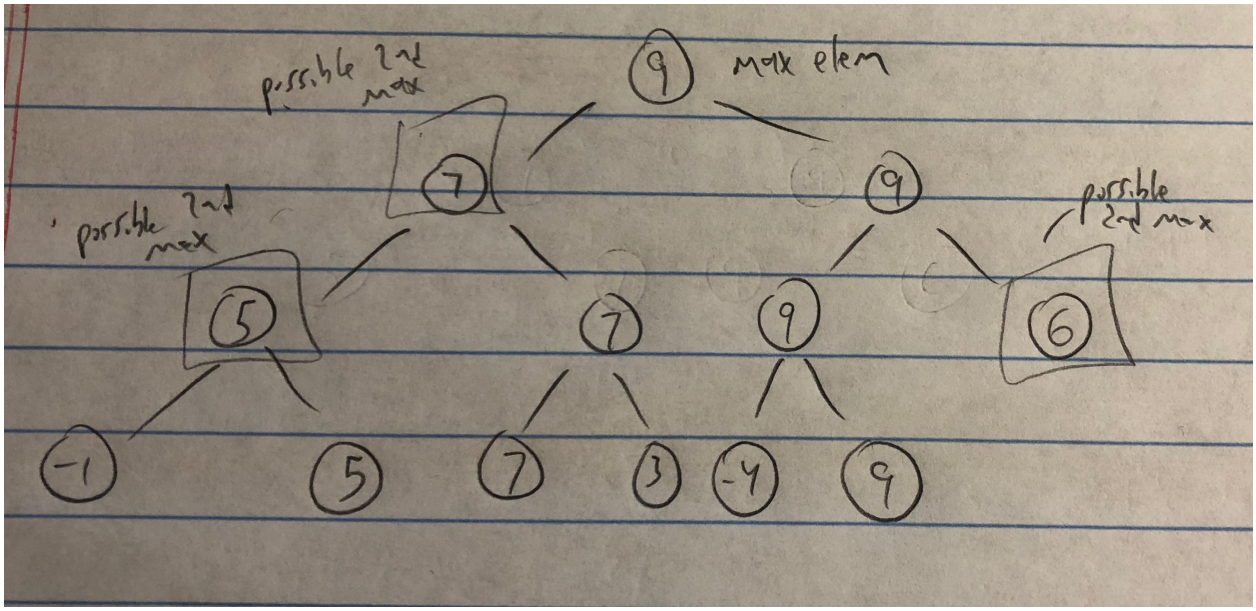
Jason Less
404-640-158
CS180

CS180 Homework 6

Question 1

The max and second max elements of an array can be found the brute force way by iterating one pass through the array and recording the two largest elements found (replacing elements along the way as new max elements are found). While this algorithm has an O(N) time complexity, it consists of 2n comparisons. To find the two max elements only using N + O(log N) comparisons, a binary tree of height log N can be constructed. The binary tree is constructed by continuously dividing the array in half until only two elements remain. At that point, compare the two elements, and copy the larger of the two to be the parent of the given subtree. After constructing the binary tree, the largest element will be at the root of the tree. The second largest element can be found by comparing all sibling nodes of the elements that sifted up through the tree. These sibling nodes will be placed in some data structure (e.g. an array) and will consist of log N elements (i.e. the height of the tree). Finally, just do log N comparisons of the elements in the array to get the second largest element. This is guaranteed to find the second largest element as the sibling nodes of the value that sifts up the tree are the largest in their respective subtrees.

As an example, take the array [-1, 5, 7, 3, -4, 9, 6]. The max element is clearly 9, while the second max (7) can be found by taking the sibling nodes of the elements that sift up through the tree (i.e. 7, 9) (not counting the bottom level) and placing them in an array [5, 6, 7].



Jason Less
404-640-158
CS180

Algorithm

```
twoMaxElems(A):
  arr = []
  maxElem = findMax(A, arr)
  secondMaxElem = max(arr)
  return (maxElem, secondMaxElem)

findMax(A, arr):
  if |A| = 1 then
    return A0
  if |A| = 2 then
    return max(A0, A1)

  n = len(A)
  L = findMax(A[0..n/2], arr)
  R = findMax(A[n/2+1..n], arr)
  if (L > R) then
    add R to arr
    return L
  else
    add L to arr
    return R
```

Jason Less
404-640-158
CS180

Question 2

(a)

A greedy algorithm will not work for any coin set; and in fact, only works for canonical coin sets (e.g. US coins => 1, 5, 10, 25). As an example that shows greedy doesn't work for all coin sets take [1, 3, 4] with C = 6. Greedy will use coins 4, 1, 1, which consists of three coins, while the optimal solution will use coins 3, 3, which only consists of two coins.

(b)

The reason that greedy works for this coin set boils down to the general idea that if the remainder is greater than C and I am only allowed to use some coins c_0 and c_1 s.t. $c_0 < c_1$, I will use as many c_1 coins as possible before considering to use c_0. As an example, say $c_0 = 1, c_1 = 5, and\ C \geq 5$. Then for any C, at most I could use 4 of c_0, as any number larger than 4 would be replaced by 1 of c_1 (which would reduce the total number of coins used by 4). In more general terms, if we have c_0 and c_1, then at most we can use C – 1 c_0 coins as any number larger than C c_0 coins could just be replaced by at least one coin of denomination c_1.
Similarly, when considering coins c^{n-2} and c^{n-1}, at most we can use C – 1 c^{n-2} coins, as any number larger than C c^{n-1} coins would just be replaced by at least 1 c^{n-1} coin. Thus, if the remainder is greater than c^{n-1}, then use as many c^{n-1} coins as possible before considering to use any c^{n-2} coins.

