6.046
PROBLEM SET 1

JOHN WANG

Collaborators: Sashko Stubailo, Ryan Liu, Varun Ganesan, Josh Ma, Brodrick Childs, Delian Asparouhov

1. Problem 1

Problem 1.1. *Assume that the modified algorithm picks the median element from groups of size 7. Revise the analysis done in lecture and CLRS, and write the recurrence relation for $T(n)$. Solve the recurrence using the substitution method. Make sure to explicitly state any assumptions you make about the base case (for example, for groups of 5, we assumed $T(n) \leq O(1)$ for $n < 140$).*

Solution First, we will obtain the recurrence relation. We will have broken the array into $n/7$ groups, which means we will have to select $n/7$ medians. This will take $T(n/7)$ time. Moreover, breaking the array into groups will take $O(n)$ time. Finally, we want want to know which elements are definitely larger than the median of medians.

First, we know that 3 elements from each group with medians larger than the median of medians will be larger. There are $(n/7)(1/2)$ of these groups. However, we must subtract off one element for the median of medians and at least 2 elements for groups which are smaller than 7. This means, we have $3((1/2)(n/7) - 3)$ elements that we know are larger than the median of medians. By symmetry these are the same number of elements we know that are smaller than the median of medians.

Thus, we know that in the worst case, we will have to run the algorithm on at most $n - 3((1/2)(n/7) - 3) = (11n)/14 + 9$ elements. We have the following recurrence then:

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{11n}{14} + 9\right) + O(n) \tag{1.1}$$

We want to prove that $T(n) = O(n)$ so that $\exists a > 0$ such that $T(n) \leq an$. We will use the substitution method, with a base case of $T(n) = O(1)$ for all $n \leq 168$. Let us assume using the inductive hypothesis that $T(k) \leq ak$ for all $k < n$. From there, we can solve:

$$\begin{align} T(n) &\leq T\left(\frac{n}{7}\right) + T\left(\frac{11n}{14} + 9\right) + cn \tag{1.2}\\ &\leq a\left(\frac{n}{7}\right) + a\left(\frac{11n}{14} + 9\right) + cn \tag{1.3}\\ &= \frac{13an}{14} + 6a + cn \tag{1.4}\\ &= an + \left(-\frac{an}{14} + 6a + cn\right) \tag{1.5}\end{align}$$

We want $-(an)/14 + 6a + cn \leq 0$. This occurs when:

$$\begin{align} a\left(\frac{n}{14} - 6\right) &\geq cn \tag{1.6}\\ a &\geq \frac{14cn}{n - 84} \tag{1.7}\end{align}$$

But we know that $n > 168$ so that $\frac{n}{n-84} \leq 2$. This means that $a \geq 28c$. Therefore, we can choose $a \geq 28$, and this will show that $T(n) \leq an$. □

Problem 1.2. *Now, consider what happens if we choose the group size to be 4. Assume that we always choose the smaller of the two middle elements for the median of every group (i.e., the 2nd smallest element). Again, write the recurrence relation for $T(n)$, and solve it using the substitution method. Be sure to state any assumptions.*

Solution We know that it will take $O(n)$ time to partition the elements into groups of 4. The median of medians will take $T(n/4)$ time to find. Once we have partitioned the elements into groups, we want to find the number of elements larger and smaller than the median of medians.

Half of the $n/4$ groups will have medians which are larger than the median of medians, and 3 elements from each group will therefore be larger. We have $3((1/2)(n/4))$ elements that are larger than the median of medians.

Half of the $n/4$ groups will have medians which are smaller than the median of medians, but only two elements from each group will be gauranteed be be smaller (since we have the second smallest element as the median). This means that we have $2((1/2)(n/4))$ elements that are smaller than the median of medians.

