Homework 4

Tejas Kamtam

305749402

Discussion 1B - Friday, 10 am

TA: Parshan

Problem 1

> Page 193, Exercise 11

Algorithm

- Given a graph G of e edges and n nodes, create a valid ordering of edges by sorting them in nondecreasing order of weight w_i
- For each edge e_i where $i \in |e|$, to each edge weight w_i of the sorted list of edges, add some constant, say $i \cdot \varepsilon$ where $\varepsilon = \frac{1}{e}$, such that for any edge in the sorted list of edges, it's new weight can be found as $w_i' = w_i + i\varepsilon$
- Run Kruskal's on this graph G using the new list of sorted edges, such that each edge has a unique weight, and produce the unique MST denoted T

Time Complexity Analysis

- The most costly operation in Kruskal's is the sorting of the edges
- Using our best sorting method (using a minheap or using merge sort), the complexity is $O(e \log e)$

- Adding the constant to each weight is $O(e)
- So, the overall time complexity of this modified Kruskal's is

$$O(e \log e)$$

Proof of Correctness by Direct Proof

- To prove this version of Kruskal's produces a unique MST, given that we have proved Kruskal's does indeed produce a valid MST in class, we must prove that the list of edges we pass to the algorithm is sorted and consist of unique edges (by weight)
- We can confirm that the ordering of this new list of edges is in fact in increasing order by looking at the constant we add: $i\varepsilon = i \cdot \frac{1}{e}$
- WLOG, consider the graph of 2 edges, initially sorted such that the list of edges (represented by their weights) is e_1, e_2
- Because the initial sorting is in nondecreasing order, $e_1 \leq e_2$
- Then, for each edge we assign it a new weight $w_i' = w_i + i \cdot \varepsilon$ where $w_i \in (e_1, e_2)$, $i \in (0, 1)$, and $\varepsilon = \frac{1}{2}$
- Comparing these new weights, e_1', e_2' we see that:

$$e_1' \leq e_2' \impliedby e_1 + i_0\varepsilon \leq e_2 + i_1\varepsilon$$

- In the worst case: $e_1 = e_2$, so we can see that:

$$e_1 + i_0\varepsilon \leq e_2 + i_1\varepsilon \implies i_0 \leq i_1$$

- Now, because the index of our list is defined to be strictly increasing for each weight in the graph, it holds that $i_0 < i_1$, therefore:

$$e_1 + i_0\varepsilon < e_2 + i_1\varepsilon \implies e_1' < e_2' \quad \forall e \in G$$

- Thus, not only is our new sorting of edges unique, every value is strictly $e_i < e_{i+1}$
- Thus, Kruskal's will produce a unique, valid MST, by the proof done in class

Problem 2

> Page 197, Exercise 17

Algorithm

- Pre-computation:
  - convert all start and end times to 24 hour time for correct comparison later on
- Create an empty list for the globally optimal schedule
- Given a set of intervals S, sort the intervals by start time
- Arbitrarily select one interval s_1 and ignore all of its overlapping intervals for the next step
- On the remaining intervals, run the regular interval scheduling algorithm to find a possible schedule containing s_1
- If this proposed schedule contains more intervals than the globally optimal schedule, update the global schedule
- Repeat these steps for each interval in the sorted list of intervals
- Return the final globally optimal schedule

Time Complexity Analysis

- Sorting the intervals is $O(n \log n)$
- Running the regular interval scheduling algo is $O(n)$

- Doing this for each sorted interval gives a total run time of

$$O(n^2)$$

Proof by Contradiction

- Suppose there exists an optimal solution produces a schedule with at least 1 more interval than our solution
- Suppose both solutions contain some interval $s_i \in S$ that must exist in the valid solution
- Neither solution contains any overlapping intervals of s_i by definition of a valid solution
- Because our solution runs the regular interval scheduling algorithm on the remaining choices of intervals after s_i, we must select at least as many intervals as the optimal solution does (given we both select s_i), by the proof of the regular interval scheduling problem done in class
- Therefore, if the optimal solution has more intervals than our solution, it must have selected some interval before s_i
- However, this contradicts the fact that our algorithm sorted by start time and selected the earliest interval possible in a valid solution; thus the optimal solution would be invalid if it selected more intervals than our solution

