Stefanie Shidoosh

CS 180: HW #6

1. Given an array A[1..n] of n numbers, we can calculate the maximum in A only by comparing numbers. Give an algorithm that finds the maximum and second maximum using exactly n + O(log n) comparisons.

Find the maximum
- Recursively compare each element with every other element not yet visited in A
- If the current element is greater than the compared element, continue
  - Append compared element to array A_0
  - If true for all elements, the current element is the maximum
- Else, append current element to array A_0
  - Move on to next element

Find the second maximum
- Recursively compare each element with every other element in A_0
- If the current element is greater than the compared element, continue
- Else, move on to next element

To find the maximum in A takes $O(logn)$ comparisons as with each iteration, one element is guaranteed to be eliminated from potential comparisons. To find the second maximum in A (the maximum in A_0) is takes n comparisons as the potential (second) maximum must be compared to the elements not yet visited, so in the worst case the (second) maximum was $A_0[1]$, and thus would need to be compared to at most n elements. Thus, the algorithm in total takes $n + O(logn)$ comparisons.

Stefanie Shidoosh

2. Given a positive integer C, find the smallest number of coins from A_n that add up to C, given that an unlimited number of coins of each type are available.

a. Suppose $a_1 = 1$. A greedy algorithm will make change by using the larger coins first, using as many coins of each type as it can before it moves to the next lower denomination. Show that this algorithm does not necessarily generate a solution with the minimum number of coins.

One example of when the greedy algorithm fails: if C = 30, and the denominations are {1, 15, 25}, the greedy algorithm will provide one 25 and 5 1's, for a total of 6 coins. However, the optimal minimum number of coins is 2 by proving 2 15's instead.

b. Show that if $A_n = \{1, c, c^2, ..., c^{n-1}\}$, $c \geq 2$, then the greedy algorithm always gives a minimal solution for the coin-changing problem (example: {1, 2, 4, 16}).

The greedy algorithm works when for all a_i in A_n, $(a_i + a_1) \geq 2a_{i-1}$.
This will always be the case for A_n's properties.

c. Design an $O(nC)$ algorithm that gives a minimum solution for the general case of A_n.

Given A_n and C, recursively traverse A_n
If C is 0, no more coins required
If C > 0, then the minimum number of coins needed to make C is
the minimum number of coins needed to make 1 + C - $A_n[i]$
recursive step until solution is reached (C=0).

The time complexity of this algorithm is $O(nC)$ as it must traverse n integers in A_n for the value of C times in the worst case (as in, say C = 25 and it needed 35 pennies, $a_1 = 1$, and it took traversing the entirety of A_n to determine this).

Stefanie Shidoosh

3. Given n items {1,...,n}, and each item i has a nonnegative weight wi and a distinct value vi. We can carry a maximum weight of W in a knapsack. The items are blocks of cheeses so that for each item i, we can take only a fraction xi of it, where $0 \leq xi \leq 1$. Then item i contributes weight $xiwi$ and value $xivi$ in the knapsack. The continuous knapsack problem asks you to take a fraction of each item to maximize the total value, subject to the total weight does not exceed W. The original knapsack problem is equivalent to say xi is either 0 or 1 depends whether the item is in or not. Give an $O(n)$ algorithm for the continuous knapsack problem.

Find the value/weight ratio for all n items, append to set S
Employ a median finding algorithm to find the optimal median
Pick an item, i, from {1,…n} arbitrarily
Partition the items' ratios into three sets, relative to set S
s_0: all items whose ratios are less than i's ratio
w_0: sum of these items' weights
s_1: all items whose ratios are equal to i's ratio
w_1: sum of these items' weights
s_2: all items whose ratios are greater than i's ratio
w_2: sum of these items' weights
if $w_0 > W$, recurse on s_0, return solution
else, while the knapsack does not exceed weight, add items from s_1
if knapsack is full, return s_0 and items added to s_1
else, $W = W - (w_0 + w_1)$
recurse on s_2, return resulting items
and the union of s_0 and s_1

The time complexity for this algorithm is $O(n)$ as the recurrence is $T(n) = T\left(\frac{n}{2}\right) + O(n)$.

Stefanie Shidoosh

4. Recall the problem of finding the number of inversions. We are given an array $A[1..n]$ of n numbers, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $A[i] > A[j]$. The problem is to count the number of inversions in A. Let's call a pair (i, j) is a *significant inversion* if $i < j$ and $A[i] > 2A[j]$. Give an $O(n \log n)$ algorithm to count the number of significant inversions in A.

Recursively divide A into 2 sets
Sort each set
Initialize count to 0
For each element in the sets
    append the min(set1[i], set2[j]) to result set, S
    if set1[i] > set2[j])
        increment the count in addition number of elements remaining in A
        continue comparing for next element for set2, j++
    else, continue comparing for next element in set1, i++
Append remaining elements in A to S
Return S and count
The count will be the number of significant inversions in A.

The time complexity of this algorithm is O(nlogn) as the most costly operation is the (merge) sorting, which has a run time of O(nlogn).