CS180 HW6

Siddharth Joshi

TOTAL POINTS

100 / 100

QUESTION 1

1 Problem 1 **25 / 25**

✓ **- 0 pts** Correct

QUESTION 2

Problem 2 25 pts

2.1 2.a **5 / 5**

✓ **- 0 pts** Correct

2.2 2.b **10 / 10**

✓ **- 0 pts** Correct

2.3 2.c **10 / 10**

✓ **- 0 pts** Correct

QUESTION 3

3 Problem 3 **25 / 25**

✓ **- 0 pts** Correct

QUESTION 4

4 Problem 4 **25 / 25**

✓ **- 0 pts** Correct

HW6

1. Divide and Conquer approach:
→ Determine max and 2nd max of 1st half
→ " 2nd half
→ Determine max and 2nd max of entire array from this

```
maxima(Array A ~~size~~ int size):
  if size == 2:
      ~~return~~ if (A[0] > A[1]):
                    return [A[0], A[1]]
                else
                    return [A[1], A[0]]
  A1 = list of first size/2 elements of A
  A2 = list of remaining elements of A
  max1 = maxima(A1, size/2)
  max2 = maxima(A2, n - size/2)
  if (max1[0] ≤ max2[1])
      return ~~[max2[0], max2[1]]~~ max2
  else if (max2[0] ≤ max1[1])
      return max1
  else if (max2[0] ≥ max2[0])
      return [max2[0], max1[0]]
  else
      return [max1[0], max2[0]]
```

Time ~~complexity~~: # of comparisons = ~~n+1~~ ~~3~~ lg n
= n + O(lg n)

as base case is where comparisons happen directly between elements and that is just 1 comparison ⇒ n overall

resultantly and each of the $\lg n$ subproblems needs 3 comparisons Hence $n + 3\lg n$

Correctness: For a 2 element array, the base case is trivially correct i.e. the smaller element is 2nd max and bigger is 1st max

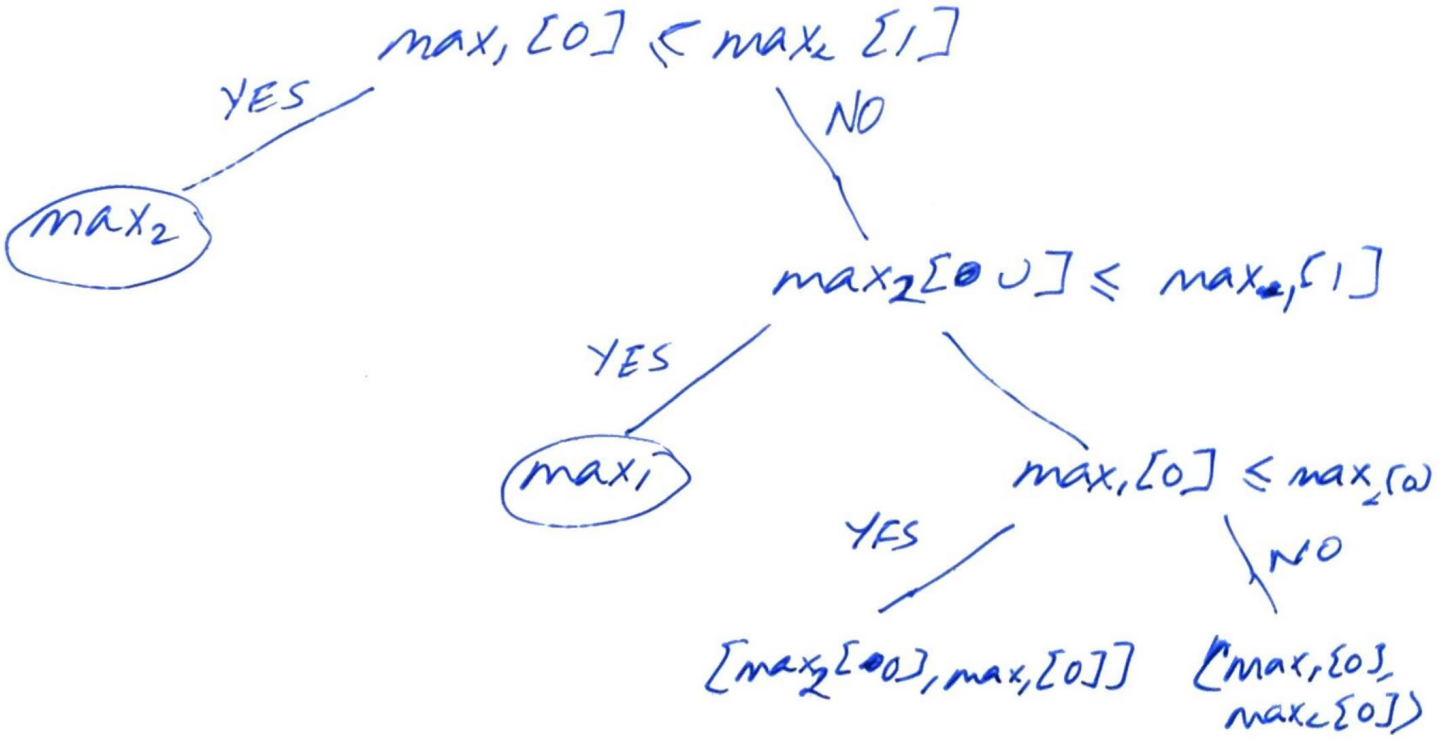
What remains is to prove that given the solution for 2 halves, they can combined correctly in $O(c)$ comparisons

