Hermmy Wang
704978214
Discussion 1B

Homework 6

1. In order to find the largest two elements in a set in n+O(logn) comparisons, we need to create a list of candidates for the second maximum. In this way, we do not need to do extra useless comparisons between numbers that will be ignored later. First, we will use a divide-and-conquer technique to divide the arrays into halves and find the maximum in each subarray. When we have found the maximum for the left subarray and the right subarray, we may eliminate the candidates of the winner's subarray. For example, if q1 > p1, then we can eliminate p2 and all the candidates less than p2 since we are sure that the second maximum is either p1 or a candidate of q1. The first element of the candidates array keeps track of the number of candidates. and the second number stores the first maximum. Once we have find a list of the candidates, the second maximum can be found by looking for the max starting from the third item in the candidates.

```
int[] find_max(A, start_i, end_i):
  if start_i = end_i
    candidates[0...n]
    candidates[0] <-- 1
    candidates[1] <-- A[start_i]

  candidates_1[] <-- find_max(A, 1, n/2-1)
  candidates_2[] <-- find_max(A, n/2, n)

  if candidates_1[1]  > candidates_2[1]
    candidates_1[0] <-- candidates_1[0] + 1
    candidates_1[candidates_1[0]] <-- candidates_2[1]
    return candidates_1[]
  else
    candidates_2[0] <-- candidates_2[0] + 1
    candidates_2[candidates_2[0]] <-- candidates_1[1]
    return candidates_2[]
```

```
int find_second_max(A)
```

```
cand <-- find_max(A, 1, n)
second_max <-- find_max(cand+2, 2, cand[0])
return second_max[1]
```

find_two_max(A)

```
max_1 <-- find_max(A, 1, n)
max_2 <-- find_second_max(A)
```

Time complexity:

find_max() does n comparisons between n elements. find_second_max() does comparisons among all elements in candidates[]. The number of candidates is at most logn, because the number of candidates increments once upon every division of the array. Therefore, a linear traversal of candidates[] results in O(logn) comparisons. The total number of comparisons is n+O(logn).

2(a).

Suppose we have three types of coins: 11, 9, 1, and we want to a total value of 40.

If we use the greedy algorithm, we will yield coin types of 11, 11, 11, 1, 1, 1, 1, 1, 1, 1, a total number of 10 coins. However, the optimal solution is 11, 11, 9, 9, where we do not use as many larger coins as possible.

The greedy algorithm does not give the optimal solution in the case that after choosing as many larger coins as possible, the value left is less than the next denomination. In order to reach the same target value, choosing the next largest possible denomination will result in a higher number of coins.

Suppose the greedy algorithm uses n coins, and the optimal solution uses m coins.

$$a_1 + (n-1)a_3 = ma_2$$
$$a_1 + na_3 - a_3 = ma_2$$

We know that $a_1 < a_2 < a_3$; therefore, $\frac{a_2}{a_3} > 1$ and $\frac{a_1 - a_3}{ma_3}$ is positive:

$$\frac{n}{m} = \frac{a_1 - a_3 + ma_2}{ma_3} = \frac{a_2}{a_3} + \frac{a_1 - a_3}{ma_3} \geq 1$$

Therefore, n >= m. The greedy algorithm will not always yield the optimal solution.

2(b).

In the given case, the maximum number of each type is (c-1), since choosing a type for the c^{th} time is the same as choosing one higher denomination.

For example, A_n = 1, 3, 9, 27, ... and C = 62.

Suppose the greedy algorithm chooses two [27]s, and it is not an optimal selection since the solution seems to skip the next coin type [9] because there are only 8 left in C now. The maximum number of coin type [3] we can choose is 2. The current number of coins is n+1+2 = n+3.

Then, suppose in the optimal selection, we are not choosing the last [27]. In order to fill the position of [27], we need at least three [9]s. The current number of coins is n+3. Next, the maximum number of coin type [3] is two. Thus, reaching the same amount costs this 'optimal' solution n+5 coins.

Therefore, we have reached a contradiction. The greedy algorithm yields better result than the optimal solution.

In a generalized case, assume C=(c-1)a_1+(c-1)a_2+(c-1)a_3. Let's say the greedy algorithm makes a mistake on a_1, then the corrected version will be (c-2)a_1+ca_2+(c-1)a_2+(c-1)a_3. Since we take out one a1, we add in c a_2 which yields the same value as a_1 with the least number of coins. The total number of coin in the greedy algorithm is 3(c-1), while the total number of coins in the corrected solution is 4(c-1). Therefore, the greedy algorithm has the minimal number of coins.

2(c).

Suppose n is the set of n types of coins and C is the expected value. We first initialize two arrays of size C. ncoins[i] holds the number of coins needed at the current value i, and coin_type[1...i] holds the types of coins needed at the current value i.

