CS 180 Homework 4

Name: Wang, Zheng
UID: 404855295

CS 180 Dis 1E
Time: F 10:00-11:50
TA: Orpaz Goldstein

Question 1 (Exercise 13, Page 194):

The initial idea to approach this question is that both the importance of a task J_i (w_i) and the time taken to finish J_i (t_i) should both contribute to our decision-making process. Now, as large t_i tends to persuade us to do the tasks later and large w_i tends to force us to the task earlier, a good way is making decision based on $\frac{w_i}{t_i}$.

Algorithm:

- For each task i we are going to assign:
  - Calculate $S_i = w_i/t_i$
- Sort the task in decreasing order based on S_i
- Return the sorted list of tasks G, which will be the optimal schedule

Proof:

We will show that for any other schedule $G' \neq G$, there exist some neighboring pair satisfying:

1. J_i is scheduled before J_k
2. $S_i < S_k$

(there must exist a neighboring (or consecutive) pair with the property above, otherwise, by a simple induction, we see that G' is sorted, and thus $G' = G$)
Then, if we swap them, we would result some $\sum_{i=1}^{n} w_i C_i$ that is no worse than original one.

Let the total weighted sum of G' be $E_0 = \sum_{i=1}^{n} w_i C_i$ and this is equivalent to $E_0 = \sum_{j=\{1,2,..,n\}\setminus\{i,k\}} w_j C_j + w_i(C_{i-1} + t_i) + w_k(C_{i-1} + t_i + t_k)$. For simplicity, we will just call the term $\sum_{j=\{1,2,..,n\}\setminus\{i,k\}} w_j C_j$ to be Σ_o.

Since if we swap J_i and J_k the rest of the schedule is not affected, after we have swap J_i and J_k, we have the total weighted sum to be:

$$E_1 = \Sigma_o + w_k(C_{i-1} + t_k) + w_i(C_{i-1} + t_i + t_k)$$

To make a comparison between the two, we take $\Delta E = E_0 - E_1 = w_k t_i - w_i t_k$. If we have $\Delta E < 0$, then we have $\frac{w_k}{t_k} = S_k < \frac{w_i}{t_i} = S_i$, a contradiction with the assumption we made.

Thus, for any $G' \neq G$, we can continue doing the above without the risk of increasing the weighted sum $E = \sum_{i=1}^{n} w_i C_i$ and we will eventually get G. Thus G is the optimal schedule.

Complexity:

1. For each element, we calculate the S_i correspond to J_i, this will take $O(n)$ to do it.
2. Sort the list based on S_i will take $O(n \log n)$ to do it.
3. Output the schedule will take $O(n)$ to do.

Thus, the overall time complexity of this algorithm will be $O(n \log n)$.

Question 2 (Exercise 15, Page 196):

The initial idea is that if we choose to include a task, then from the start of that task, to the start of that task next day, we have a 24-hour interval we can schedule. Thus, we can simply apply the technique we used in Interval Schedule Problem to find the optimal solution with that interval.

Algorithm:

- Sort the intervals based on their end point
- For each interval in the list
  - Take that event to be in the schedule
  - Remove all of the intervals overlaps with it
  - Add interval which ends the earliest among the rest of the list
  - Repeat until all of the interval are gone
- For the each of the resulting schedule above
  - Find the one with maximum number of tasks and return it

Proof:

I will show that if a optimal schedule S' contains some task k, then the algorithm mentioned above will find a schedule S no worse that.

First, observe that the algorithm above must run an iteration with a schedule containing task k.

Now the start of k "today" and the start of k "next day" will form the 24-hour interval of "Interval Schedule Problem" for both S and S'.

Suppose that the first n ($n \geq 1$ as k must be the same for S and S') intervals are the same for S and S', then since S will pick intervals that ends first, the $n+1$ interval I'_{n+1} in S' must ends later than the $n+1$ interval I_{n+1} of S. Thus, we will have no trouble substitute the I'_{n+1} of S' with the I_{n+1} in S without the risk of decreasing the total number of intervals in S'. We will call this schedule after substitution S''.

Then we just generate a schedule S'' with $n+1$ interval matching with S and performs no worse than S'. We can continue this process for S'' until it is identical to S and performs no worse than S'.