assume max_1 is the array containing 1st max of 1st half, 2nd max of 1st half

similarly max_2 — " — for 2nd half

then $max = [max_1, max_2]$ or $[max_1[0], max_2[0]]$ or $[max_2[0], max_1[0]]$

this can be determined using the following decision tree, where max depth = max compares = 3



1 Problem 1 25 / 25

✓ - 0 pts Correct

2 a) Counter example:

assume

$A_3 = \{1, 100, 150\}$

$C = 500$

OPT = $\{2 \times 150, 2 \times 100\}$ · 4 coins

sol given by algorithm = $\{3 \times 150, 50 \times 1\}$ = 53 coins

since sol takes more coins than OPT, the naive greedy algorithm doesn't get the min # of coins needed

b) Assume that the greedy solution takes i instances of a certain (largest) coin in its final solution set and the optimal solution takes j of them

Let the value of this coin be V

The greedy approach takes as many as it can i.e.

$$0 \leq C - iV < V$$

In specific i $\lfloor C/V \rfloor$

the subproblem left after choosing this is the problem same problem with $C = C - iV$

∴ For the optimal solution to be 'better' it chooses j s.t. j + min coins w value $C = C - jV$ is less than i + min coins w value $C = C - iV$

· either ① $i > j$ and minCoins $(C - iV) \geq$ minCoins $(C - jV)$

or ② $i = j$ and minCoins $(C - iV) >$ minCoins $(C - jV)$

(Note $i < j$ is impossible as greedy takes as many coins of value V as possible)

For ① it is sufficient and ② it is sufficient to see that minCoins $(C - iV) \leq$ minCoins $(C - jV)$ as the next to the optimal solution chooses at most $X-1$ instances of next largest coin x but since as $X^2 a = C$ and taking more

2.1 2.a 5 / 5

✓ - 0 pts Correct

2 a) Counter example:

assume

$A_3 = \{1, 100, 150\}$

$C = 500$

OPT = $\{2 \times 150, 2 \times 100\}$ · 4 coins

sol given by algorithm = $\{3 \times 150, 50 \times 1\}$ = 53 coins

since sol takes more coins than OPT, the naive greedy algorithm doesn't get the min # of coins needed

b) Assume that the greedy solution takes i instances of a certain (largest) coin in its final solution set and the optimal solution takes j of them

Let the value of this coin be V

The greedy approach takes as many as it can i.e.

$$0 \leq C - iV < V$$

In specific i $\lfloor C/V \rfloor$

the subproblem left after choosing this is the problem same problem with $C = C - iV$

∴ For the optimal solution to be 'better' it chooses j s.t. j + min coins w value $C = C - jV$ is less than i + min coins w value $C = C - iV$

