Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

1. Given an unsorted integer array, find all pairs with a given difference k in it without using any extra space (sorting is allowed).

arr = [1, 5, 2, 2, 2, 5, 5, 4]   k = 0

Output: (2, 5) and (1, 4)

1, 2, 2, 5, 5, 6

Algorithm:

```
● Initialize i and j pointer, where i = 0, j = 1
● Sort the Array using heap sort
● While i and j are less then the size of arr:
    ○ If arr[j] - arr[i] == k:
        ■ If arr[i] != arr[i -1] and arr[j] != arr[j -1]
            ● Print (arr[i], arr[j])
        ■ i += 1
        ■ j += 1
    ○ Else if arr[j] - arr[i] < k:
        ■ j += 1
    ○ Else:
        ■ i += 1
```

Proof:

- Assume we miss a pair of numbers with difference of k
- After we sort the array, we initialize the pointers as i = 0, and j = 1 to compare elements arr[i] and arr[j]
- If we miss a pair, it means that our condition doesn't work for all the numbers

- However, our if condition **arr[j] - arr[i]** == **k,** ensures that every element gets inspected since we are starting with index i = 0 and j = 1, and test out to all of them until we reach to the last element to see if **arr[j] - arr[i]** result will be equal to **k.**
- Therefore, it means that we can't miss any pair of elements that have a difference equal to k. Contradicting our initial statement saying that we will miss a pair.

Time Complexity:

- Sorting using heap sort takes O(n log n)
- While loop takes O(n), it keeps going while it fits the condition
- Checking the if and else conditions takes O(1) time
- Incrementing the index takes O(1) time
- Overall it takes O(n log n) time

Fulati Aizihaer **Discussion:** 1F

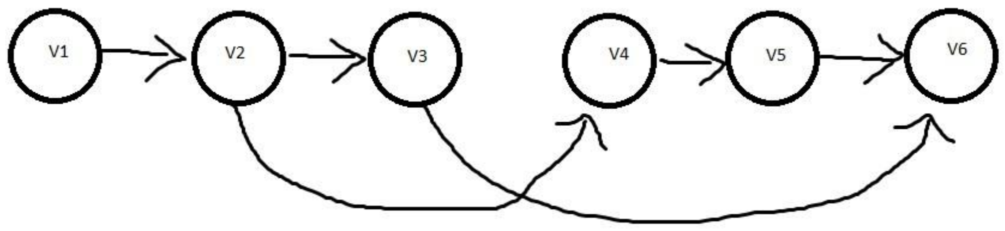
Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

2.Exercise 3 on page 314

PART A:

- The given algorithm doesn't work correctly, because it uses greedy methods and always tries to get the smallest j possible.
- One example this wouldn't work is in the graph below



- Based on the algorithm, we start at V1, and check for the node that is as small as possible which is V2.
- We then go to V2, and chose the next smallest node possible, which is V3
- We then go to V3, and chose the only possible node, which is the end V6
- Using this algorithm, we get longest path as L = 3
- However, by looking at the graph, we can see that if we take the path from V2 to V4 instead of V3, we can achieve a longer path of L = 4.
- This proves that the algorithm given doesn't work.

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

PART B:

Algorithm:

- Set an array **opt[1 … n],** where **opt[i]** holds the longest path length from V1 to Vi
- Set **opt[1] = 0** and set rest of the M[i] as **- infinity**
- For **i** from 1 to **n**:
    - For all the vertices **j** connected to vertex **i** by an edge (Vi, Vj)
        - If **opt**[i] not equal to **- infinity**:
            - **opt**[j] = max(1 + **opt**[i], **opt**[j]):
- Return **opt**[n] as the longest path from V1 to Vn

Proof:

- Given graph above from part a), n = 6
- We start the algorithm by looking at V1, and since its distance from itself is 0, we get **opt**[1] is not equal to **- infinity**
- We then look at its connected vertex V2, and assign **opt**[2] = 1, since 1 + **opt**[1] > - infinity
- We then go to V2 and look at all of its connected vertices.
    - We do **opt**[3] = 1 + **opt**[2], which is = 2
    - We do **opt**[4] = 1 + **opt**[2], which is = 2
- We continue this way and get the following:
    - V3's connected vertices, **opt**[6] = 1 + **opt**[3] = 3
        - Although we reached the end, we didn't go through all of the vertices, so we continued.
    - V4 → **opt**[5] = 1 + **opt**[4] = 3

- ○ V5 → we assign **opt**[6] to 1 + **opt**[5] = 4 or to its previous value, which is 3
    - ■ Since 4 > 3, we assign **opt**[6] = 4
- ○ Now that we have finished, we can conclude that the longest path from V1 to Vn is **opt**[6] = 4.

