E0 225 Design & Analysis of Algorithms **Assignment - 1**
Sai Phanindra Korlepara — **24142** — M.Tech CSA Dept.

Problem 1. *JE Book, Chapter 1, Problem 16*
Suppose we are given a set S of n items, each with a value and a weight. For any element $x \in S$, we define two subsets

- *$S_{<x}$ is the set of elements of S whose value is less than the value of x.*
- *$S_{>x}$ is the set of elements of S whose value is more than the value of x.*

For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The weighted median of R is any element x such that $W(S_{<x}) \leq w(S)/2$ and $W(S_{>x}) \leq w(S)/2$.

Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all the values are distinct and all weights are positive.

Solution. We can fashion an algorithm that is similar to Quick Select Algorithm in it's operation such that the average runtime complexity is of the order $O(n)$.

Algorithm

1. From the set $S[1..n]$, select a pivot using median of median method from the list of n elements.
2. Swap the elements smaller than the pivot to it's left.
3. Estimate the $W(S_{<x})$ and $W(S_{>x})$ assuming the chosen pivot is the weighted median.
4. If $W(S_{<x}) \geq w(S)/2$, then the weighted median is to the left of the current pivot.
5. If $W(S_{>x}) \geq w(S)/2$, then the weighted median is to the right of the current pivot.
6. Recursively change the pivot element till we have $W(S_{<x}) \leq w(S)/2$ and $W(S_{>x}) \leq w(S)/2$ for the chosen pivot.
7. The pivot that satisfies this condition is the Weighted median for the given set.

Runtime Analysis

Similar to the Quick select algorithm, this algorithm also has an average case runtime complexity of $O(n)$.

- To estimate the cost of selecting the pivot, we are using median of median method. If we were to use a block size 5, then in each round of MOM recursion, cost of finding the medians is

$$\frac{n}{5} * const(cost\, of\, finding\, median\, of\, 5\, Numbers)$$

The mom algorithm also recursively calls itself to find the median of the medians with a runtime of $T(n/5)$. This makes the Recurrence equation for MOM method as

$$T(n) = T(n/5) + O(n)$$

Using third point of Master Method, Runtime complexity of MOM method can be estimated to $O(n)$.

- Using MOM method, in the worst case, pivot chosen will at worst be $n/4$ elements away from real median. Swapping the elements to their right place takes $O(n)$ time.
- Estimating $W(S_{<x})$ and $W(S_{>x})$ can also be done in $O(n)$ time.
- During the next call of recursion in the the worst case, $3n/4$ elements still remain this makes the Recurrence equation

$$T(n) = O(n) + O(n) + O(n) + T(3n/4)$$

(or)

$$T(n) = O(n) + T(3n/4)$$

Using third point from master method, the time complexity for this method to find the weighted median is $O(n)$. □

Problem 2. *JE Book, Chapter 1, Problems 21 (a) & (b)*
(a). Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. Describe an algorithm to find the median element in the union of A and B in $\Theta(logn)$ time. You can assume that the arrays contain no duplicate elements.
(b). Suppose we are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B$ in $\Theta(log(m+n))$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$.

Solution.
(a). We can use binary search on one of the arrays & and pair a matching element from the second array to find the median of the union of the two arrays.

Algorithm

1. For any given array size n, the union array will have $2n$ elements which is an even number. To find the median of the union array we need to find nth and $n+1$th smallest elements in the union array, whose average gives the median of the Union array

2. Suppose every element $<$ ith element in the array A contribute to the n smallest elements of the union array.(Range of i $\in$ [1,n+1])

3. Then the first $n-i+1$ elements from the array B contribute to the n smallest elements in the union array.

4. To find the correct position for i, we perform a binary search on the array A & check for the following condition,

$$A[i-1] \leq B[n-i+2] \,\&\, A[i] \geq B[n-i+1]$$

5. If suppose $A[i-1] > B[n-i+2]$ during next iteration search the left half during the next iteration of binary search.

6. But if $A[i] < B[n-i+1]$ during next iteration search the right half during the next iteration of binary search.

7. The nth element would be the larger of $A[i-1]$ & $B[n-i+1]$. The $n+1$th largest element would be the smaller of $A[i]$ & $B[n-i+2]$.