· either ① $i > j$ and minCoins $(C - iV) \geq$ minCoins $(C - jV)$

or ② $i = j$ and minCoins $(C - iV) >$ minCoins $(C - jV)$

(Note $i < j$ is impossible as greedy takes as many coins of value V as possible)

For ① it is sufficient and ② it is sufficient to see that minCoins $(C - iV) \leq$ minCoins $(C - jV)$ as the next to the optimal solution chooses at most $X-1$ instances of next largest coin x but since as $X^2 a = C$ and taking more

than X of coin X is always sub optimal as you can choose a single coin of value X instead of X coins of X and since the greedy solution can also pick anywhere from 0 X-1 instances of each X → it is easy to see that it performs at least as well as the optimal at every stage ⇒ it is optimal

c) Subproblems: remaining C
DP[C], DP[C-1] DP[0]

② guess which coin to take next (take all possible)

③ $DP[C] = \min_{\forall a_i \leq C} (1 + DP[C - a_i])$ } this recurrence is correct as it tries all options and this recurrence proves the correctness of the algorithm

④ Topological sort:

∴ 0 ... C

⑤ Final answer is DP[C]

```
min Coins(Array A, value int C):
    Let DP be an array of size C+1 and set DP[0] = 0
→ For j in range(1, C):
    • min = +∞
    • For i in range(1, n):
        * if (a_i ≤ j)
            → if(DP[j - a_i] + 1 < min)
                • min = DP[j - a_i] + 1
    • DP[k] = min
→ Return DP[C]
```

Time Complexity: inner for loop = O(n) outer for loop = O(C)
∴ Total time complexity = O(nC)

2.2 2.b 10 / 10

✓ - 0 pts Correct

than X of coin X is always sub optimal as you can choose a single coin of value X instead of X coins of X and since the greedy solution can also pick anywhere from 0 X-1 instances of each X → it is easy to see that it performs at least as well as the optimal at every stage ⇒ it is optimal

c) Subproblems: remaining C
DP[C], DP[C-1] DP[0]

② guess which coin to take next (take all possible)

③ $DP[C] = \min_{\forall a_i \leq C} (1 + DP[C - a_i])$ } this recurrence is correct as it tries all options and this recurrence proves the correctness of the algorithm

④ Topological sort:

∴ 0 ... C

⑤ Final answer is DP[C]

```
min Coins(Array A, value int C):
    Let DP be an array of size C+1 and set DP[0] = 0
→ For j in range(1, C):
    • min = +∞
    • For i in range(1, n):
        * if (a_i ≤ j)
            → if(DP[j - a_i] + 1 < min)
                • min = DP[j - a_i] + 1
    • DP[k] = min
→ Return DP[C]
```

Time Complexity: inner for loop = O(n) outer for loop = O(C)
∴ Total time complexity = O(nC)

Correctness in more detail:

The dp solution relies entirely on the idea that the optimal solution for value C is can be reduced to subproblems in this way:

$$DP[C] = \min_{\forall a_i \leq C} (1 + DP[C - a_i])$$

this is true as the solution contains of some coin # (some a_i) and - i.e. atleast 1 instance of some coin (except for DP[0] base case = 0) and therefore the optimal answer for C can be constructed by adding 1 to the optimal answer for DP[C - a_i] + C - a_i for some optimal a_i - which can be determined by trying all. Moreover since the DP relies on answers lower than it and since the table DP is constructed from 0 → C, these lower values always exist in the table.

2.3 2.c 10 / 10

✓ - 0 pts Correct

3.

Assume the input is an array of N ratios s.t. R

R_i $\forall i$ (if it isn't → can be made in O(n) trivially

W_i

fractionalKnapsack(Array R, int W, int N, Array Weights):

defined above / index / N weight of knapsack num of items

→ pick a number to pivot randomly from 1..N

→ Let X be = R[pivot]

→ Let smaller and larger be 2 empty lists

→ int sSize = lSize = 0

→ For i in {1.. N}:

- if (R[i] < R[pivot]
  - insert i in smaller
  - sSize + W[i]
- else
  - insert i in larger
  - lSize + W[i]

