. In this case, the array of k length A[2 : k + 1] = A'[1 : k] satisfies the induction hypothesis.
A'[1] = A[2] ≥ A[3] = A'[2]
A'[K-1] = A[K] ≤ A[K+1] = A'[K]
So, A'[1] ≥ A'[2]
And A'[K-1] ≤ A'[K] .

Therefore, A'[1:K] has a local minima. And as A[2 : k + 1] = A'[1 : k] , A[2 : k + 1], And as A[2 : k + 1] also has a local minima.

Consider the following algorithm with array A of length n as the input, and return value as a local minimum element.

Algo
If n = 3, return A[2]
If n > 3,
k ← ⌊n/2⌋.
If A[k] ≤ A[k1] and A[k] ≤ A[k + 1] then return A[k].
If A[k] ≥ A[k-1] then call the algorithm recursively on A[1 : k]
else call the algorithm recursively on A[k : n].

Complexity: The algorithm gives us the recurrence T(n) = T(⌈n/2⌉) + θ (1).
Here, $n^c = 1$ and f(n) = 1.This is case2. so applying the generalized Masters theorem,
T (n) = θ (log n). for some ε > 0.

3.

This we can solve using Dynamic Programming.

Subproblem : Let OPT[m,p] , be the minimum of the total time taken to travel the cities when Marco starts at 'm' and Polo starts at 'p' city.

Base case: OPT[m,p] = 0 when m=n-1 and p=n
When m < p and m != p-1, then m to p-1 cities should be travelled by Marco as Polo can not traverse back.
We have $OPT[m,p] = T[m,m+1] + T[m+1,m+2]+....+T[p-2,p-1] + OPT[p-1,p]$
$= d[m] - d[p] + OPT[p-1,p]$ where $d[m] = \sum_{k=m}^{n-1} T_k , T_{k+1}$

Let us consider Marco is at one city behind Polo, i.e Marco at the city "p-1" and Polo at city "p".
To calculate OPT[p-1,p], we should consider 4 cases.

1: Marco visits from p-1 to j+1 and Polo visit from p to k where k=p+2, …., n
OPT[p-1,p] = min (OPT[p+1, k] + $T_{p-1,p+1}$ + $T_{p,k}$) where k > p+1

2: Polo visit from p -> p+1 and Marco visit from p-1 -> k
OPT[p-1,p] = mink ($T_{p,p+1}$ + $T_{p-1,k}$ + OPT[k, p+1]) where k > p+1

Case 3: Marco visits "p-1" and the rest of the cities are travelled by Polo.
OPT[p-1,p] = d[p]

Case 4: Polo visits city "p" and the rest of the cities are travelled by Marco.
OPT[p-1,p] = $T_{p-1,\ p+1}$ + d[p+1]

return OPT[1,p] where p is from 2 to n.

ALGO:
For m=n-2 to 1:
  For p=n to i+1:
    OPT[p-1, j] = min (min (OPT[p+1, k] + $T_{p-1,p+1}$ + $T_{p,k}$), min ($T_{p,p+1}$ + $T_{p-1,k}$ + OPT[p+1, k]),
                d[p],
                ($T_{p-1,\ p+1}$ + d[j+1]))

    OPT[m,p] = T[m,m+1] + T[m+1,m+2] + …. + T[p-2, p-1] + OPT[p-1, p]

return OPT[1,p] where 2<=p<=n

Time Complexity: For calculating d[p], it takes linear time. As we have 2 for loops, that will give us $O(n^2)$. And for case m< p-1, it will take $O(n^2)$ time. So total time = $O(n+ n^2+ n^2) = O(n^2)$

4.

Let a be the array of days starting from 1 to n days. And b be the array having optimal number of lectures that Erica can take. K is the minimum number of lectures she should finish in every consecutive days.

We solve the problem using Dynamic Programming.

Sub problem : Let b[i]. be the optimal number of lectures that Erica can take on i^{th} day.

Base case: i =1
a[1] = b[1]

Our recurrence relation will be :
b[i] = max (a[i], k-b[i-1])

Algorithm :
Loop from i=2 to n
b[i] = max (a[i], k-b[i-1])
return b

Complexity : Complexity of the algorithm will be O(n) as it traverse until n one time.

5.

Let a be the array of n positive integers and a[i] be the points obtained at i^{th} step.

Subproblem : Let opt[i] be the optimum points when starts the game at position i.
We will be moving from right to left.

Recurrence: *OPT[i] = a[i] + max OPT[j] ; where j >= i + a[i]*

Base case : when we are at n^{th} position , *OPT[n] = a[n]*

This takes $O(n^2)$ as we iterate i times and finding OPT[j] also takes linear time, thus making total complexity as $O(n^2)$.

The complexity can be reduced by using memorization which takes constant time to find OPT[j], thus making total complexity as O(n).

Our recurrence will be *OPT[i] = a[i] +b[i] ; where b[i] = max(a[i] + OPT[j], OPT[i+1]) where j= i+ a[i]*

Our algorithm will be :

OPT[n] = a[n]
maximum = OPT[n]
Loop from n-1 to 1:
  OPT[i] = a[i] + max(OPT[j]) ; where j >= i + a[i]
  If maximum < OPT[i]
    maximum = OPT[i]

Return maximum

Complexity : we use memorization to find b[i] and the array is travelled only once. Hence, the time complexity is *O(n).*

6.