8. Now we can calculate the median of the union of two arrays by averaging the nth & $n+1$th smallest elements.

Runtime Analysis

The only recursive call during the algorithm is the binary search, where the search array size gets halved during each iteration. In each of the iterations the calculation performed is constant time.

Hence the runtime complexity of this algorithm is limited to how many recursive calls the binary search algorithm will make, which is $\Theta(logn)$ times till the correct i is found.

Hence we have shown an algorithm that will always give correct solution with a time complexity of $\Theta(logn)$.

(b). We can solve this problem similar to how we have solved the first problem in this set. We will assume that all the elements smaller than ith element in the array A contribute to elements smaller than kth. Then we can perform binary search to find the kth smallest element.

1. Let m be smaller than n, if not swap the two arrays so that smaller array will be the array A.

2. To find the kth smallest element in the union of two arrays, Let's assume that the elements smaller than ith of the array A contribute to the elements $\leq$ kth smallest element.

3. Then the first $k-i+1$ elements from the array B must contribute to the elements smaller than kth in the union array.

4. To find the i that satisfies the property, we perform binary search on the array A.

5. During each iteration we will verify if $A[i-1] \leq B[k-i+2]$ & $A[i] \geq B[k-i+1]$. If this condition is satisfied we have reached the correct i where the array A contributes these i elements to the elements $\leq k$th element.

6. If suppose $A[i-1] > B[k-i+2]$ during next iteration search the left half during the next iteration of binary search.

7. But if $A[i] < B[k-i+1]$ during next iteration search the right half during the next iteration of binary search.

8. The kth element would be the larger of $A[i-1]$ & $B[k-i+1]$.

Runtime Analysis

By doing this binary search only on the smaller array we can find the correct position for i and find the kth smallest element.

The time complexity of this algorithm is similar to previous case $\Theta(log(min(m,n)))$, which is faster but exponentially equivalent to the time complexity expected in question. □

Problem 3. *KT Book, Chapter 4, Problem 5*
Let's consider a long, quiet country road with houses scattered very sparsely along it. (we can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these house are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Solution. We can try to come up with a greedy solution to figure out the minimum number of base stations that are required to cover all the residents.

Algorithm

1. Start from the eastern most house.

2. Setup the first base station 4 miles away from this house towards the west end (or the western most point of the road if the distance between this house and the end is less than 4 miles).

3. All the houses up to 8 miles west of the easternmost house are covered by this base station.

4. Repeat steps 1,2,3 starting the easternmost house that is uncovered at this stage of the algorithm

5. Terminate when there is no house left that is not covered by any of the base stations

By placing each new station 4 miles away from the first house encountered, we will be covering all the homes while trying to minimize the empty distance and overlapping coverage of the base stations.

Runtime Analysis

In this algorithm we start from east end and scan for houses till the last one on the west end. We come across each house only once and there are no Loops or Recursive calls. Hence the Runtime complexity is $O(n)$, where n is the number of houses.

Proof of Correctness

If we can show that the algorithm has Greedy choice property and Optimal Sub-structure, the solution produced by the algorithm must be optimal

- Let H_1 be the eastern most house and B_1 be the base station determined by the algorithm described above
- If any base station is placed east of B_1, would cover fewer houses than B_1. If the first base station is placed west of the spot B_1 is in, then H_1 would not be covered by this new base station position
- Thus placing the base station at B_1 is a locally optimal choice and we can say that the algorithm gives a locally optimal solution and has the Greedy Choice Property.
- After placing the first base station at B_1, all the houses 8 miles east of H_1 are covered.
- Now the problem boils down to finding the optimal number of base stations required to cover remainder of the houses.
- The problem that remains is structurally identical to the initial problem. Line of houses with base stations to cover them.
- We can follow the same reasoning from above to find the optimal spot for the next base station location using the greedy property of the problem.
- Hence we have also showed the Optimal Sub-Structure property of the Algorithm

Hence we have proved that the Greedy algorithm discussed always does provide the Optimal & Correct solution. □

Problem 4. *KT Book, Chapter 4, Problem 17*
Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs *on the processor. Each such job comes with a* start time *and an* end time *; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)*

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

Solution. We can solve this problem by iterating through the jobs that run overnight one after the other, select from the remaining jobs using the standard scheduling algorithm, then consider the schedule that can accommodate most requests.