At every value of C, we compare the result of adding every type of coins to the current total number of coins. If adding a coin will result in a lower total number, we update the current number of coins. After iterating all the coin types, we have the best option for the current value. ncoins[value] will hold the current number of coins needed, and coin_type[value] will hold the latest coin added to the solution.

In the end, we push the coin types in coin_type[] to a solution array. Every time we add a coin, decrement the value of the coin from the total value. The solution is finished when the total value is added.

```
int[] coin_change(C, n)
   ncoins[C] <-- ∞
   coin_type[C] <-- ∞

   for value from 1 to C:
      curr_coin <-- null
      curr_n <-- ∞
      for coin in n:
         if ncoins[value-coin]+1 < curr_n:
            curr_n = ncoins[value-coin]
            curr_coin = coin
      ncoins[value] <-- curr_n
      coin_type[value] <-- curr_coin

   solution[ncoins[C]] <-- 0
   value <-- 0
   while value < C:
      add coins[value] to solution[]
      value <-- value + coins[value]

   return solution[]
```

Time complexity: O(nC)

The first two-level loop runs in O(nC) and the second while loop runs in O(C). Therefore, the time complexity is dominated by O(nC).

3.

The maximum total value is achieved by adding as much as high value/weight ratio items as possible. To avoid sorting the ratios in O(nlogn) time, we can use the median selection algorithm to find the median of the ratios. The steps of fractional knapsack problem are:

(1) Find the median of the value/weight ratio in O(n)

(2) Check if I can fit all the items with greater ratios than the median into the Knapsack in linear time:

- If yes, recursively fill in the knapsack for the rest n/2 items
- If no, recursively fill in the knapsack for the upper n/2 items and discard the lower n/2 items

int median_ratio_helper(ratios[n], int k)

```
l[] <-- 0
g[] <-- 0
p[] <-- 0
pivot <-- ratios[k]    //arbitrarily choose a pivot
for i <-- 1 to n
   if ratios[i] < pivot
      add to l[]
   else if ratios[i] = pivot
      add to p[]
   else
      add to g[]

if l.length = g.length
   return pivot
if l.length >= k
   median_ratio(l, k)
else
   median_ratio(g, k-l.length-p.length)
```

int median_ratio(items[n], k)

```
for i <-- 1 to n:
   ratios[i] <-- items[i].value/items[i].weight
return median_ratio_helper(ratios[n], k)
```

int f_knapsack(items[], n, W)

```
if n = 0 or W = 0
   return 0
if n = 1 and items[n].weight >= W
   return items[n].weight/W*items[n].value

median <-- median_ratio(ratios, n/2)
w <-- 0
greater[] <-- 0
less[] <-- 0
for i <-- 1 to n:
   if ratios[i] > median
      add items[i] to greater[]
      w <-- w+items[i].weight
   else
      add items[i] to less[]
if w < W
   k <-- k + f_knapsack(less[], less.length, W-w)
else
   k <-- k + f_knapsack(greater[], greater.length, W)
return k
```

Time complexity: O(n)

median_ratio_helper() finds the median of the ratios in linear time. Since it constantly breaks the array into approximately halves and do comparisons on each of the subarray's elements, T(n) = n+n/2+n/4+n/8+....=2n=O(n). median_ratio() calculates ratios and calls median_ratio_helper(), so its time complexity is also O(n). f_knapsack() finds the median on the current array and calls itself on the either the upper subarray or the lower subarray; therefore, T(n) = T(n/2) + O(n) = O(n)

4.

In order to find the number of significant inversions in O(nlogn), we can use a divide and conquer method in mergesort. We sort one half of the array each time and merge the left half with the right half. During the merging process, we can examine whether the left array contains any elements larger than twice of any element in the right array. If we have found one, since the left and the right array are already sorted, we know the elements in the rest of the left array are all greater than the current element in the right array. Thus, we can implement the counter. By recursively sorting and merging subarrays, we find all the significant inversion pairs in O(nlogn), which is the time complexity of mergesort.

int merge(A, l, m, r)

```
i <-- 0
j <-- m+1
k <-- 0
arr[l-r] <-- 0
while i <= m and j < r
   if A[i] <= A[j]
      arr[k++] <-- A[i++]
   else
      arr[k++] <-- A[j++]
      if A[i] > 2*A[j]
         count <-- count + (m - i)

while i <= m
   arr[k++] <-- A[i++]
while j < r
   arr[k++] <-- A[j++]
A <-- arr

return count
```

int merge_sort(A, l, r)

```
if l < r
   return 0
else
   count <-- 0
   mid <-- (l+r)/2
   count <-- count
```

```
            + merge_sort(A, l, m)
            + merge_sort(A, m+1, r)
            + merge(l, m+1, r)
    return count
```

Time complexity: O(nlogn)

Merging takes a time complexity of O(n) because it needs to traverse every element in the array. In the recursive function merge_sort(), we divide the current array into two halves and call itself on each half. Afterwards, we merge the two arrays. Therefore, $T(n) = 2T(n/2)+O(n) = O(nlogn)$.