Thus, the algorithm will result in a schedule that is optimal.

Complexity:

1. The original sorting will take $O(n \log n)$ time to complete.
2. Each iteration of the loop requires a scan of all intervals, which takes $O(n)$ time to run. since we have to repeat this for all n intervals, the loop will overall take $O(n^2)$ time to run
3. We will generate n such "possibly optimal" schedule and we want to scan them all to find the best one. Thus, this part will take $O(n)$ time to run.

Thus, overall, we will need $O(n^2)$ time to run this algorithm.

Question 3 (Exercise 2, Page 246):

The idea is that when we are do merge sort, we can do two merges. The first one is used to the get a sorted list after the merge. The second one will modify the one of the lists to count the number of strong inversions.

For the algorithm described below, assume that the left-array has N_L strong inversions and the right-array has N_R strong inversions.

Algorithm:

- Define function ***merge*** *(left-array, right-array)*
  - Initialize two pointers, each point to the beginning of left-array and right-array
  - Initialize an empty array of the merged_list
  - While neither pointer reaches the end
    - Compare the value of the two pointers
    - If left_value $\leq$ right _value
      - Put left_value to the merged_list
      - Increment the left pointer
    - If left_value $>$ right_value
      - Put right_value to the merged_list
      - Increment the left pointer
  - End while
  - For the array whose pointer is not at the end, put everything after the pointer (including the pointer's position) to the merged_list
  - Set each element e_i of the right-array to be $2 \times e_i$.
  - Initialize two pointers, each point to the beginning of left-array and right-array.
  - Initialize an integer $N = N_L + N_R$
  - While neither pointer reaches the end
    - Compare the value of the two pointers
    - If left_value $\leq$ right _value
      - Increment the left pointer
    - If left_value $>$ right_value
      - Increment the left pointer

- Take I to be the number of elements from the left pointer (inclusive) to the end of the left-array
- Set $N = N + I$

  - End while
  - Return merged_list with attribute N

- Recursively divide the whole list $a_1, a_2, a_3, \dots, a_n$ into left-array of size $\lfloor n/2 \rfloor$ and right-array of size $n - \lfloor n/2 \rfloor$ until the we can no longer divide the list (i.e. each sub-list is of size 1). Set all of the resulting list (which are of size 1) with attribute $N = 0$;
- Recursively call ***merge*** to merge the divided list together until we get a sorted list $a'_1, a'_2, \dots, a'_n$. Get the value of N of the resulting sorted list $a'_1, a'_2, \dots, a'_n$.

Explanation:
The basic idea is very similar to that of normal inversions. But since we are looking for cases such that $i < j$ and $a_i > 2a_j$, we need to do another merge with the 2*right-array.
Let the left array be $a_1, a_2, \dots, a_k$ and the right array be $a_{k+1}, a_{k+2}, \dots, a_n$. It is obvious that the overall number of strong inversions is the sum of strong inversions happens in the left array, in the right array, and those happens between the two.
Since we are doing divide and conquer, we first assume that both lists are sorted. Also, we assume that we know the number of strong inversion local to the left and right array, so it suffices to find the number of strong inversions happens between the two.
Since every element a_i in the left array and every element a_j in the right array has property $i < j$. Then if we find $a_i > 2a_j$, we are sure that $a_{i+1} > 2a_j$, $a_{i+2} > 2a_j$, …, $a_k > 2a_j$.
Thus, we will find $I = k - i + 1$ (the number of elements from the position of the left pointer (inclusive) to the end of the left-array) strong inversions if we find one instance of $a_i > 2a_j$. So, whenever we find an instance of that, we will increment N by I.
Finally, if one of the pointers has reached end, that means we have no more strong inversions happening between the two array and we are done.

Complexity:

1. We can in maximum divide the list into half $O(\log n)$ times
2. For each merge, in the worst case we are going to go through n element to construct a sorted list and n elements in order to count the number of strong inversions between the two list (basically, we are doing another merge). Thus, this will take $O(n)$ time to run.

Thus, the overall complexity is $O(n \log n)$ since we are doing merge whenever we divide.

