Fulati Aizihaer **Discussion:** 1F

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

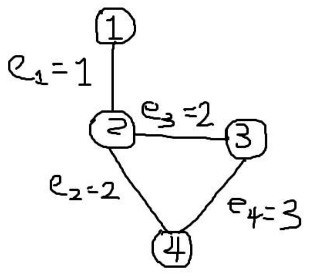
TA name: Parshan

1. Exercise 11 on page 193

Proof:

- Sort all the edges of **G**
- In order to get a specific minimum spanning tree **T** that contains a specific edge **e**, modify the edge ordering to put **e** before any other edges with the **same cost**.
- for all edges from lowest cost to highest cost, until an **MST** is found:
    - If adding the edge does not create a cycle, then we add it to **MST**
    - Else: do nothing
- By prioritizing **e** compared to other edges that have the same cost, while making sure that ordering of all the edges cost in the given graph is nondecreasing, we can produce any minimum spanning tree **T** that can be found in **G**
- However, you can't get two different MST using the same ordering of the edges (EXP below)
- Therefore, there exists a valid execution of Kruskal's Algorithm for any minimum spanning tree T, where you just need to have the valid ordering for the edge **e** you prioritize compared to the edges that share the same cost

EXP:



- We have two MST in this graph: T_1= (e1, e2, e4), T_2= (e1, e3, e4), where both have a total cost of 6.
- If we want T_1, then we can have the order of edges as such: e1, e2, e3, e4, with respective costs (1, 2, 2, 3), so it will chose (e1, e2, e4)
- If we want T_2, then we can have the order of edges as such: e1, e3, e2, e4, with respective costs (1, 2, 2, 3), so it will chose (e1, e3, e4)
- With valid ordering of edges, you can get all the MST in the graph G
- However, you can't get both T_1 and T_2 only using one ordering of (e1, e2, e3, e4) or (e1, e3, e2, e4)

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

2. Exercise 17 on page 197

Algorithm:

- Create an empty **maxJobs** list
- For each job in the list:
    - Create a **currentJobs** list
    - Arbitrarily select a job **n**, and set it as the our first job in the currentJobs list
    - Implement the regular interval scheduling algorithm to the **currentJobs** list
        - Take first job with earliest end time
        - Check the start time and only append it to the list if its start time is later than the last job's end time
    - If the number of jobs in **currentJobs** is greater than the number of jobs in **maxJobs**:
        - Replace **maxJobs** with **currentJobs**
- Return **maxJobs**

Proof by contradiction:

- Assume that there is a there is a optimal algorithm that completes more job than ours algorithm
- But since our greedy algorithm chooses the most optimal schedule out of all the possible optimal schedules, it means that our algorithm completes jobs >= jobs completed by the optimal algorithm.
- This contradicts our initial assumption that the optimal algorithm completes more jobs than our algorithm.
- Therefore, our algorithm is the most optimal algorithm

Time Complexity:

- Implementing the regular interval scheduling algorithm takes O(n) time
- Since we have to do this for each job in the list, it takes O(n) time
- Overall, it takes **O(n^2)** time to go through all the jobs and find the most optimal schedule by comparing each schedule.

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

3. Exercise 3 on Page 246

Algorithm:

- If there is **n** = 1 card
    - Return that card
- If there is **n** = 2 cards:
    - If card **a** is equal to card **b**:
        - Return either card **a** or **b**
- Partition the left half of the cards as **p1** and right half as **p2**
- Recursively call the algorithm for **p1**:
- If a card is returned:
    - Equivalence test that card with others to verify if it is majority equivalence or not
- If majority equivalence card is NOT found:
    - Recursively call the algorithm for **p2**:
    - If a card is returned:
        - Equivalence test that card with others to verify if it is majority equivalence or not
- Return the majority equivalence card if there is one

Proof:

- In order to find a majority equivalence card, we must have at least n/2 + 1 of that specific card
- In the algorithm above, we divide the cards into half and recursively do that until we are left with one or two cards.
- If we have one card left, check if that is a majority by equivalence testing to other cards, then return it if it is majority equivalent.
- If we have two cards left, then check if they are equivalent.
    - If they are, then return either card
    - Check if the returned card is a majority by equivalence testing to other cards, then return it if it is majority equivalent
- In the end, you will get the card that is a majority equivalent (n/2 + 1)

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

Time Complexity:

- The algorithm above halves the list size each recursion which takes O(log n)
- Checking if the returned card is really a majority takes O(n)
- Total time complexity is O(n log n)

