Homework #7

You should try to solve these problems by yourself. I recommend that you start early and get help in office hours if needed. If you find it helpful to discuss problems with other students, go for it. **You do not need to turn in these problems. The goal is to be ready for the in class quiz that will cover the same or similar problems.**

Problem 1: Greedy Ski Distribution

Consider the problem of matching a set of available skis to a set of skiers. The input consists of n skiers with heights $p_1, \ldots p_n$ and n sets of skis with heights $s_1, \ldots s_n$. The problem is to assign each skier a ski to minimize the average difference between the height of a skier and his or her assigned set of skis. That is, if the i^{th} skier is given the $\alpha(i)^{th}$ pair of skis, then you want to minimize:

$$\frac{1}{n}\sum_{i=1}^{n}|p_i - s_{\alpha(i)}|$$

Design a greedy algorithm that solves this problem. Prove that your algorithm is correct.

Solution

The greedy choice is to give the shortest ski to the shortest skier, give the second shortest ski to the second shortest skier, and so on, until we run out of skis and skiers.

The running time is $O(n \log n)$. We have to sort the skis and skiers. Once the lists are sorted, the greedy pairing algorithm takes $O(n)$ time.

The algorithm that makes the greedy choice is optimal. Suppose it wasn't. That is, suppose there was a different optimal solution that did not pair the skiers with skis in the same way as the greedy solution. This solution is a list of pairs $T = \{(p_1, s_{j(1)}), (p_2, s_{j(2)}), \ldots, (p_n, s_{j(n)})\}$. The greedy solution is the list of pairs $G = \{(p_1, s_1), (p_2, s_2), \ldots, (p_n, s_n)\}$. Let p_i be the first skier assigned different skis in T than in G and let s_j be the skis assigned to p_i in T. In T, let s_i be the skis assigned to p_k. We create a solution T' that replaces (p_i, s_j) and (p_k, s_i) in T with (p_i, s_i) and (p_k, s_j), respectively. We claim that we made T' no *worse* than T and may have made it better. The total cost of T' is given by:

$$Cost(T') = Cost(T) - 1/n((|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|))$$

There are six cases because $p_i \leq p_k$ and $s_i \leq s_j$; for each, we must show that $((|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|)) \geq 0$:

1. $p_i \leq p_k \leq s_i \leq s_j$. Then there was no change in the value of T to T':

   $$(|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|) = (s_j - p_i) + (s_i - p_k) - (s_i - p_i) - (s_j - p_k) = 0$$

2. $p_i \leq s_i \leq p_k \leq s_j$. Then the value of T' is less than the value of T (a contradiction to T being optimal:

   $$(|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|) = 2(p_k - s_i) \geq 0$$

3. $p_i \leq s_i \leq s_j \leq p_k$. Then the value of T' is less than the value of T (a contradiction to T being optimal:

   $$(|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|) = 2(s_j - s_i) \geq 0$$

4. $s_i \leq s_j \leq p_i \leq p_k$. Then there was no change in the value of T to T':

   $$(|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|) = 0$$

5. $s_i \leq p_i \leq s_j \leq p_k$. Then the value of T' is less than the value of T (a contradiction to T being optimal:

   $$(|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|) = 2(s_j - p_i) \geq 0$$

6. $s_i \leq p_i \leq p_k \leq s_j$. Then the value of T' is less than the value of T (a contradiction to T being optimal:

   $$(|p_i - s_j| + |p_k - s_i|) - (|p_i - s_i| + |p_k - s_j|) = 2(p_k - p_i) \geq 0$$

Problem 2: Huffman Coding

Consider the binary tree that is constructed in the Huffman Coding procedure. Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

Solution

Proof by contradiction. Suppose not. That is, suppose there is an optimal prefix code represented by a non-full binary tree. Then at some place in this tree, there is a node that has exactly one child. Then remove the parent and replace it with the child. Assume that the child codes for some element that is actuall used, the code for this element is one shorter than in the "optimal" tree. So the new tree represents a more optimal code, which is a contradiction.

Problem 3: Fibonacci

The Fibonacci function is defined as follows:

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \text{ for } n > 2 \end{aligned}$$

1. Assume you implement a recursive procedure for computing the Fibonacci sequence based directly on the function defined above. Then the running time of this algorithm can be expressed as:

   $$T(n) = T(n-1) + T(n-2) + 1$$

   Choose from the following asymptotic bounds the one that best satisfies the above recurrence and explain your selection:

   i $T(n) = O(n)$
   ii $T(n) = O(n^2)$
   iii $T(n) = \Omega(c^n)$, for some constant c
   iv $T(n) = \Theta(n^n)$

   Solution

   $T(n) = \Omega(c^n)$, for some constant c. Because the pure recursive procedure creates duplicate subproblems, and the number of subproblems solved in this way grows exponentially.

2. What specifically is wrong with your algorithm? (i.e., what observation can you make to radically improve its running time?)

   Solution

   It solves exponentially many subproblems, even though it has only n unique subproblems to solve. That is, it is duplicating work.

3. Give a memoized recursive algorithm for computing $F(n)$ efficiently. Write the recurrence for your algorithm and give its asymptotic upper bound.

Solution

The memoized version is based on the following recurrence: $F[i] = F[i-1]+F[i-2]$. (Which is the same recurrence as stated above.) However, filling in the solution with memoization, we shortcut the direct recursive approach by performing a table lookup before calling a new subproblem:

MFIB(i)

```
1  F[0] = F[1] = 1
2  if F[i − 1] is empty
3      then F[i − 1] = MFIB(i − 1)
4  if F[i − 2] is empty
5      then F[i − 2] = MFIB(i − 2)
6  return F[i − 1] + F[i − 2]
```

$T(n) = T(n-1) + T(n-2) + c$. But this isn't useful for evaluating its asymptotic upper bound. Instead, by aggregate analysis, each entry in F is computed only once; the remaining work takes constant time. Therefore, the overall running time is $O(n)$.

Problem 4: Divide and Conquer

Suppose you are given a sorted sequence of *distinct* integers $\{a_1, a_2, \ldots a_n\}$. Give an $O(\log n)$ algorithm to determine whether there exists an index i such that $a_i = i$. For example, in $\{-10, -3, 3, 5, 7\}$, $a_3 = 3$; there is no such i in $\{2, 3, 4, 5, 6, 7\}$. Write the recurrence for your algorithm and show that its recurrence solves to $O(\log n)$ (e.g., using the Master Method, a recursion tree, or the substitution method).

Solution

Let MATCH(i, j) return true if $a_i = i$ or if $a_{i+1} = i + 1 \ldots$ or if $a_j = j$. We call this function at the beginning with MATCH$(1, n)$.

MATCH(i, j)

```
 1  if i > j
 2      then return false
 3  if i = j
 4      then if a_i = i return true
 5      else  return false
 6  if i < j
 7      then m ← ⌊(i+j)/2⌋
 8       if a_m = m then return true
 9       if a_m > m then return MATCH(i, m − 1)
10       if a_m < m then return MATCH(m + 1, j)
```

The recurrence for the above is derived as follows. It generates one subproblem of size $n/2$. The other work (outside of the recursive call) takes $O(1)$ time. So the recurrence is $T(n) = T(n/2) + O(1)$. Using the master method, $a = 1$, $b = 2$, $l = 0$, $k = 0$. This is case 2 ($\log_b(a) = 0 = l$). So the solution is $O(n^l(\log n)^{k+1}) = O(\log n)$.