Jason Less
404-640-158
CS180

CS180 Homework 5

Question 1

To design an algorithm to minimize the sum of the cube of sum of numbers of extra space characters at the end of lines, I will make use of several data structures. The first will be a 2D array called spaces, which represents the number of extra spaces at the end of a line containing word i to word j, where $spaces[i,j] = M - j + i - \sum_{k=i}^{j} l_k$. Additionally, another 2D array called lineCost, which is the cost of including a line containing word i to word j in the sum that we are trying to minimize, where:

$$lineCost[i,j] = \begin{cases} \infty \; if \; spaces[i,j] < 0 \\ 0 \; if \; j = n \; and \; spaces[i,j] \geq 0 \\ spaces[i,j]^3 \; otherwise \end{cases}$$

The lineCost is equal to infinity represents that word i to word j doesn't fit on the given line, lineCost equal to 0 represents that the last line will have a cost of 0 (as word i to word j fit on the given lines). Setting the lineCost equal to infinity also prevents adding such a case to the sum we are trying to minimize. As the goal is to try to minimize the sum of the lineCost array over all the lines in the paragraph, we can think of a subproblem of how to optimally arrange words 1 to j. In this optimal arrangement, it is the case where the last line contains word i to word j, which means that the preceding lines contains words 1 to word i – 1. Another array costs will be used to represent the cost of an optimal arrangement of a given sequence of words, where costs[j] represents the cost of the optimal arrangement of word 1 to word j. Using this array, if the last line contains word i to j, then costs[j] = costs[i – 1] + lineCost[i, j]. It can be seen that the base cases in this case would be costs[0] = 0 and costs[1] = lineCost[1, 1].

Using this subproblem, we still need to find out which word should be the first word on the last line, and this can be done by trying all possibilities for word i recursively, and picking the one with the minimum cost. Therefore, costs can be represented as:

$$costs[j] = \begin{cases} 0 \; if \; j = 0 \; (base \; case) \\ \min \; (costs[i-1] + lineCost[i,j] for \; i = 1 \; to \; j \end{cases}$$

As each costs value depends on earlier values, costs[j] can be computed from left to right. Finally, to keep track of what words belong on which line, a final array will be used called places, which points to where each costs value finished at. If costs[j] is based on some value of costs[k – 1], then set places[j] = k. After computing the costs of all words up to costs[n], the places array can be traced back to see where to place each words and break the lines in the paragraph. The algorithm follows.

Jason Less
404-640-158
CS180

Question 1 (cont.)

Algorithm:

```
# Compute spaces[i, j] for 1 to n
for i = 1 to n
  spaces[i, i] = M - li
  for j = i + 1 to n
    spaces[i, j] = spaces[i, j - 1] - lj - 1
# Compute lineCost[i, j] for 1 to n
for i = 1 to n
  for j = i to n
    if spaces[i, j] < 0
      lineCost[i, j] = inf
    else if j == n and spaces[i, j] >= 0
      lineCost[i, j] = 0
    else lineCost[i, j] = (spaces[i, j])^3
# Compute costs[j] and places[j] for 1 to n
costs[0] = 0
for j = 1 to n
  costs[j] = inf
  for i = 1 to j
    if costs[i - 1] + lineCost[i, j] < costs[j]
      costs[j] = costs[i - 1] + lineCost[i, j]
      places[j] = i
return costs and places
```

Jason Less
404-640-158
CS180

Question 2

(a)

The longest ascending subsequence (or Longest Increasing Subsequence LIS) can be found by using dynamic programming. The idea is to find the LIS when looking at just the first index in the array, then find the LIS when looking at the second index in the array, and so on. As an example, take the array [4, 5, 6]. Initialize a dp array to have the values [0, 0, 0] and a variable LIS to keep track of the LIS initialized to 0:

- Looking at index 0 (i.e. val 4)
  - A single element has LIS 1 (base case) -> dp = [1, 0, 0, 0] and LIS = 1
- Looking at index 1 (i.e. val 5)
  - Set dp[1] = 1
  - Now using another index j = 0, take the new search of [4, 5], and try to find an increasing subsequence. if (4 < 5) then dp[1] = max(dp[1], 1 + dp[0])
  - Set dp[1] = 2, LIS = 2
- Looking at index 2 (i.e. val 6)
  - Set dp[2] = 1
  - j = 0, [4, 6] -> dp[2] = max(dp[2], 1 + dp[0])
  - Set dp[2] = 2
  - j = 1, [5, 6] -> dp[2] = max(dp[2], 1 + dp[1])
  - Set dp[2] = 3, LIS = 3
- Thus, the LIS = 3

Algorithm:

```
LIS = 0
dp = [0..n]
for i = 0 to n
  dp[i] = 1
  for j = 0 to j = i - 1
    if dp[j] < nums[i]
      dp[i] = max(dp[i], 1 + dp[j])
      LIS = max(LIS, dp[i])
return LIS
```

The algorithm is O(N^2) as the outer for loop does one pass through the array, but the inner loop at worst spans the whole array (i.e. the last iter). The space complexity is O(N).

Jason Less
404-640-158
CS180

Question 2 (cont.)

(b)

The LIS can be computed in O(N log N) time using dynamic programming and binary search. The dp array keeps track of the LIS up to the current point in the execution of the algorithm, and binary search is used to find the position in the dp array to insert the next largest value for the current increasing subsequence. If the current value being looked at in the input array can replace the smallest value in our current increasing subsequence, then replace it. As an example, look at the array [10, 9, 2, 5, 3, 7]. For the sake of brevity, in the example below I will say just larger, but this means the first element that doesn't match less than the current value we are looking for (i.e. >=).

- Looking at index 0, is there a value in our dp array (currently empty), that is just larger than 10 (i.e. an increasing subsequence)?
    - There isn't, so add 10 to the dp array -> dp = [10]
- Looking at index 1, is there a value just larger than 9 in the dp array?
    - Yes, it is 10, so replace 10 with 9 -> dp = [9]
- Looking at index 2, is there a value just larger than 2?
    - Yes, it is 9, replace 9 with 2 -> dp = [2]
- Index 3, is there a value just larger than 5?
    - No, add 5 to dp -> dp = [2, 5]
- Index 4, is there a value just larger than 3?
    - Yes, it is 5, replace 5 with 3 -> dp = [2, 3]
- Index 5, is there a value just larger than 7?
    - No, so add 7 to dp -> dp = [2, 3, 7]

Algorithm:

```
for i to n
  iterator = binary_search nums[i] in dp
  if no such val exists >= nums[i]
    add nums[i] to end of dp
  else *iterator = nums[i]
return size of dp
```

The time complexity of this algorithm is O(N log N) as it makes one pass through the input array and in each pass there is a log N binary search portion. The space complexity is O(N) if we count our dp array.

Jason Less
404-640-158
CS180

Question 4

The 0/1 Knapsack problem cannot be solved by taking a greedy approach, and requires a DP solution. To show why greedy doesn't work look at a counterexample. Consider an array containing the values of items vals = [100, 10, 120] and the associated weights array wts = [2, 2, 3], and our knapsack can carry at most 4. The greedy approach would add values 100 and 10 to the knapsack with our knapsack having value 110 with weight 4. However, the optimal value is 120 by just taking the last item with value 120 and weight 3. Thus, greedy failed.