Question 4 (Exercise 3, Page 246):

The idea is that if there is a majority (more than $n/2$ of the elements in the set are equivalent) in the whole set of the card, then, if we divide the set into two, no matter how we choose to divide, the majority is preserved in one of the subsets.

We will begin by proving the above statement.

Proof:

Suppose that there are m cards that are equivalent (let's label these cards C) and is the majority of a pile of n cards, then we have $\frac{m}{n} > \frac{1}{2}$.

Now suppose towards contradiction that if we divide the whole pile into two, each with k and k' cards, and m_k and $m_{k'}$ card C, but C is not the majority of either of the sub-piles of cards.

Then, $\frac{m_k}{k} \leq \frac{1}{2}$ and $\frac{m_{k'}}{k'} \leq \frac{1}{2}$. So $k \geq 2m_k$ and $k' \geq 2m_{k'}$. Now we have

$$\frac{m}{n} = \frac{m_k + m_{k'}}{k + k'} \leq \frac{m_k + m_{k'}}{2m_k + 2m_{k'}} = \frac{1}{2}$$

A contradiction. Thus, the majority is preserved.

Algorithm:

- Recursively divide the cards into two piles with size $\lfloor n/2 \rfloor$ and $n - \lfloor n/2 \rfloor$ until we no longer need to divide (of size 2 or 1)
- For each of the sub-piles of size 2 or 1
  - If the size of the pile is 1, set maj of the pile to be that card
  - If the size of the pile is 2 and they are equivalent, set maj of the pile to be one of them
  - If the size of the pile is 2 and they are not equivalent, set maj of that pile to be NONE
- Recursively
  - Merge the two sub-piles to reverse the divide done before
    - Check the attribute of the two piles need to be merge
    - If both maj is NONE
      - Put two pile together to get the merged-card-pile
      - RETURN the merged-card-pile with maj set to NONE for next merge
    - If right pile has $maj = C_r$
      - Put two pile together to get the merged-card-pile
      - Use the card tester to test every card in the merged-card-pile against C_r
      - If number of cards that are equivalent to C_r is more than half of the size of the merged-card-pile
        - RETURN the merged-card-pile with maj set to C_r for next merge
    - If the left pile has $maj = C_l$

- Put two pile together to get the merged-card-pile
- Use the card tester to test every card in the merged-card-pile against C_l
- If number of cards that are equivalent to C_l is more than half of the size of the merged-card-pile
    - RETURN the merged-card-pile with maj set to C_l for next merge.

- Put two pile together to get the merged-card-pile
- RETURN the merged-card-pile with maj set to NONE for next merge

Explanation:
Firstly, we will show if there is majority C in the original pile of cards, then, we will find it.

We notice that due to the argument I proved at the beginning, if there is a majority (call it card C) in the original set, then the when recursive dividing stops, one of the sub-piles will have a majority.

This is because the initial divide will result in a sub-pile with majority C, and when we divide this sub-pile, same thing will hold, and we get a "sub-sub-pile" with majority C. If we keep doing this until the dividing ends, we can see that this will result in a sub-pile (of size 1 or 2) with majority C.

Then, during the merging process, a merge that gives the original pile p with majority C (the pile whose split result in p_m) will always involve a sub-pile p_m with $maj = C$. Since p_m has $maj = C$, and p has majority C, by comparing all cards in p with C will, certainly, give us C.

Then we will show that if there is no majority, we will get NONE.

When we are at the last merge (the one that will give us back the original pile of cards), we have two cases:

1. Both sub-piles have $maj =$ NONE
   Then, certainly we will get NONE.
2. At least one of the sub-piles have maj that is not NONE
   Then, by checking the maj with the original pile, we will find out that it not actually the majority of the original pile, and we get NONE.

Thus, if there is no majority, we will get NONE.

Complexity:

1. Dividing recursively will take $O(\log n)$ to run
2. We will do $O(\log n)$ merger, as a reverse of the "divide" we done.
3. In each merge, we will in maximum do two complete check of whole card pile with the candidate majority C_r and C_l. This will take $O(n)$ time to run

Thus, the overall time complexity is $O(n \log n)$.