Algorithm
From the list of Jobs that are requested to be scheduled, to optimize the schedule such that we can maximize the number of jobs that are executed in a day

1. Sort all the job requests according to their finish time in an ascending order.
2. We will loop through cases, one for each one of the jobs that cross the midnight point and one case without any of the jobs crossing the mid night point.
3. Remove all the job requests that conflict with the chosen job in the previous step.
4. Pick the first job in the remaining list of job requests as the one to be executed.
5. Remove all the jobs that conflict with the selected job.
6. Once this job is finished, Follow steps 4 & 5 till there are no more jobs in the sequence.
7. Record the number of jobs that were selected in this schedule and update the minimum if required.
8. Restart from the third step with a different job that is crossing the mid night point.
9. Once all the possibilities are checked we have the maximum possible number of Job requests that can be accepted.

Runtime Analysis
The portion of the Algorithm from steps 3-7 is the same as the normal Scheduling algorithm. If we account for the iterations for each of the jobs crossing the mid night point we can estimate the runtime complexity of this algorithm

- In step 1, the run time complexity for best sorting algorithm is $O(nlogn)$.
- The number of jobs that cross the midnight point in the worst case is of order $O(n)$.
- During the steps 3-4, we scan through the list of jobs, once for each job, hence the time complexity for steps 3-4 is of order $O(n)$.
- The total runtime complexity for this algorithm can be calculated as

$$T(n) = O(nlogn) + O(n) * O(n) = O(n^2)$$

Hence the overall runtime complexity for this algorithm is of the order $O(n^2)$.

Proof of Correctness

We can prove that the algorithm give an optimal and correct solution by using Proof by induction that there cannot be a more optimal solution than the one estimated by the algorithm.

- Any optimal solution for a given set of job requests can only accept one or none of the job requests that cross the mid night point.
- In the step-2 of the algorithm, we consider all possible options for jobs that cross the midnight point (including the case where none of them are selected). Hence any solution that is optimal is being considered further.

Claim : The sequence of jobs chosen during the steps 3-7 is the optimal solution given the job chosen in step-2.

- **Base Case :** When there is only 1 job request the greedy algorithm chooses the only available job. This is trivially optimal solution as it is not possible to select any more jobs.
- **Inductive Hypothesis :** Suppose the greedy algorithm works optimally for any set of k jobs
- **Inductive Step :** we will show that the greedy algorithm is also optimal for a set of $k + 1$ jobs.
- During this step, The greedy algorithm will select the job that is finishing first. If there were an "optimal" solution that selects any other job at this step, it must mean that the job selected by this "Optimal" solution will end later than the job selected by the greedy algorithm.
- For any given sequence of jobs that come after the $k + 1$th job in this "optimal" solution, we can argue that even if we choose the greedy choice, we can still make the same choices that were made by this "optimal" solution.
- This is because all the jobs after the $k + 1$th job chosen by the "optimal" solution, start after the greedy choice, and the greedy algorithm could make the same choices for the jobs after the $k + 1$th.
- In conclusion, By the inductive hypothesis, since the greedy algorithm is optimal for any set of k jobs, and since we have shown the selection of $k+1$th choice by the greedy algorithm is optimal & leaves most time for subsequent jobs, the greedy algorithm is also optimal for $k + 1$ jobs.

Thus by induction, we have shown that the greedy algorithm for the circular scheduling problem will select the maximum possible number of non overlapping job requests with a runtime complexity of $O(n^2)$. $\square$
