CS535 Design and Analysis of Algorithms - Assignment 4

Batkhishig Dulamsurankhor - A20543498

October 27, 2024

1. To find the contradiction, we need to find all pairs of patients (i, j) with $(h_i > h_j)$ and $(w_i < w_j)$ or $(h_i < h_j)$ and $(w_i > w_j)$. A trivial approach would be comparing each of patient's weight and height with every other's. This algorithm works but not efficient and takes $O(n^2)$.

   Instead, we can solve the problem with the following algorithm:

   - Sort the patients by their heights in ascending order using algorithms like merge sort. This step takes $O(nlogn)$.
   - Now we have list of patients ordered by their heights. If there is no contradiction here, weights should also be in ascending order. Merge sort has a really convenient pattern here to detect any contradiction in the hypothesis. When merging, everytime there when there is a $w_i > w_j$ where $h_i < h_j$, it chooses w_j first and then w_i, ensuring the correct order. We can take advantage of this logic to count every contradiction pair. In short, we can sort the list again by their weights using merge sort and instead of just merging, we count the number of times when have to choose from the right list first before left list is empty. After this sort, we are left with the result. This step also takes $O(nlogn)$.

   In total, the time complexity is $O(nlogn + nlogn) = O(nlogn)$.

   Algorithm pseudocode:

```
merge(P, L, M, R):
  result=0
  i=L, j=M
  replacement=[]
  while i<M && j<R:
    if P[i].weight<=P[j].weight:
      replacement.add(P[i])
      i++
    else:
      result+=M-i+1
      replacement.add(P[j])
      j++
  P[L,R+1]=replacement
  return result

merge_sort(P, L, R):
  result=0
  if (L<R):
    result+=merge_sort(P,L,(L+R)/2)
    result+=merge_sort(P,(L+R)/2+1,R)
    result+=merge(P,L,(L+R)/2,R)

main(P):
```

```
sort(P) by height;
result=merge_sort(P, 0, length(P)-1)
return result;
```

2. The tree T is a perfect binary tree since it is a complete binary tree with $n = 2^d - 1$ nodes. For a node to be a local minimum, all its neighbors' values are more than its value. Because we are given a binary tree, there are at most 3 neighbors, a parent and two children. A root has 2 children neighbors and a leaf node has 1 parent neighbor.

   We can achieve $O(logn)$ complexity with the following approach:

   - We start from the root of T.
   - We check the value of each of its neighbors and compare them with the current node's value.
   - Then we go to the smallest value node.
   - If the current node is the smallest, we found a local minimum.

   This algorithm is correct because we only go down the tree and never go back up. Going back up to a parent means $x_{parent} < x_{child}$. However, we are moving down the tree only when the parent's value is more than at least one of its children's value. We would never go down to the child if $x_{parent} < x_{child}$ was the case in the first place.

   This means in the worst case scenario, the length of the traversal is only the depth of the binary tree d. The time complexity of this algorithm is therefore $O(d)$ and we know that $n = 2^d - 1$ so $d = log(n+1)$. This makes the time complexity $O(log(n+1)) = O(logn)$.

   Algorithm pseudocode:

```
local_min(T):
  if (T is leaf node):
    return T
  if (T.val<T.left.val && T.val<T.right.val):
    return T

  if (T.left.val<T.val && T.left.val<T.right.val):
    return local_min(T.left)
  else:
    return local_min(T.right)

main(T):
  return local_min(T)
```

3.

4.

5.