Problem 3

> Page 246, Exercise 3

Algorithm

- Given a set of n cards
- If `n==1`, return it as the majority
- If not, divide the set of n cards int to 2 halves, `a` and `b`
- Recursively do this for each half separately to find the majority of each division
- Compare majorities of each subdivision of each half until there is a majority for each half, say `maj_a` and `maj_b` and track the number of cards for each of the majorities present in the halves
- If `mah_a==maj_b`, then return it as the majority
- If not, check each `maj_a` with each card from half `b`
- If there are more than $n/2$ cards of `maj_a`, return `maj_a`
- If not, do the same with `maj_b` but with half `a`
- If neither has more than $n/2$, return no majority

Time Complexity Analysis

- This algorithm is divide and conquer
- Dividing the groups and recursively checking the majority for each half is $O(n \log n)$ total for both halves
- All further checks if the cases are not met is $O(n)$
- Thus, the overall time complexity in terms of equivalence calls is

$$O(n \log n)$$

Proof by Contradiction

- Assume our algorithm incorrectly states B as the majority, when in reality it is A
- This must mean that at some point in the recursive process, we made an incorrect choice in finding the majority of a critical subgroup

- For us to have incorrectly returned B, then we must have returned "no majority", because if we returned A in this critical subgroup (a subgroup that determined the majority), the overall output would have been correctly A
- But, because our algorithm instead returned "no majority", our overall solution should have also output "no majority" because this was a critical subgroup
- But, this contradicts that our algorithm actually output B
- Thus, our algorithm must be correct

Problem 4

> Page 248, Exercise 5

Algorithm

- Given n lines, divide the lines in half
- Recursively, divide the lines into halves → subgroups, compare the slopes of individual lines
- If the slopes are the same, the line with the greater intercept will hide the other line, so remove this hidden line
- If the slopes are not the same, a line is hidden if it intersects another line at an x value before the origin
- Begin merging these subgroups of lines and recursively check if they are hidden and handle them accordingly
- Return the remaining set of lines as those visible

Time Complexity Analysis

- This algorithm uses divide and conquer to split the set of lines into indivisible subgroups and merge them

- Splitting takes $O(\log n)$ and merging is $O(n)$ for each division
- So, the overall time complexity is

$$O(n \log n)$$

Proof by Cases

- Case 1: slopes are the same
  - Lines of the same slope are only visible if they intercept the y-axis at the highest point
- Case 2: different slopes
  - Case 1: lower slope
    - the lower slope line is visible if it is intercepted by the higher slope line after x=0
  - Case 2: higher slope
    - the higher slope line will always be visible at some x as $x \to \text{inf}$
- Thus, after dividing, we only contain lines that are not hidden

Problem 5

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

Algorithm

- Given an array `a`, find the middle value where the index `i=n/2`
- We have found $k = i$ if `a[i-1] > a[i]`

- If this is not true, then we compare the value at `a[i]` to the last value in the list `a[n-1]`
  - If `a[i] < a[n-1]`, then we check the slice from index `0` to index `i` and select the middle element of this slice
  - Otherwise, we check the index `i` to the index `n-1` and select the middle element of this slice
- We continue to do this until we have found k
- If we reach a point where there is no value left of `a[i]` and `a[i] < a[n-1]`, then the array is not shifted, and we return $k = 0$

