CS 180 Homework 4

Name: Wang, Zheng CS 180 Dis 1E

UID: 404855295 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 () and the time taken to finish () should both contribute to our decision-making process. Now, as large tends to persuade us to do the tasks later and large tends to force us to the task earlier, a good way is making decision based on .

Algorithm:

* For each task we are going to assign:
  + Calculate
* Sort the task in decreasing order based on
* Return the sorted list of tasks , which will be the optimal schedule

*Proof:*

We will show that for any other schedule , there exist some neighboring pair satisfying:

1. is scheduled before

(there must exist a neighboring (or consecutive) pair with the property above, otherwise, by a simple induction, we see that is sorted, and thus )

Then, if we swap them, we would result some that is no worse than original one.

Let the total weighted sum of be and this is equivalent to . For simplicity, we will just call the term to be .

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

To make a comparison between the two, we take . If we have , then we have , a contradiction with the assumption we made.

Thus, for any , we can continue doing the above without the risk of increasing the weighted sum and we will eventually get . Thus is the optimal schedule.

Complexity:

1. For each element, we calculate the correspond to , this will take to do it.
2. Sort the list based on will take to do it.
3. Output the schedule will take to do.

Thus, the overall time complexity of this algorithm will be

**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 contains some task , then the algorithm mentioned above will find a schedule no worse that.

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

Now the start of “today” and the start of “next day” will form the 24-hour interval of “Interval Schedule Problem” for both and .

Suppose that the first (as must be the same for and ) intervals are the same for and , then since will pick intervals that ends first, the interval in must ends later than the interval of . Thus, we will have no trouble substitute the of with the in without the risk of decreasing the total number of intervals in . We will call this schedule after substitution .

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

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

Complexity:

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

Thus, overall, we will need 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 strong inversions and the right-array has 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 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 of the right-array to be .
  + Initialize two pointers, each point to the beginning of left-array and right-array.
  + Initialize an integer
  + While neither pointer reaches the end
    - Compare the value of the two pointers
    - If left\_value right \_value
      * Increment the left pointer
    - If left\_value right\_value
      * Increment the left pointer
      * Take to be the number of elements from the left pointer (inclusive) to the end of the left-array
      * Set
  + End while
  + Return merged\_list with attribute
* Recursively divide the whole list into left-array of size and right-array of size 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 ;
* Recursively call ***merge*** to merge the divided list together until we get a sorted list . Get the value of of the resulting sorted list .

Explanation:

The basic idea is very similar to that of normal inversions. But since we are looking for cases such that and , we need to do another merge with the 2\*right-array.

Let the left array be and the right array be . 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 in the left array and every element in the right array has property . Then if we find , we are sure that , , …, .

Thus, we will find (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 . So, whenever we find an instance of that, we will increment by .

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 times
2. For each merge, in the worst case we are going to go through element to construct a sorted list and elements in order to count the number of strong inversions between the two list (basically, we are doing another merge). Thus, this will take time to run.

Thus, the overall complexity is 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 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 cards that are equivalent (let’s label these cards ) and is the majority of a pile of cards, then we have .

Now suppose towards contradiction that if we divide the whole pile into two, each with and cards, and and card , but is not the majority of either of the sub-piles of cards.

Then, and . So and . Now we have

A contradiction. Thus, the majority is preserved.

Algorithm:

* Recursively divide the cards into two piles with size and 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 of the pile to be that card
  + If the size of the pile is 2 and they are equivalent, set of the pile to be one of them
  + If the size of the pile is 2 and they are not equivalent, set 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 is NONE
      * Put two pile together to get the merged-card-pile
      * RETURN the merged-card-pile with set to NONE for next merge
    - If right pile has
      * Put two pile together to get the merged-card-pile
      * Use the card tester to test every card in the merged-card-pile against
      * If number of cards that are equivalent to is more than half of the size of the merged-card-pile
        + RETURN the merged-card-pile with set to for next merge
    - If the left pile has
      * Put two pile together to get the merged-card-pile
      * Use the card tester to test every card in the merged-card-pile against
      * If number of cards that are equivalent to is more than half of the size of the merged-card-pile
        + RETURN the merged-card-pile with set to for next merge.
    - Put two pile together to get the merged-card-pile
    - RETURN the merged-card-pile with set to NONE for next merge

Explanation:

Firstly, we will show if there is majority 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 ) 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 , and when we divide this sub-pile, same thing will hold, and we get a “sub-sub-pile” with majority . 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 .

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

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 NONE

Then, certainly we will get NONE.

1. At least one of the sub-piles have that is not NONE

Then, by checking the 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 to run
2. We will do 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 and . This will take time to run

Thus, the overall time complexity is .

**Question 5 (Exercise 6, Page 246):**

To do this problem in 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
  + Probe its two children
    - If both children’s value and is larger than
      * RETURN root\_subtree
    - Else
      * RETURN whichever child whose value is less than
* 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 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 ). In this case, if we take as the root of the subtree, it will be exactly the same as the case for as we discussed before. This is because we already know that has value larger than , so it suffices to see if ’s children have value larger than .

Now, by argument above, there is a chance to use to find an **internal node** 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 time to run
2. We will repeat the process in (1) for each layer of the tree. Since there will be layers, in total, we will complete this algorithm in time

**Question 6 (Exercise 6, Page 246):**

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

Algorithm:

* Set =
* Set =
* Set = 1
* Recursively do the following:
  + If ==
    - Return
  + If == or ==
    - If >
      * Return
    - Else
      * Return
  + Compare ] with
    - If is smaller
      * Set =
      * Set =
    - Else if is larger
      * Set =
      * Set =

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 . This will take time to run
2. Since we always divide the list into half after each iteration, then there could be in maximum divides.

Thus, the algorithm will run with time complexity .