Proof by induction:

- We have a base case **opt[1] = 0**
- Inductive step is that at i you go through all the j connected to i, and you pick the longest path from **opt**[j] = max(1 + **opt**[i], **opt**[j]), which is the longest path we can get,therefore my recurrence relation is correct

----

Time Complexity:

- The first loop of going through all the elements in **opt** takes O(n) time
- The second loop of going through all the vertices **j** connected to a vertex i takes O(n) time
- Array lookups, computing maxes and addition takes O(1) time
- Overall, since we have a nested for loop, we get O(n^2) time

----

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

3.Exercise 5 on page 316

Algorithm:

- Given a string **y[i],** where **i** is the index of each letter
- Set an array **opt[1 … n],** where **n** is the length of **y**, and **opt[i]** holds the numerical quality(x)
- Set **opt**[0] = 0
- For **i** in range of 1 to **n**:
  - For **j** in range of 1 to **i**:
    - **opt**[i] = max(quality(y[j] … y[i]) + **opt**[j - 1], **opt**[i])
- Return **opt**[n]

Proof:

- Assume we are given a string "meetateight"
- If we have 0 letters in the word, then it means that our quality value would be 0 as well.
- We then start the for loop with i = 1, and j = 1,
  - **opt**[1] = max value between quality("m") + **opt**[0], or **opt**[1], and since we don't have an **opt**[1] already, we will go with quality("m") + M[0].
- The condition for checking the max quality "quality(y[j] … y[i]) + **opt**[j - 1]" is designed specifically so that it checks each segment of the string, starting from index **j** to **i** and sees if it is the highest quality that we have gotten so far and if it is, it combines the previous segment where we stopped before j, and adds that quality value with the current one and assigned it to **opt**[i].
- If the quality of that segment isn't the highest, then we continue with the previous **opt**[i] value we got.

- For example, when we get to the segment “meet”, it is the highest quality we have seen compared to previous segments, so we add quality(“meet”) with **opt**[j-1], where **opt**[j-1] is **opt**[1-1] → **opt**[0], and **opt**[0] is = 0, so we have **opt**[i] = quality(“meet”) + 0.
- We continue doing this recurrence and keep getting the highest quality we can get from the segments and combine it with the previous segments we get.
- Ultimately, after we finish the loop, our **opt**[n] value, which is the last value in the array, will contain the highest quality we can get from that string which contains the highest quality possible segments of that string.

Proof by induction:

- We have a base case of **opt**[0] = 0
- The induction step is that at i, you try all the j from 1 to i, and you pick the best quality for the segments of words which will return the highest we can get, therefore my recurrence relation is correct.

Time Complexity:

- The first loop of going through all the indexes in the string y takes O(n) time
- The second loop of going through the substring from index 1 to index i from first loop takes O(n) time
- Array lookups, computing maxes and addition takes O(1) time
- Overall, since we have a nested for loop, we get O(n^2) time

----

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

4. Exercise 10 on Page 321

Part a):

- The given algorithm doesn't work when the given data is like below

	Min 1	Min 2
A	10	20
B	5	100

- The answer given by algorithm is: 10 + 20 = 30, where it chooses A both times
- Where the correct answer is 5 + 100 which is 105, which is choosing B both times

----

Part b):

Algorithm:

- Given **a[i]** and **b[i]** where each represents the value for machine **a** and **b** at time **i**
- Initialize **opt_a[n]** and **opt_b[n]** to hold values for the each machine for **1** to **n** minutes
- Set **opt_a[0]** = 0, **opt_b[0]** = 0
- For **i** in range from 1 to **n**:
    - **opt_a[i]** = **a[i]** + max(**opt_a[i-1]**, **opt_b[i-2]**)
    - **opt_b[i]** = **b[i]** + max(**opt_b[i-1]**, **opt_a[i-2]**)
- Return max(**opt_a[n]**, **opt_b[n]**)

Proof:

- With the given example from above
- Starting from i = 1:
    - Opt_a[1] = a[1] + max(0, opt_b[i-2]), → opt_a[1] = 10
    - Opt_b[1] = b[1] + max(0, opt_a[i-2]), → opt_b[1] = 5
- For i = 2:
    - Opt_a[2] = a[2] + max(10, 0), → opt_a[1] = 20 + 10 = 30
    - Opt_b[2] = b[2] + max(5, 0), → opt_b[1] = 100 + 5 = 105
- We return the bigger value between opt_a[n] and opt_b[n] which is 30 and 105, and return 105

Proof by induction:

- We have a base case of **opt_a[0]** = 0, **opt_b[0]** = 0
- Inductive steps are as we pick both machines and enter them into the opt_a and opt_b to get the max value for both.
- In the end we then return the max between the two opt, therefore my recurrence relation is correct

----

Time Complexity:

- Setting the **opt** values takes O(1) time
- Our for loop takes O(n) time
- Array lookups, computing maxes and addition takes O(1) time
- Overall, it takes O(n) time

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

5.Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if the length of the rod is 8 and the values of different pieces are given as follows, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6).

