CS 180 - Algorithms and Complexity
Homework 3

Jayant Mehra
Discussion 1D

November 8, 2018

Solution 1. This is the solution to problem 13 on page 194 of the textbook.

1. sort the (w_i, t_i) pairs in descending order of w_i/t_i
2. Execute jobs in that order

Algorithm Proof of Correctness Let our solution be represented by G and the optimal solution be represented by O.

For a single job, both solutions are equal.

For n jobs, let the solutions be equal up to k jobs. let i represent the greedy different job and j represent the one in O. We know the $w_j/t_j \leq w_i/t_i$. Moreover, the adjacent job in O will have this property. We need to show that swapping them will not increase our amount.

Let the time up till now be T. In the optimal solution, before the swap, the partial sum was $w_j * (T + t_j) + w_{j+1} * (T + t_{j+1} + t_j)$. After the swap, it becomes, $w_j * (T + t_j + t_{j+1}) + w_{j+1} * (T + t_{j+1})$. But since, w_j/t_j is less than or equal to w_{j+1}/t_{j+1}, this means that the new sum is less than or equal to the old sum. Therefore, we have shown that the sum does not increase and we can swap all such pairs to obtain G from O and thus our solution is optimal.

Algorithm Analysis After the O(nlogn) sort, it is a linear scan. Thus, the algorithm is bounded by polynomial O(n^2) run time.

Solution 2. This is the solution to problem 17 on page 197 of the textbook.

1. for $j \in$ jobs that cross noon
2. remove all jobs that overlap with j

3. sort in ascending order of finish times (we have removed the cyclic nature of the problem) (we need to do this only the first time)
4. solve using the greedy algorithm for maximum interval scheduling
5.
6. if size of current scheduling is greater than global maximum
7. change the global maximum

Algorithm Analysis We sort the list once which is O(nlogn). But for each interval that crosses noon, we produce an ordering in O(n), thus, our total run time becomes $O(n^2)$.

Solution 3. This is the solution to problem 2 on page 246 of the textbook.

The problem is similar to the inversion problem using a variant of merge sort we did it in class. The only difference will be in the merge and count step. Instead of counting inversions and merging at the same time, we will will first find the significant inversions and then merge. The new part in the merge and count routine would look something like:

Given two sorted arrays, we maintain a pointer to each array. If the element on the right is significantly smaller than the element on the right, we increase our significant inversion count by the number of remaining elements on the left including the current element and move our right pointer ahead by one step. If it is not significantly smaller, then we can move our left pointer ahead by one element since it would never be significantly greater than any other element on the right since it is sorted in ascending order.
Thus, we explore each element only once and therefore our runtime is still O(nlogn).

Solution 4. The following is the solution to problem 3 on page 246 of the textbook.

1. if n = 1
2. return True, element[1]
3.
4. lpresent, 1card = divide(1 ... n/2)
5. rpresent, rcard = divide(n/2+1 ... n)
6.
7. if lpresent = True $\wedge$ rpresent = True
8. if lcard is equivalent to rcard
9. return True, lcard
10. count equivalency of lcard in right and equivalency of rcard in left
11. if any appears more than half
12. return True, that card
13.
14. if lpresent = True
15. count its occurrence in right
16. if total occurrence greater than half

```
17.         return True, lcard
18. if rpresent = True
19.     count its occurrence in left
20.     if total occurrence greater than half
21.         return True, rcard
22.
23. return False, NULL
```

Algorithm Explanation The problem is similar to the majority element problem and can be also be done in O(n) time using Moore's algorithm covered in class. But I use a divide and conquer approach to solve it in the required O(nlogn) time. In the following explanation I mean greater than n/2 by majority.

If there exists a majority element, it should exist in either the left half, right half, or both. If both halves have a majority card and they are card, then that is the majority card. If they are not equivalent, we need to count the find the number of equivalent cards in the other half for both majority card. Whichever card crosses majority is the majority card.
The other case is that only the left or right has majority. In that case we just count the cards equivalent to the majority card in the non majority half and if the total count crosses majority for that n, we say that majority exists.

Algorithm Analysis We divide the problem set into size of two until we hit 1 element arrays. Thus, there are logn levels. At the conquer step, we just do a linear pass. Thus, the total run time is O(nlogn) as required.

Solution 5. The following is the solution to problem 6 on page 248 of the textbook.

```
1. findLocalMinimum(root)
2.     if root is a leaf
3.         return root
4.
5.     if both children exist and root is smaller than both
6.         return root
7.
8.     if root's left child exists and is smaller than the right child
9.         return findLocalMinimum(root's left child)
10.
11.    return findLocalMinimum(root's right child)
```

Algorithm Explanation At each step we explore the child which is smaller than the current root. If that child is smaller than its children, we return it. Or if it is a leaf, we return it since it was smaller than its parent anyway. Thus, we always find the local minimum with

this recursive algorithm.

Algorithm Analysis At each step we divide our search space by 2 (since we either explore the left half or the right half; never both). Thus, our algorithm just make logn *probes*.

Solution 6. We need to either find the element which is in elements both of which are larger to it or the one which is in between elements both of which are smaller to it. Or the element can be at the end or start. If it is at the end and is smaller than the start, then it is the element we are looking for. Similarly, if the element is at the start and is greater than the last element, then that is the element we are looking for.
We perform a modified binary search.

```
1.  findElement(low, high)
2.      if high < low
3.          Not Rotated or Rotated full circle
4.
5.      if low = high and the element satisfies the above conditions
6.          return the number of elements to the left of it. If it is the first element, then 1.
7.
8.      mid = (high + low)/2
9.      if mid satisfies the conditions
10.         return the number of elements to the left of mid
11.
12.     if element at mid is less than the low element
13.         return findElement(low, mid)
14.     return findElement(mid, high)
```

Algorithm Explanation We just use the definition of the "pivot" element to recursively find it. If we find the pivot element, then the elements to the left of it is k. We divided our search space by the fact that at any moment in our search, if there does exist a rotation, then the mid element will either be greater than the last element or smaller than the first element. Thus, we search in the right direction.

Algorithm Analysis On each step, we divide our search space by 2. Thus, the algorithm runs in O(logn).