Let a and b the two strings that the Joseph has received as gift and having the length n. We will first compare both the strings if they match. We return TRUE, if they match. Else we check if it is even, then divide the strings into half and compare to satisfy the given 2 cases. If it is not even, then we return false.

We follow divide and conquer approach here.

We have to consider two things:

- Don't split when the string length is odd.
- Merge the string BLOCKWISE lexographically.

Our Algorithm would be:

If a is J-similar to b
  Return True
Else:
 If even
  ((a1 is J-similar to b1) and (a2 is J-similar to b2)) OR

((a2 is J-similar to b1) and (a1 is J-similar to b2))

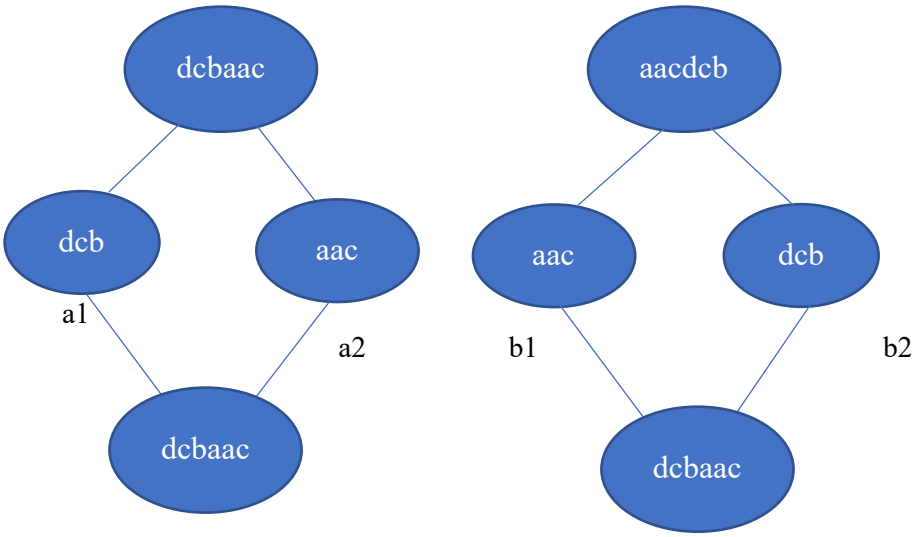
else
  return False

Our recurrence relation will be:

$T(n) = 4T(n/2) + O(n)$
We have $n^c = n^2$ and $f(n) = n$
This is case 1. so applying the generalized Masters theorem, $T(n) = \theta(f(n^c)) = \theta(n^2)$ for some $\varepsilon > 0$.

Ex1: a = abcde
b = abcde
Our algorithm returns True.

Ex2: a = dcbaac
b = aacdcb
Our algorithm returns True as a1 = b2 and a2 = b1 here.



7.

Let *p* be the array of n bits.– Every time we invert the bits and add, we get the compliment of p. The finally formed array will be say S will be of the form,

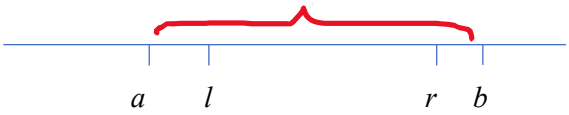
$$S = p\bar{p}\bar{p}p\bar{p}pp\bar{p} \ldots.$$

As $\bar{p}$ is inverted array of p, We can write, $sum(p) + sum(\bar{p}) = n$, where n is the total number of bits in our input array.

Now, we have to find the sum between given points *a* and *b*.
We are considering each *p* and $\bar{p}$ together one block and are given block index *k*.

Now, we will choose an arbitrary point *'l'*, which is the first point from *a* whose index is a multiple of 2n. Index of our arbitrary point *'l'* can be written as n. 2^k where n is length of *p* and k is the index of that block.

Similarly, we are choosing an arbitrary point *'r'*, which is the first point from *b* whose index is a multiple of 2n. Index of our arbitrary point *'r'* can be written as n. 2^k where n is length of p and k is the index of the block.



We can find the sum from *l* to *r* in constant time.
If *l* is at the index *n.* 2^k, then sum till l is given by ***nk*** and similarly we can find the sum till *r* from 1.

sum(*l* to r) = *sum(1 to r) – sum (1 to l)*

Now we have to find the sum(*a* to *l*) and sum (*r* to *b*). To find the value at each index, we will have to use divide and conquer to individually find whether 1 is present or 0 is present at that particular index.
To find the value at any particular index, we do make use of the relation.

$$valueAt[i] = valueAt[\, i - 2^{\log_2 i}].$$

sum(a to b) = sum(a to l) + sum(l to r) + (r to b)

where, ***sum(a to l)*** $= \sum_{i=a}^{l} valueAt[i]$
sum(r to b) $= \sum_{i=a}^{r} valueAt[i]$
sum(l to r) = sum(1 to r) – sum (1 to l)

To find the value at the i^{th} index, our recurrence relation would be:

if i =0 , value[i] = P,
if i =1, value[i] = $\bar{p}$
if i>=2, value[i] = value[i-$2^{\log_2 i}$ *]*

value(i,p):

- *count = 0*
- *while i>= 2;*
  count = count +1
  i = i-$2^{\log_2 i}$

if count is odd:
return p
Else
return $\bar{p}$

Complexity:

Sum from *l to r* takes constant time. Finding the value at index i, is taking *O(logb).* Thus, over all complexity is ***O(log b).***