Length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

Algorithm:

- Given the array **p[ ]** of prices starting from index 0
- Initialize an array **opt**[0…n], where **n** is the length of the rod
- Set **opt**[0] = 0
- For **i** in range of 1 to **n**:
    - For **j** in range of 0 to **i**:
        - **opt**[i] = max(**opt**[i], p[j] + **opt**[i - j - 1])
- Return **opt**[n]

Proof:

- We have a base case of **opt**[0] = 0
- We will initialize a new array **opt**[0 … n], where i in **opt**[i] are the different lengths and **opt**[i] = to the price of the length.
- We initialize the **opt**[0] = 0, because if we have a rod with 0 length, it means its price will be 0.

- The first loop to go over each possible lengths **i** of the rod from 1 to n
- The second loop goes over each possible lengths **j** from rod length of 0 to **i**
- **opt**[i] = max(**opt**[i], p[j] + **opt**[i - j - 1]) is a recurrence, where we use values from previous lengths.
    - **opt**[i] contains the price of the length i from the first loop
    - p[j] is the price in our given price array at position j
    - **opt**[i-j-1] gets the price of previous recurrence array element
    - We first check whether the previous assigned value of **opt**[i] is smaller than p[j] + **opt**[i - j - 1]:
        - If **opt**[i] is smaller, then we assign p[j] + **opt**[i - j - 1] as our new **opt**[i] value
- In the end, we return the last element in **opt** which is **opt**[n], which will ultimately contain the largest price possible

----

Time Complexity:

- First loop takes O(n) time to go through all the possible lengths of the rod
- Second loop takes O(n) time to go through all the possible lengths from lengths 1 till the length **i** of the first loop
- Array lookups, computing maxes and addition takes O(1) time
- Overall, it takes O(n^2) time

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

6. Consider a row of n coins of values v1 . . . vn, where n is even. We play a game against an opponent by alternating turns (you can both see all coins at all times) . In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can win if we move first, assuming your opponent is using an optimal strategy.

Example 1: [5, 3, 7, 10] : The user collects maximum value as 15(10 + 5) - Sometimes the greedy strategy works
Example 2. [8, 15, 3, 7] : The user collects maximum value as 22(7 + 15) -- In general the greedy strategy does not work

Algorithm:

- Given the array **c[ ]** of coins
- Initialize an array **opt**[n…0, 0…v], where **n** is number of coins, and **v** is the value of the coin
- For **i** in range from **n** to 1 (decrement):
    - For **j** in range of 1 to **n**:
        - **opt**[i, j] = max(c[i] + min(**opt**[i - 2, j], **opt**[i - 1 , j + 1]),
          c[j] + min(**opt**[i - 1, j + 1], **opt**[i , j + 2]))
- Return **opt**[1, v] for the maximum value

Proof:

- We will initialize a new array **opt**[n…0, 0…v] that will store the value of each coin from n to 0 and 0 to v.
- First loop will start from the back of the given array **c**[i]
- Second loop will start from the front of the given array **c**[j]
- Choosing the most optimal coin:

  - We can only choose the first coin c[j] or the last coin c[i] in array c[j … i]
  - We can't always pick the highest value coin from those two options, because there will be occurrences where picking a lower value coin might allow us to pick a higher value coin.
  - Our recurrence condition will contain j + 1 (first coin), j + 2 (second coin), i - 1 (last coin), i - 2 (second to last coin)
  - After we pick, our opponent will be using the same algorithm to pick the most optimal coin that gives the highest value, where then we will be left with a smaller value coin to pick.
  - Using the recurrence code in the algorithm, it ensures that we can still get the maximum value in the end after choosing a max value coin then a small value coin.
  - c[i] + min(**opt**[i - 2, j], **opt**[i - 1 , j + 1]) gives us the maximum value we can get when we pick the last coin first, then let the opponent pick, then we pick a smaller value either from the front or the end.
  - c[j] + min(**opt**[i - 1, j + 1], **opt**[i , j + 2]) gives us the maximum value we can get when we pick the first coin first, then let the opponent pick, then we pick a smaller value either from the front or the end.
  - We continue to do this through the loop until the end of the loop.
- In the end, we return the value stored at **opt**[1, v] which will contain the max value we can get

----

Time Complexity:

- Initializing an array with two parameters and settings values takes O(1) time
- The first for loop takes time of O(n) where n is the number of coins and we go in a reverse direction
- The second for loop takes time of O(n) and this time in normal direction from 1 to n

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

- Array lookups, computing maxes and addition takes O(1) time
- Overall, it takes O(n^2) time for the whole algorithm