In the worst case, we will always recurse on the half with a larger half, since there are $(6n)/8$ elements in that case, while only $(5n)/8$ elements when k is smaller than the median of medians. Thus, we have the following recurrence:

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{6n}{8}\right) + O(n) \tag{1.8}$$

We will show that $T(n) \le an\log(n)$. First, we take a base case of $T(n) \le O(1)$ for $n \le 2$. Using the inductive hypothesis, we assume that $T(k) \le ak\log(k)$ for all $k < n$. Then we have:

$$\begin{aligned}
T(n) &= T\left(\frac{n}{4}\right) + T\left(\frac{6n}{8}\right) + O(n) && (1.9)\\
&\le \frac{an}{4}\log\left(\frac{n}{4}\right) + \frac{6an}{8}\log\left(\frac{6n}{8}\right) + cn && (1.10)\\
&= \frac{an}{4}\log(n) - \frac{an}{4}\log(4) + \frac{3an}{4}\log(n) + \frac{3an}{4}\log\left(\frac{3}{4}\right) + cn && (1.11)\\
&= an\log(n) + \left(-\frac{an}{4}\log(4) + \frac{3an}{4}\log\left(\frac{3}{4}\right) + cn\right) && (1.12)
\end{aligned}$$

We must show that the right side of the equation is less than zero. This occurs when:

$$\begin{aligned}
\frac{an}{4}\log(4) &\ge \frac{3an}{4}\log\left(\frac{3}{4}\right) + cn && (1.13)\\
\frac{a}{4}\log(4) - \frac{3a}{4}\log(3) + \frac{3a}{4}\log(4) &\ge c && (1.14)\\
a\left(\log(4) - \frac{3}{4}\log(3)\right) &\ge c && (1.15)
\end{aligned}$$

Since $\log(4) = 2\log(2)$ and because $-(3/4)\log(3) < -(3/4)\log(2)$, we can write:

$$\begin{aligned}
c &\le a\left(2\log(2) - \frac{3}{4}\log(2)\right) && (1.16)\\
c &\le a\left(\frac{5}{4}\log(2)\right) && (1.17)\\
c &< 2a && (1.18)
\end{aligned}$$

Therefore, if we choose $a > c/2$, then we will have shown that $T(n) = O(n\log n)$. $\square$

2. Problem 2

Problem 2.1. *The inhabitants of 1-D land live in one-dimensional regions. You own a stretch of land in the shape of a circle, and would like to sell it to n customers. Each of these customers is interested in a contiguous segment on the perimeter (i.e., an arc), which can be represented in polar coordinates. For example, customer i wants to have θ_{i1} to θ_{i2} on the circle. Unfortunately, many of these arcs overlap, and you cannot sell a portion of a customer's request. Give an efficient algorithm to maximize the number of customer requests you can fulfill.*

Solution The first solution can be thought of as a special case of problem 2.2. We simply use the following weight function: $w(i) = 1$ for all $i \in \{1, \dots, n\}$ where i is an arc in the input. This, we attempt to maximize the number of customer requests we can fill as the weight function will simply correspond to the number of filled requests. We will prove the correctness of the algorithm in the solution to problem 2.2. The runtime, as analyzed in problem 2.2, is $O(n^2)$. $\square$

Solution A second, more efficient solution, takes $O(n\log n)$ time. The algorithm, called $CircleArcMaximization$ uses a variant of dynamic programming to cache subproblem results. The input to the algorithm is a list $(\theta_{11}, \theta_{12}), \dots, (\theta_{n1}, \theta_{n2})$ of starting and ending coordinates on a circle.

The algorithm takes the first (or any arbitrary) starting point S in the list of coordinates. We remove all arcs that intersect this starting point, and insert the $(\theta_{i1}, \theta_{i2})$ tuples into a list $CachedIntersections$. Next, we use our $LineProfitMaximization$ algorithm from class (the dynamic programming version of task scheduling on a line) with weights of 1 for each arc. This function sorts the events into times $t_1, t_2, \ldots, t_m$ which can be either starting or ending times. Note that $t_k = (\theta_{ij} - S) \mod 2\pi$ so that $t_k = 0$ if $\theta_{ij} = S$. So we have basically defined 0 of our polar coordinates at S. The $LineProfitMaximization$ function will take in these times and will cache a result $DP(i)$ in a dictionary DP which corresponds to the maximum number of arcs that can fit into the interval from the starting point S to time t_i. It will also cache a result $BS(i)$ in a dictionary BS which corresponds to the best starting time that can be achieved for $DP(i)$ arcs in the interval S to t_i. Thus, if $DP(i) = q$, then $BS(i)$ stores the largest possible starting time of a solution with q arcs inside of the interval from S to t_i.