Time Complexity Analysis

- We implemented a binary search that cuts the solution space in half each time we select a new middle value
- Thus, the overall time complexity is

$$O(\log n)$$

Proof by Cases

- Because the array is sorted, we know that k must be equal to the index of the minimum value
- Again, because this list is sorted increasingly then shifted, the minimum value must be the value where the value left of it is greater than it (if there is no such value, then the list has not been shifted)
- Because we know this fact, we can select the middle value and compare it to the values left and right of it
- We will see that if the left value is less than it, then there are **2 non-trivial cases** (if not, then we have found the minimum value):
- Case 1: we are in the shifted part of the list

    - Because the list is sorted and shifts occur to the right, if we are in the shifted part of the list, then the value at i must be greater than the last value in the shifted list
    - If so, then we know that the minimum value must occur to the right of where we currently are, so we should check the part of the list from indices `i` to `n-1`
- Case 2: We are in the sorted part of the list
    - Similar to the previous case, we can check what part of the list we are on by comparing the value at our current location, i to the last value in the shifted list
    - If the last value is greater, then we know we are in the sorted part of the list, and the minimum value must be to the left of where we are
    - So, we should then search the slice of the list from indices `0` to `i`
- Because the values are distinct, these are the only possible cases when we select the middle value; so, based on these cases, we should select our new middle value to be the middle of the new slice (which we will know based on our answer to the above cases)
- As we continue to do this, we cut our possible range of values to check (the slice) in half each time, thus a time complexity of $O(\log n)$
- Repeating this until our check is true or we reach the beginning of the list, we can determine the number of shifts k as the final index we land on i

Problem 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 it).
> Ex.
> Input : `Array 1 - 2 3 6 7 9`
> `Array 2 - 1 4 8 10`
> `k = 5`
> Output : `6`

Algorithm

- Given 2 sorted arrays `a,b` of size `n,m` respectively, and a 1-indexed position `k`, we want to find the value at the `k`-th value of the merged array (without actually merging)
- We can use a recursive helper function that takes left and right pointers for `a` denoted `l1,r1` respectively and left and right pointers for `b` denoted `l2,r2` respectively, and a calculated temporary value of the number of elements in a proposed merged array denoted `kc`
  - e.g. `helper(l1,r1,l2,r2,kc)`
  - the rest of the implementation will be done in this recursive helper function
- we have base cases when the left pointer passes the right pointer
  - if `l1 ⩾ r1`, then we return the value at `b[l2 + kc -1]` (the value at the end of the slice of the `b` array)
  - if `l2 ⩾ r2`, then we return the value at `a[l1 + kc - 1]` (the value at the end of the slice of the `a` array)
- We then determine middle indices and the counts of each array from the start (left) index to the middle index, defined as the following:
  - `m1 = (l1+r1) / 2` (using integer division)

    - `m2 = (l2+r2) / 2` (using integer division)
    - `mid1_count = m1 - l1 + 1` (the number of values in `a` from index `l1` to `m1`)
    - `mid2_count = m2 - l2 + 1` (the number of values in `b` from index `l2` to `m2`)
- Now, for the general case, we have 2 cases:
- if `mid1_count + mid2_count ⩽ kc`, then our imaginary merged array of `kc` values does **not** include `k` so we have 2 sub-cases:
    - if the values `a[m1] < b[m2]`, then we recursively call the helper function with the following parameters:
        - `helper(l1=(m1+1), r1=r1, l2=l2, r2=r2, kc=(k-mid1_counts))`
    - otherwise, `a[m1] ⩾ b[m2]`, then we recursively call the helper function with the following parameters:
        - `helper(l1=l1, r1=r1, l2=(m2+1), r2=r2, kc=(k-mid2_counts))`
- otherwise, `mid1_counts + mid2_counts > kc`, then the `k`-th value is somewhere in our imaginary merged array, but we do not yet know where, so we need to decrease the size of the slices of the arrays, with 2 cases:
    - if `a[m1] < b[m2]`, then we call the helper function with the parameters:
        - `helper(l1=l1, r1=r1, l2=l2, r2=m2, kc=kc)`
    - otherwise `a[m1] ⩾ b[m2]`, then we call the helper function with the parameters:
        - `helper(l1=l1, r1=m1, l2=l2, r2=r2, kc=kc`
- Now that we have our helper function defined, our algorithm should simply begin with starting indices of `0` and ending indices of `n` and `m`, respectively, and pass `kc=k` directly