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

4. Exercise 5 on page 248

Algorithm:

Number of lines = n, L_i are the lines in **visibleLines**

NOTE: The **first** and **last** line in the sorted list will always be visible since they are will be the steepest lines in each side

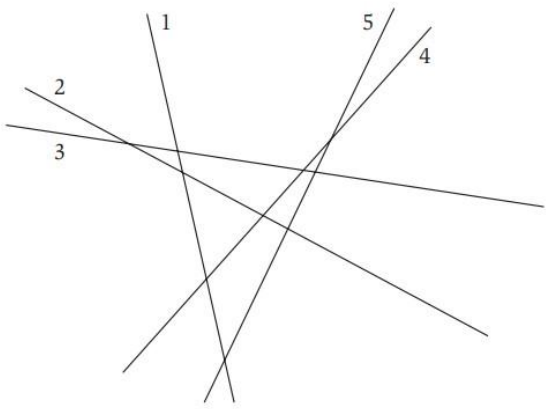
- Create a list that holds **visibleLines**
- Create a list to hold intersection points
- Sort the lines in increasing order of their slopes
- If **n** = 2:
    - If **L1** and **L2** have the same slope (**a**):
        - If **L1**'s y-intercept point (**b**) is > L2's **b**:
            - Append **L1** to **visibleLines**
        - Else:
            - Append **L2** to **visibleLines**
    - Else:
        - Append **L1** and **L2** to **visibleLines**
- If **n** = 3:
    - Append **L1** and **L2** lines to **visibleLines**
    - If **L2** crosses **L1** to the left of where **L3** crosses **L1**:
        - Append **L2** to **visibleLines**
- If **n** > 3:
    - Use divide and conquer method and partition the given list to **left** and **right** sub lists.
    - Recursively implement the algorithm starting from step 3 onto both the **left** and the **right** sub lists
    - Calculate the intersection points C_i for **left** list, between L_i and L_{i+1} in **left** list
        - If the C_{i+1}'s x-coordinate value < C_i's x-coordinate value:
            - Then C_i will be the intersection point between L_i and L_{i+2} instead
            - Remove L_{i+1} from the **visibleLines**
        - Append C_i to the **left** intersection points list

- Calculate the intersection point C_i between the **last** line in the **left** list and **first** line in the **right** list.
  - Append it as the last intersection point C_k on **left** list
- Calculate the intersection points C_i for **right** list, between L_i and L_{i+1} in **right** list
  - If the C_{i+1}'s x-coordinate value < C_i's x-coordinate value:
    - Then C_i will be the intersection point between L_i and L_{i+2} instead
    - Remove L_{i+1} from the **visibleLines**
  - Append C_i to the **right** intersection points list
- Merge the list of intersection points C_i from both **left,** and **right** sub lists recursively in a **combined** list that orders them by increasing x-coordinate

- Return the **visibleLines** list that contains the lines that are visible

----

Proof:



- Given the lines above
- First we sort the lines by slope **(1, 2, 3, 4, 5)**
- Since we have n > 3, we will divide the lines in half **recursively**. Making **(1, 2, 3)** and **(4, 5)**.
- For **(1, 2, 3)**, we will **only** append line **1** and **3** to **visibleLines**, since **2** doesn't fit the condition where it crosses line **1** to the left of line **3** crosses line **1**

- For **(4, 5)**, we append both to **visibleLines**
- Now that we are out of the recursion, we will begin to merge and test out "visible" conditions
- Since **2** is not in the **visibleLines** list, we get the intersection of line **1** and **3** (from the left list)
- Get the intersection of line **3** and **4** and append it to the end of left list intersection points
- Get intersection of line **4** and **5** (from the right list)

NOTE: The left of the point should be uppermost to the line with smaller slope, and the right of the point should be the line with bigger slope

- Now merge together the left and right list of intersection points
    - Keep doing it till you get one list with all the points
- Return the visibleLines list which will output **1, 3, 4, 5**
- Therefore, all the lines except for **2** are visible.

Time Complexity:

- Sorting the slopes take O(n log n) time
- We divide the problem by half each iteration, which takes O(log n) time
- Merging after we done takes O(n)
- The overall runtime of this algorithm is O (n log n)

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

5. Suppose you are given an array of **distinct** sorted integers that has been circularly shifted k positions to the right. For example taking (1 3 4 5 7) and circularly shifting it 2 positions to the right you get (5 7 1 3 4).
Design an efficient algorithm for finding K.