Note that the BS dictionary can be updated in $O(n)$ total time for all i if it is constantly updated as the dynamic programming algorithm proceeds. The dynamic programming algorithm will go through a loop of all times t_i, where it will update $DP(i)$. However, it will also be updating $BS(i)$ concurrently. Thus, if $DP(i) = DP(i-1)$, then $BS(i) = BS(i-1)$ or if $DP(i) = 1$, then $BS(i) = t_i$. In effect, $BS(i)$ will copy the updates of $DP(i)$, but we will replace BS for DP in the recursion.

Once the algorithm has finished its DP for all i not in the $CachedIntersections$ dictionary, we can examine the intersecting arcs. Create a new array called $LineResults$. We loop through all of the $(\theta_{i1}, \theta_{i2})$ tuples in the $CachedIntersections$ list and convert them into our new polar coordinates (t_{i1}, t_{i2}). Then, we do a binary search for the closest ending point s_k coming before t_{i1}, and we check if $BS(k) > t_{i2}$. If so, then we append $DP(k) + 1$ to $LineResults$, else we simply append $DP(k)$ to $LineResults$. We do this for all arcs in the $CachedIntersections$ dictionary. Finally, we return the maximum of $LineResults$. Here is the pseudocode:

```
def CricleArcMaximization(ThetaList):
     S = GetArbitraryStart(ThetaList)
     Remap coordinates so that 0 <= Theta < 2 Pi for all Theta
             and S is defined as the 0 of the polar coordinate system
     TimeList = []
     CachedIntersections = []
     for (start, end) in ThetaList:
          If (start, end) does not intersect S:
                TimeList.append((start, end))
          Else:
                CachedIntersections.append((start, end))

     LineResults = []
     DP, BS = LineProfitMaximization(TimeList)
     for (start, end) in CachedIntersections:
           Binary search for first starting time s > end in TimeList
           i = Index of s
           If BS(k) > start:
                 LineResults.append(DP(i) + 1)
           Else:
                 LineResults.append(DP(i))
      for i in TimeList:
           LineResults.append(DP(i))
      return max(LineResults)
```

We will now go through an example of the algorithm. Consider the input $\{(0, \pi/2), (\pi/4, \pi), (\pi, 2\pi)\}$. The maximum number of arcs that one can fit is clearly 2 by inspection. The algorithm will proceed as follows. It will first pick an arbitrary starting point. Let us say this starting point is $\pi/4$. Then it will reorder the input so that $\pi/4$ is now defined as the zero in polar coordinates, we will now have $\{(7\pi/4, \pi/4), (0, 3\pi/4), (3\pi/4, 7\pi/4)\}$. Now, since the intersecting arc at 0 is $(7\pi/4, \pi/4)$, so that $CachedIntersections = [(7\pi/4, \pi/4)]$. Now, we sort our inputs and number them from 1 to m. We have $TimeList = \{0, 3\pi/4, 7\pi/4\}$. We perform our DP algorithm and find that $DP(1) = 0, DP(2) = 1, DP(3) = 2$ and $BS(1) = None, BS(2) = 0, BS(3) = 0$.

Now, we add $(7\pi/4, \pi/4)$ back in, binary searching for first start time after $\pi/4$. This is $3\pi/4$, and it has a DP value of 1. We check its BS value, and we find it is 0. Thus, we take the maximum of all the DP values and $1 + 0 = 1$. This gives 2, which is what we wanted.

The run time of the algorithm is $O(n \log n)$. This is because the initialization and remapping of coordinates takes $O(n)$ time. It also takes $O(n)$ time to check for intersection of S. Next, the dynamic programming algorithm in $LineProfitMaximization$ takes at most $O(n \log n)$. Finally, we iteration through at most $O(n)$ arcs in $CachedIntersections$, where each binary search takes $O(\log n)$. Thus the loop takes a total of $O(n \log n)$ time, and finding the maximum of $LineResults$ takes $O(n)$. Thus, $T(n) = O(n) + O(n \log n) + O(n \log n) + O(n) = O(n \log n)$.