(c)

The coin change for the general case can be solved using dynamic programming. This is because the problem is of optimal substructure. Given that we have an optimal solution for making change of C, and we know that the optimal solution uses a coin whose value is c, where the optimal solution uses k coins. The claim is that the optimal solution for the problem of size C must contain an optimal solution for the problem C – c. Using the cut and paste argument, we can clearly see that k – 1 coins will be used in the solution for C – c. If such a solution existed to the C – c problem with fewer than k – 1 coins, then we would use this solution to produce the solution to the C problem that uses fewer than k coins, which contradicts the optimality of the solution.

Since we have optimal substructure, we can try to use dynamic programming. Given c[j] = the minimum number of coins to make val j and coins have denomination $d_0, d_1, \dots, d_n$. If the optimal solution for the problem of making change of value j using a coin of denomination d_i, then c[j] = 1 + c[j - d_i]. The base case is c[j] = 0 for j <= 0, while the recursive property:

$$c[j] = \begin{cases} 0\ if\ j \leq 0 \\ 1 + \min\{c[j - d_i]\}\ for\ 1 \leq i \leq k\ if\ j > 1 \end{cases}$$

Using this property, we can build up c[j] values in order of increasing j using a 1D table.

Jason Less
404-640-158
CS180

Algorithm

```
CoinChange(coins, C)
  n = size of coins
  dp[1..C] where each dp[i] = inf
  for j = 1 to n
    for i = 1 to n
      if i >= coins[j] and dp[i - coins[j]] + 1 < dp[i]
        dp[i] = 1 + dp[1 - coins[j]]
  return dp[size of dp - 1]
```

This algorithm clearly runs in O(nC) time.

Jason Less
404-640-158
CS180

Question 3

The fractional (or continuous) knapsack problem can be solved using a greedy approach by taking the ratio of $\frac{v_i}{w_i}$ for each item and then sorting them in decreasing order. Then, add items greedily in their entirety starting with the one with the largest ratio as long as there is enough space. When there is not enough space to fit an entire item, then take a fraction of the item. This algorithm runs in O(N log N) time complexity due to sorting at the beginning. However, if we can remove the sorting the algorithm can run in linear time. This can be achieved by using a modified quick select algorithm to find the median of medians. At each step, a good pivot is chosen to achieve this linear time complexity.

Algorithm
fractionalKnapsack(vals, wts, W)
    compute ratios of $\frac{v_i}{w_i}$ for each element and store in arr ratios
    return helper(ratios, W)

helper(ratios, W)
    Choose elem r from ratios at random
    $R_1 = \left\{\frac{v_i}{w_i} > r, for\ 1 \le i \le n\right\}, W_1 = \sum_{i \in R_1} w_i$
    $R_2 = \left\{\frac{v_i}{w_i} = r, for\ 1 \le i \le n\right\}, W_2 = \sum_{i \in R_2} w_i$
    $R_3 = \left\{\frac{v_i}{w_i} < r, for\ 1 \le i \le n\right\}, W_3 = \sum_{i \in R_3} w_i$
    if $W_1 > W$ then
        recurse on R_1 and return computed solution
    else
        while there is space in knapsack and R_2 not empty
            add elems from R_2
        if knapsack is full
            return items in R_1 and items just added from R_2
        else
            reduce knapsack total capacity by $W_1 + W_2$
            recurse on R_3 and return items in $R_1 \cup R_2$ and the items returned from recursive call

Jason Less
404-640-158
CS180

Question 4

The problem of counting significant inversions is largely similar to the original problem of counting inversions, except for a small difference in the merge step, in which the merge step is divided into two parts: (1) count significant inversions and (2) merge the two arrays. The reason that this step has to be split into two steps is because we need to check all comparisons when counting the inversions, and can't just skip over the remaining elements in A if the condition is satisfied.

As an example, take the array [5, 3, 1, 4, 2], where A = [1, 3, 5] and B = [2, 4]. When just counting inversions, where $a_i = 3$, and $b_j = 2$, we conclude that this is an inversion, and also conclude that (5, 2) is an inversion. Then, we advance b_j to 4. For significant inversions, where $a_i = 3$, and $b_j = 2$, this would not be a significant inversion, so we add 2 to our output array, and advance b_j to 4. However, this would result in missing the significant inversion (5, 2). Therefore, we will count inversions first, and then merge the two sorted arrays into one after.

Algorithm

```
Sort-And-Count(L):
  if |L| = 1 then no inversions
  else
    n = len(L)
    (rA, A) = Sort-And-Count(L[0..n/2])
    (rB, B) = Sort-And-Count(L[n/2+1..n])
    (rL, L) = Merge-And-Count(A, B)
  return (r = rA + rB + rC, L)

Merge-And-Count(A, B):
  Maintain curr ptrs i and j into each list, init to front elems
  Maintain cnt of significant inversions
  while both lists are not empty:
      if A[i] > 2 * B[j] then
        incr. cnt by num elems remaining in A
        j++
      else
        i++

  C = []
  reset i and j to start of the two lists
  while both lists are not empty:
    if A[i] > B[j] then
      append B[j] to C
      j++
    else
      append A[i] to C
      i++
  once one list is empty, append the remainder of the other list to C
  return (cnt, C)
```