CS180 HW5

Siddharth Joshi

TOTAL POINTS

95 / 100

QUESTION 1

1 Problem 1 25 / 25

✓ - 0 pts Correct

- 7 pts dp algorithm not correct
- 5 pts time complexity not correct; no run time analysis
- 20 pts wrong answer but showed efforts
- 25 pts wrong answer, no answer
- 10 pts No detailed algorithm, no dp equations
- 5 pts Does not show algorithm
- 20 pts Does not answer the question
- 25 pts No answer found

QUESTION 2

Problem 2 25 pts

2.1 2.a 15 / 15

✓ - 0 pts Correct

- 5 pts did not update at each position
- 10 pts wrong answer but showed efforts
- 15 pts no answer

2.2 2.b 10 / 10

✓ - 0 pts Correct

- 7 pts wrong answer but showed efforts
- 10 pts no answer

QUESTION 3

3 Problem 3 20 / 25

- 0 pts Correct
- 10 pts Need more explanation

✓ - 5 pts Fail to make MST or visited status accessible among all processors

- 5 pts Fail to show time complexity analysis
- 25 pts No answer found

QUESTION 4

4 Problem 4 25 / 25

✓ - 0 pts Correct

HW5

1

vector of size n

minSpacePrint (int M, vector<string> words):
- → let spaces be a 2D array of size [n][n] initialized to all 0s
- → let cost be a 2D array of size n×n or [n][n] initialized to all 0s
- → for i from 0 to words.size()-1:
  - for j from i+1 to words.size()-1:
    - ★ spaces[i][j] = M - j + i $\sum_{k=i}^{j}$ words[k].size()
    - ★ if spaces[i][j] < 0 then
      - → cost[i][j] = INT_MAX // as this is impossible
    - ★ elif j = words.size()-1 and spaces[i][j] ≥ 0 then
      - cost[i][j] = 0
    - ★ else
      - cost[i][j] = (spaces[i][j])³
- → let minCost be a vector of size n with first value 0 and rest = INT_MAX
- → let print be an empty vector
- → for i from 0 to words.size()-1:
  - for j from 0 to i-1:
    - ★ if minCost[j] + cost[j+1][i] < minCost[i] then
      - minCost[i] = minCost[j] + cost[j+1][i]
      - print[i] = j+1
- → recursivePrint(print, words.size())
- → return minCost[words.size()-1]

recursivePrint(result, counter):
- → if result[counter] = 1 then return

contd. on next pg

→ recursivePrint(result, result[counter] - 1)
→ printToScreen(words[result[counter]])

Correctness: The algorithm's fundamental logic lies in the recursive relation:

$$minCost[n] = \min_{j=0}^{n-1} (minCost[j] + cost[j+1][n-1])$$

where minCost[n] is the min sum of abs of extra spaces for words 0 n-1 and cost[m][n] is the extra space at the end of the line if words from m to n-1 are on the same line

Time Complexity: Main computational bottleneck are the 2 for loops → $O(n^2)$ is the time complexity
Space complexity - $O(n^2)$ as well

1 Problem 1 **25 / 25**

✓ **- 0 pts Correct**

- 7 pts dp algorithm not correct

- 5 pts time complexity not correct; no run time analysis

- 20 pts wrong answer but showed efforts

- 25 pts wrong answer, no answer

- 10 pts No detailed algorithm, no dp equations

2. a) Divide and Conquer Approach

longestSubsequence (Array A[1 ... n]):
→ let S[1 ... n] be an array of size n initialized to all 0s that is used to indicate the longest subsequence that
→ ~~For every num in A~~ contains the element from A at the corresponding index
→ For every num in A:
  - S[index of num] = longestSubsequenceEnd(index of num) + longestSubsequenceStart(index of num) − 1

a.k.a. LSS

→ ~~For~~ int max = 0
→ For i: 1 through n:
    if S[i] > max
      max = S[i]
→ Return max

a.k.a. LSE

longestSubsequenceEnd (Array R, index of num):
  max over all LSE for sub arrays of R starting at 1 index of num and end at index of num

LSS i.e. - can be defined similarly

Correctness: Obvious from the brute force way that the longest Subsequence Start (LSS) and longest Subsequence End (LSE) functions run. Simply try nearly all combinations.