To prove correctness, we first will show that the optimal solution can be at most $k + 1$, where k is the solution given by $LineProfitMaximization$. This follows because we can add at most a single arc (originally overlapping at the start point) to the solution. We can only add a single arc because at the start point, all arcs are already overlapping with each other when they are removed, so the optimal solution can only have at most one extra arc.

Next, we will show that using the DP and BS caches, we will always be able to find an optimal solution if one exists. Suppose the optimal solution contains an arc that was removed in the beginning and placed in $CachedIntersections$. Then the dynamic programming algorithm will cache the best beginning point for the ith end point. Doing a binary search to find the next closest endpoint means the cache contains the optimal solution without the removed arc. Moreover, because we have the best starting time at which this subset of the optimum is attained, we can also check if we can append the new arc to obtain a new the optimal solution. Since we perform this operation for all removed edges, we can be gauranteed that we wil find an new optimum of $k + 1$ arcs if it exists. This completes the algorithm.

□

Problem 2.2. *You are a firm believer in the free market and competition. So, in addition to each customer giving you an arc that she would like, she also offers a price. However, you still cannot sell a portion of any request. Give an efficient algorithm to maximize your profit.*

Solution I will present an algorithm that maximizes profit in $O(n^2)$ time. The algorithm uses the dynamic programming solution presented in class for maximizing profit on a flat line. We shall call this the $LineProfitMaximization$ function, which takes in arguments $(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)$ as a sorted list of starting and ending times and $w_1, w_2, \ldots, w_n$ as a list of weights for each task. Note that this function takes in a sorted list of starting and end times, and uses the dynamic programming recursion seen in class to output the maximum profit. This $LineProfitMaximization$ function takes $O(n)$ time because it is being passed a sorted array.

The input of our algorithm, which we shall call $CircleProfitMaximization$, is a list of arc beginnings and endings $(\theta_{11}, \theta_{12}), \ldots, (\theta_{n1}, \theta_{n2})$ and weights for each arc $w_1, w_2, \ldots, w_n$. First, we will create coordinates $t_1, t_2, \ldots t_m$ where each t_i can be either a starting or finishing time and $m \leq 2n$ by definition. As we more through our list $(\theta_{11}, \theta_{12}), \ldots, (\theta_{n1}, \theta_{n2})$, we take $t_1 = \theta_{11} \mod 2\pi, t_2 = \theta_{12} \mod 2\pi, \ldots$. We will then sort these m times into ascending order, knowing that we must have $0 \leq \theta_{ij} < 2\pi$ for all i and j. Then we pass in our list of times $t_1, t_2, \ldots, t_m$ into the $LineProfitMaximization$ function, removing any arcs where $\theta_{i1} > \theta_{i2}$, or in other words, any arcs that intersect $\theta = 0$. We shall store this value in an array $LineResults$ and perform this same operation, but now taking the next largest starting point t_k and letting this be the new start position. Thus, all the t_i values will become $t_i + t_k \mod 2\pi$ for all $i \in \{1, \ldots, m\}$. At the end of the algorithm, we return the maximum result in $LineResults$. The pseudocode for this algorithm is given below:

```
def CircleProfitMaximization(ThetaList, Weights):
    TimeList = []
    StartList = []
    for (start, end) in ThetaList:
        Append (start % 2pi) and (end % 2pi) to TimeList, each containing
                pointers startPointer and endPointer which point to
                (start) and (end) respectively
        Append (start % 2pi) to StartList
    TimeList.sort()
    StartList.sort()

    LineResults = []
    for start in StartList:
        CurrentList = []
        for time in TimeList:
            start = (GetOriginalTheta(startPointer) - start) % 2pi
            end = (GetOriginalTheta(endPointer) - start) % 2pi
            Append to CurrentList if (start < end)
            Reorder CurrentList into sorted order removing duplicates
            LineResults.append(LineProfitMaximization(CurrentList, Weights))
     return max(LineResults)
```