Algorithm:

- **n** = number of elements in the array
- **left** stores the index of the first element
- **right** stores the index of the last element
- If value of left = right:
    - Return **K = left**
- **mid** stores the mid index of the array: (length of array) / 2
- **NOTE:** if size of array is odd, round down, then plus 1 (EXP: 5 / 2 = 2, 2 + 1 = **3**)
- If left <= mid <= right:
    - It's sorted, so **K = left**
- If n = 2:
    - Return K = index of the smaller element
- Else If n = 3:
    - If value of **mid** is greater than value of **right**:
        - Return **K = right**
    - If value of **left** is greater than value of **mid**:
        - Return **K = mid**
- Else:
    - If value of **mid** is greater than value of **right** :
        - Make **left** = **mid**
        - Recursively implement the algorithm to the right half array
    - Else (value of **left** is greater than value of **mid**):
        - Make **right** = **mid**
        - Recursively implement the algorithm to the left half array

----

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

Proof:

- Shifted means that we look for the smallest element in the list and find its index since that shows how many times its been shifted (assuming index starts from 0)
- The algorithm above checks if the list is sorted or not
    - if it's not sorted, we cut the list in half
    - Get the half that is not sorted, since the other half will always be sorted.
    - Recursively implement our algorithm on the unsorted half we got from step above until we are left with only two or three elements
        - Then return K based on the condition of the if statements for either n = 2 or n = 3

EXP 1:
1,2,3,4,5,6,7,8,9 → 4,5,6,7,8,9,1,2,3 → (4,5,6,7) (8,9,1,2,3) → (8,9,1,2,3) is unsorted
8, 9,1,2,3 → (8,9,1) (2,3) → (8,9,1) is unsorted → 9 > 1 so return index of 1

EXP 2:
1,2,3,4,5,6,7,8,9 → 8,9,1,2,3,4,5,6,7 → (8,9,1,2,3) (4,5,6,7) → (8,9,1,2,3) is unsorted → (8,9,1) (2,3) → (8,9,1) is unsorted → 9 > 1, so return index of 1

Time Complexity:

- The algorithm above halves the list size each recursion, O(log n)
- Comparing if elements takes O(1)
- Returning the **index** value after comparison takes O(1)
- Overall the time complexity is **O(log n)**

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

6. Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the k-th position of the final sorted array.
Note that a linear time algorithm is trivial (and therefore we are not interested in).

Input : Array 1 - 2 3 6 7 9
Array 2 - 1 4 8 10
k = 5
Output : 6

Algorithm:

- **Receive the input arrays**
- **midA** stores the mid index of the array **m,** (length of m / 2)
- **midB** stores the mid index of the array **n** (length of n / 2)

NOTE: if size of array is odd, round down, then plus 1 (EXP: 5 / 2 = 2, 2 + 1 = **3**)

- If **k <= midA** + **midB**:
    - If there is only 2 elements left in **m** + **n** (one each):
        - If k = 2:
            - Return the larger number of the two elements
        - If k = 1:
            - Return the smaller number of the two elements
    - Recursively divide the left half of both arrays starting from their respective mid indexes
- Else (**k > midA** + **midB)**:
    - k = k - (midA + midB)
    - Recursively divide the right half of both arrays starting from their respective mid indexes

Fulati Aizihaer **Discussion:** 1F

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

TA name: Parshan

Proof:

- Assume we have **m = 2, 3, 6, 7, 9,** **n = 1, 4, 8, 10,** **k = 5**
- We first get the midpoint of both arrays where **midA = 3**, and **midB = 2**
- Since **k = 5 <= (3 + 2):**
- We have the new half arrays **m = (2, 3, 6)**, and **n = (1, 4)**
- Since there are still more than 2 elements left in **m + n** combined, we recursively divide again using our conditions
- **midA = 2, midB = 1**, since **k = 5 > 3**:
    - **k = 5 - 3 = 2**, recursively divide the right half, **m = (6)**, **n = (4)**,
- **k = 2 <= 2**:
- Since there are only two elements left **m + n** combined, we stop the recursion:
    - Since **k = 2**, we return the larger number between **6** and **4**
- Therefore, when **k = 5**, our element value is **6**

Time Complexity:

- The algorithm above halves the list size each recursion, O(log n)
- Comparing the last two element will take O(1) time
- Overall, the time complexity takes **O(log n)**