Time Complexity = $n^2 + 2(\_)^2 \cdot \ldots \cdot n = O(n^2)$
(given in Q)

2.1 2.a **15 / 15**

✓ **- 0 pts Correct**

- 5 pts did not update at each position

- 10 pts wrong answer but showed efforts

- 15 pts no answer

b)

longestIncreasingSubsequence (Array A[1 ... n]):

→ integer array dp is an array of size n

→ int length = 0

→ for every num in A:

- int i = binarySearch (dp, num)
- if (i < 0):
  i = -(i+1)
- dp[i] = num
- if (i == length):
  length++

→ return length.

binarySearch (dp, num): // Works like Arrays.binarySearch from java lib.

- if found return position
  else return -(insertion position - 1)

Correctness: At all points our array dp has a valid subsequence in it by the nature of its construction and hence returning the max length it takes → gives solution

Time Complexity: It's easy to see that the runtime = O(n log n) where n is size of Array A as for every element we do a binary search ∴ Runtime (n × log(n)).

T time for binary search

2.2 2.b **10 / 10**

✓ **- 0 pts** **Correct**

- 7 pts wrong answer but showed efforts

- 10 pts no answer

3. Claim: ~~Assumption~~ : polynomial of n processors can find min in $\log n$ rounds

~~as~~ proof : Let each processor hold a value
compare the value in each processor with an adjacent processor such that $\frac{n}{2}$ comparisons are made

keep repeating this.
It is easy to see that finding min using polynomial of n processors can be done in $\log n$ time in a manner similar to binary search

∴ MST (Graph G)
→ ~~Find each remaining edge~~ MST = empty
→ while (MST not found):
Find remaining min edge that doesn't result in a cycle and add it to MST
→ Return MST

Correctness : This is essentially Kruskall's using query and ~~find~~ union and thus correctness follows

Time complexity : $O(n \log n)$
Find = $\log n$ and Find is done n times
∴ $O(n \log n)$

3 Problem 3 20 / 25

- **0 pts** Correct
- **10 pts** Need more explanation
- ✓ **- 5 pts** **Fail to make MST or visited status accessible among all processors**
- **5 pts** Fail to show time complexity analysis
- **25 pts** No answer found

Assume n objects

4.a)

greedyKnapsack (vector of 3-tuples with 1st element in tuple indicating its weight, the second its value and the 3rd the quantity available)

→ Let ratio be a vector of size n where every element corresponds to the value/weight ratio of the element at the corresponding index of the input vector
→ Sort input vector s.t. element w highest ratio is at front
→ While (knapsack has space):
- Pick as many of the item at the front of the input vector while the knapsack has space and there are instances of the item remaining
- if item runs out, delete this item from the input vector and continue to next iteration

→ Return the value of items in the knapsack

b) Assume knapsack of size 50 lb
Assume 3 items: item 1 → (weight = 10 lb, value = 60$, quantity = 1)
ratio = 6
item 2 → (weight = 20 lb, value = $100, quantity = 1)
ratio = 5
item 3 → (weight = 30 lb, value = $120, quantity = 1)
ratio = 4

The greedy algo picks 1 and 2 and then stops as no more can fit in the knapsack, but the optimal solution picks 2 and 3 as val(1+2) = $160 but val(2+3) = $220

c) and d) Proving d) directly as d) generalizes c)

Let $v_1 \dots v_n$ be the values of the items

Let $s_1 \dots s_n$ be the sizes

Let k be the index of the first item that greedy doesn't take

$\therefore \underbrace{v_1 + v_2 + \dots + v_{k-1}}_{s_1 + s_2 + \dots + s_{k-1} \leq 1} =$ greedy solution

Let OPT be the optimal value

$\therefore \sum_{i=1}^{k} v_i \geq OPT$

rather $\sum_{i=1}^{k-1} v_i + \alpha v_k \geq OPT$ where α is the fraction of item k that the knapsack can fit

$\therefore$ either $\sum_{i=1}^{k-1} v_i \geq \frac{OPT}{2}$ or $v_k \geq \frac{OPT}{2}$

4 Problem 4 25 / 25

✓ **- 0 pts** **Correct**

- 5 pts Does not show algorithm

- 20 pts Does not answer the question

- 25 pts No answer found