Time Complexity Analysis

- At each recursive call of the helper function, we cut the solution space by half of the given array we decide to slice
- Thus, at each recursive step, we perform either $\log n$ or $\log m$ operations
- So, in the worst case, our algorithm's run time is of

$$O(\log n + \log m)$$

Proof by Cases

- The idea behind the divide and conquer solution is that we select slices of each of the input arrays such that we make the proposed merged array up till the length of k without actually merging the two arrays
- Assuming we are given valid inputs, and a solution exists (true if the inputs are valid), we can observe the bae cases are when one array is empty and the other is valid
  - e.g. if `a` is valid and `b` is empty (its left index passes/reaches its right index), then the k-th value must exist only in a specifically in the slice of a we determine to contain k, and vice versa for when b is valid and a is empty
- Then, to maximize the rate at which we cut out values, we can arbitrarily choose to split using the middle index (it's easy to compute with integer division as well and allows for a log-scale run time)
- The idea behind picking the middle value as a slice is that we try to merge the first `l1:m1` values of `a` with the first `l2:m2` values of `b` until we have k values, and we know the k-th value is at the end of this merged array, so **we have 2 scenarios** when we select middle values to slice with:

- Scenario 1: k is not in the union of the sliced arrays; we can check this by counting the number of values in each slice of the arrays (`mid1_counts` and `mid2_counts`) and seeing if it's greater or less than k; then we know we have to expand our selection of slices, and we have 2 cases:
  - if `a[m1] < b[m2]`, then we not only know we want to expand our slice selection, but we also know that in our currently imaginary merged array, `b[m2]` must be the last value (because each `a` and `b` are sorted)
    - so we know that k cannot be in the slice of `a[0 : m1]`, thus we should now check the rest of a, so we can update the lft pointer for a `l1` such that we now select the slice `a[m1+1 : n]` and re-call our algorithm
    - because we are shifting our window of `a` to **expand**, we also want to update the relative position of `k` by adding and subtracting the previously calculated mid-counts of a such that we set `kc = k - mid1_counts`
  - if this is not true, then, similarly, `a[m1]` must be the last value in our imaginary/proposed merged array; thus we know k is not in the range `b[0 : m2]`, so we can shift `b's` left-pointer `l2` such that we select the slice `b[m2+1 : m]`
    - similarly, because we are now checking a new slice, we must shift the relative position of k by defining `kc = k - mid2_counts`
- Scenario 2: we over-select the slices and k is in our proposed imaginary merged array, but we do not know where, so we can again cut (chosen arbitrarily to be half to ensure an optimal run time), and check in this new slice, but to make a cut, we have 2 cases (same as in the previous scenario):

    - if `a[m1] < b[m2]`, then `m2` must be the last index in our proposed merged array and is past the index k so we can cut `b` in half such that we use the slice `b[0 : m2]` by updating `b's` right pointer `r2`
        - BUT, we don't change the relative position of k because we have not determined values that are guaranteed to occur before k (as we do in scenario 1)
    - if not, `a[m1] ⩾ b[m2]`, then `m1` must have passed beyond k and so we can cut this in half by looking at the selection `a[0 : m1]` by updating `a's` right pointer `r1`
        - Again, we do not change `kc` because we have not yet determined which values are guaranteed to come before k as we do in scenario 1
- Now, given that we have passed valid inputs, these are all the possible cases/sub-cases that can occur, so we are guaranteed not to have some unhandled edge cases
- Also, since we cut by half each recursion, we are guaranteed to reach a point where we no longer have any element in at least one of the arrays, so we will reach our base case and not cause infinite recursion - the algorithm is guaranteed to halt at some point.