Question 5 (Exercise 6, Page 246):

To do this problem in $O(\log n)$ time, it is very likely that we will decide which branch of the subtree will have a local minimum within constant number of probes.

Algorithm:

- Define function ***find_possible_min*** (root_subtree)
  - If root_subtree has no children
    - RETURN root_subtree
  - Probe the root_subtree to obtain x_r
  - Probe its two children
    - If both children's value x_1 and x_2 is larger than x_r
      - RETURN root_subtree
    - Else
      - RETURN whichever child whose value is less than x_r
- Set node1 = root_of _binary_tree
- Set node2 = ***find_possible_min*** (node1)
- While TRUE
  - If node2 == node1
    - Break and RETURN node2
  - Else
    - Set node1 = node2
    - Set node2 = ***find_possible_min*** (node1)
- End while

Explanation:

We will first analyze the root r of the whole tree:

If its two children are larger than itself, then the root will be the local minimum.

Otherwise, we will go to whichever child whose value is less than the root (call this r_s). In this case, if we take r_s as the root of the subtree, it will be exactly the same as the case for r as we discussed before. This is because we already know that r has value larger than r_s, so it suffices to see if r_s's children have value larger than r_s.

Now, by argument above, there is a chance to use to find an **internal node** r_s that is the local minimum, then we can stop.

If we fail to find an internal node that is a local minimum, then we necessarily reach a leaf of the tree. But since the leaf must be less than its parent and as the parent is the only node the leaf is connected with, then the leaf will be the local minimum.

Thus, we will be guaranteed to find a local minimum by the algorithm.

Complexity:

1. We will probe 3 nodes and then decide where to go for the next layer. This will take $O(1)$ time to run
2. We will repeat the process in (1) for each layer of the tree. Since there will be $O(\log n)$ layers, in total, we will complete this algorithm in $O(\log n)$ time

Question 6 (Exercise 6, Page 246):

Suppose the array A has length n and the indexing start from 1.

Algorithm:

- Set $\text{end} = n$
- Set $\text{middle} = \lceil n/2 \rceil$
- Set $\text{begin} = 1$
- Recursively do the following:
  - If $\text{end} == \text{begin}$
    - Return begin
  - If $\text{middle} == \text{begin}$ or $\text{middle} == \text{end}$
    - If $A[\text{begin}] > A[\text{end}]$
      - Return begin
    - Else
      - Return end
  - Compare $A[\text{begin}]$ with $A[\text{middle}]$
    - If $A[\text{begin}]$ is smaller
      - Set $\text{begin} = \text{middle}$
      - Set $\text{middle} = \lfloor (\text{end} + \text{middle})/2 \rfloor$
    - Else if $A[\text{begin}]$ is larger
      - Set $\text{end} = \text{middle}$
      - Set $\text{middle} = \lfloor (\text{begin} + \text{middle})/2 \rfloor$

Explanation:

The idea for this algorithm is that we can use one comparison to eliminate half of the positions in the array. Now, I will show that it actually works.

If the *middle* is larger than the first element, then it is necessary that the circularly shifted position has span more or equal of half of the list.

This is because the above statement indicates left half of the list is sorted and if the circular shift does not span the left half, the value at *middle* will be something near the head of the sorted list and thus should be smaller than the *begin,* which is a contradiction.

With the exact reverse argument, we see that if value at *middle* is smaller than the first element, then the circular shift does not span the left half of the list.

Thus, each iteration of the recursive call will guarantee to include the position where the circular shift is located in the sub-list defined by *begin* and *end.*

Notice we can regard this sub-list as another circularly shifted array as whichever part we excluded by tuning *begin* and *end* will be a sorted list. Thus, the above argument can apply recursively, until the sub-list's size is too small and we can no longer find a *middle* that is different from *begin* or *end.* In this case, we know that the list size is reduce to 2 or 1, and we will return the position of the larger element (as the circularly shifted position is always the largest element in the list).

Complexity:

1. For iteration, we do constant number of things to find which half of the list we should go to find k. This will take $O(1)$ time to run
2. Since we always divide the list into half after each iteration, then there could be in maximum $O(\log n)$ divides.

Thus, the algorithm will run with time complexity $O(\log n)$.