→ If (lSize = W)

return larger as items used, all used fully i.e. $\forall x_i = 1$

$\forall i \in$ larger and value achieved =

→ else if (lSize > W)

return fractionalKnapsack(larger, W, num of elements in larger, weights of elements in larger)

→ else

- items = recurse w items = smaller, knapsack weight = W - lSize

included =

→ add the item not included yet w largest ratio in some fraction that fits into W

→ return included + the last item added (for all items in included quantity = 1 and for last item = fraction that fit)

→ Return R since all items fit fully ∴ $x_i = 1 \ \forall i \in R$

Time Complexity: In the average case, the set larger ~ N/2 items ∴ subproblems solved are of ½ size
∴ $T(n) = T(n/2) + O(n)$ ~~$O(n^2)$~~
... solving the recurrence
~~$O(n) + O(n)$~~
$T(n) = O(n + \frac{n}{2} + \frac{n}{4} + \dots 1) = O(2n) = O(n)$

Avg case ↓

Note However in the worst case the pivot is chosen 'badly' and the subproblems are of size ~ N ∴ time complexity $= O(n^2)$

Correctness: Since fractions from 0 ... 1 are allowed and each item quantity $\in [0, 1]$, the greedy solution's idea to maximize per unit value is correct and can be done by picking larger elements of w larger ratios earlier as is done by this algorithm. Moreover the only choice to recurse makes sense as if items in larger don't all fit in the knapsack, the solution must contain some subset of larger items. Finally the last it is clear from this logic that at the end some Y item w largest ratio may not fit entirely but would fit 'fractionally' and hence a fraction of this item is added (This is obvious as assume that a previous item X was added fractionally to be taking ⇒ X didn't fit fully but then a larger fraction of X should be taken to take Y's space and set then Y is not the last item ⇒ contradiction)

3 Problem 3 25 / 25

✓ - 0 pts Correct

4.

sortAndCount(Array A):
→ if A.size() = 1
  • return [A and, 0]
→ [A1, invs1] = sortAndCount(first half of A)
→ [A2, invs2] = sortAndCount(remainder of A)
→ [sortedA, inversions] = mergeAndCount(A, A2)
→ inversions += invs1 + invs2
→ return [sortedA, inversions]

mergeAndCount.
→ inversions i j 0, sortedA = new empty list
→ while (i < A1.size())
  if (j = A2.size() or A1[i] > 2 A2[j])
    ▵ increment inversions += (A1.size - i) × j
    ▵ increment i
  • else
    ▵ increment j
→ while (i < A1.size() and j < A2.size())
  • if (A1[i] < A2[j])
    ▵ append A1[i] to sortedA and increment i
  • else
    ▵ append A2[j] to sortedA and increment j
→ while (i < A1.size())
  • append A1[i] to sortedA and increment i
→ while (j < A2.size())
  • append A2[j] to sorted A and increment j
→ return [sortedA, inversions]

wrapper (Array A): // actual func called, wraps sort and count to only return # of inversions

[temp, ans] = sortAndCount(A)

return ans

Correctness: The # of inversions (as inversions are defined by the @ i.e., $A[i] > 2A[j]$ and $i<j$) pairs (i,j) s.t.

can be thought of as # of inversions in the 1st half + # of inversions in the second half + # of inversions in the array formed by concatenating a sorted version of the 1st half to a sorted version of the 2nd half

as for every pair (i,j) it's @ i & j are either both in half 1 or both in half 2 or i is in half 1 and j is in half 2 as $i<j$.

But the # of inversions in the merge can be found trivially by trying to 'merge' sorted half 1 & sorted half 2 ~~wreath element doubled by~~ and counting the pairs of elements ~~(i,j)~~ where ~~i~~ is $(A_1[i], A_2[j])$ s.t. $A_1[i] > 2A_2[j]$ done by formula = (size(A_1) $- i$) $\times j$ at each step (i.e. each value of i and j where the inequality is true).

Time Complexity: Each call to mergeAndCount = $O(n)$ as only single while loops exist in it. sortAndCount → breaks A into $\log n$ subproblems ∴ total complexity = $O(n) \times \log n = O(n \log n)$

4 Problem 4 25 / 25

✓ - 0 pts Correct