The runtime analysis is simple. The beginning step of changing the input data tuples into a single list of starting and finishing times takes $O(n)$ time. Creating pointers and creating a second *StartList* does not change this. Sorting this takes $O(n \log n)$. Next, we loop through each time in the start list $O(n)$ times, because there are $O(n)$ starting positions. In each part of the loop, we perform $O(n)$ operations to modify the times so that zero degrees is defined at the current starting position in the StartList. Checking for intersections and appending the non-interseting with zero times takes $O(n)$. Redordering *CurrentList* into sorted order only takes $O(n)$ because there is a split point at 0 (which can be found in $O(\log n)$ with binary search) below which everything is still sorted and larger than the last element, so it can be appended to the end of the list. Finally, the *LineProfitMaximization* function takes $O(n)$ time. Thus, the entire loop takes $O(n) * (O(n)) = O(n^2)$ time. Finding the maximum at the end takes $O(n)$ time since there are at most n elements in *LineResults*. The total runtime is then:

$$T(n) = O(n) + O(n \log n) + O(n^2) + O(n) = O(n^2) \tag{2.1}$$

Therefore, we have found an $O(n^2)$ algorithm for solving our problem.

Let us now go through a simple example of the algorithm. We will have the following inputs: $\{(0, \pi/2), (\pi/4, \pi), (\pi, 2\pi)\}$ and weights $3, 1, 4$. Clearly, the optimal solution is to choose arcs 1 and 3 to obtain a weight of 7. Our algorithm will proceed as follows. Let us start with 0 as the beginning point. The sorted array of *TimeList* will then be $\{0, \pi/4, \pi/2, \pi, 2\pi\}$. Nothing overlaps with 0 at this point, so we will perform our dynamic programming algorithm on this array. We will obtain a result of 7, and we will append this to our *LineResults* array.

Next, we will move on to $\pi/4$ being the next starting point. The new *CurrentList* will look like this: $\{0, 3\pi/4, 7\pi/4\}$, which is just shifted by $\pi/4$ from the original *TimeList*, modulo 2π, with the $(0, \pi/2)$ arc removed since it intersects with $\pi/4$. It is easy to resort this in $O(n)$ time by binary searching for 0, and moving everything to the left of 0 (in this case there is nothing and the array is already sorted) to the end of the array. We perform the dynamic programming algorithm, and it returns 5, which we append to our *LineResults* array.

Finally, we move on to π being our starting point. We will not remove any arcs, since nothing overlaps at π. Our *CurrentList* will become $\{\pi, 5\pi/4, 3\pi/2, 0, \pi\}$. We can resort this list by finding 0 and moving everything to the left of 0 (namely $\{\pi, 5\pi/4, 3\pi/2\}$) to the end of the array to obtain: $\{0, \pi, 5\pi/4, 3\pi/2\}$ (if we remove the duplicate π). Now we perform our dynamic programming algorithm and it returns 7.

Thus, we have *LineResults* $= [7, 5, 7]$, so taking the maximum, we return 7 which is what we wanted.

Theorem 2.2. *If the optimal solution has weight W returned, then the CircleProfitMaximization algorithm returns W.*

Proof. First, we know from class that if k^* is an optimal solution on a line, then *LineProfitMaximization* will return k^*. Next, we know that if there is an optimal solution on the circle, then none of the arcs overlap. Let us say an optimal solution consists of $(\theta^*_{11}, \theta^*_{12}), \ldots (\theta^*_{k1}, \theta^*_{k2})$.

Note that if the *LineProfitMaximization* algorithm is begun at any starting or ending point θ^*_{i1} or θ^*_{i2} which belongs to an optimal solution (with the correct modifications of starting times so that the polar coordinates begin at θ^*_{i1} or θ^*_{i2}), then *LineProfitMaximization* will return the optimal solution on a circle. This is because we know the optimal solution can be peeled apart into a line by cutting at any arbitrary starting or ending point. Since there are no overlaps in the optimal solution, we are gauranteed not to have any arcs intersecting our arbitrarily selected starting point. This means that there exists an isomorphism from the circular space to the linear space. Since we know *LineProfitMaximization* returns an optimal solution in the linear space, it will also return an optimal solution in the circular space if we begin *CircleProfitMaximization* at a starting or ending point of the optimal solution.

Since we start *CircleProfitMaximization* at every possible starting point, we are gauranteed to begin the algorithm on some arc which belongs to an optimal solution. This means *LineProfitMaximization* will return an optimal solution W, which will be appended to $LineResults$. Since we take the maximum of $LineResults$, which will be W, $CircleProfitMaximization$ will return W. $\square$

This completes the solution to problem 2.2.
$\square$