Authors: Aidan Dalgarno-Platt and Liam Coleman

Problem 1)

The program takes in the jobs as input and at the same time checks if any of the jobs are pro bono. If there aren’t any pro bono jobs, it outputs 0 and terminates. If there is at least one pro bono job, the jobs are sorted in ascending order by end time and a dynamic programming array of size n is created. The algorithm then loops through the sorted jobs from index 1 to index n and calculates I[i] while keeping track of the most recent compatible index. If the last compatible job’s end time is less than the current job’s start time, then that job is the new most recent compatible job.

Heart of the Solution:

What

* I[i] is the maximum number of intervals that can be scheduled in the range 0 to i

How

* I[i] = max(1 + I[last\_compatible\_index], I[i - 1])

Where

* I[n] contains the answer.

Taking in the input takes O(n) time. Sorting the jobs by end time takes O(nlogn) time. Setting up the initial dynamic programming array takes O(n) time each. In the main algorithm, the loop from 1 to n takes O(n) time. Therefore, the total runtime is O(n + n log n + n), which is just O(n log n).

Problem 2)

a)

In this problem, we are given a set of numbers that are guaranteed to be positive integers. For a set of positive integers, adding before multiplying always yields a greater or equal value when compared to multiplying and then adding, since positive integers are guaranteed to be greater than or equal to 1. Take this equation, for example:

2 \* 2 + 3

Multiplying first yields 4 + 3 = 7. Adding first yields 2 \* 5 = 10. Adding the 3 first means the 3 is effectively counted twice, since it is multiplied by the 2 afterwards. This can be proved with a basic application of the distributive property.

2 \* (2 + 3) = (4 + 6) = 10

To generalize, take an integer x >= 1 and a sum of integers y1 + y2 + … yn that are also >= 1. By the distributive property, adding the n integers together first and then multiplying with x yields a greater result than adding any of the n integers last.

x \* (y1 + y2 + … yn) = x\*y1 + x\*y2 + … x\*yn

If x >= 1, then x multiplied by any integer will be greater than or equal to the original integer, so multiplying after adding always yields a greater result.

Taking in the numbers as input takes O(n), summing all the numbers first takes O(k), where k is the number of numbers joined by plus signs, and multiplying the sums together takes O(s) where s is the number of sums. The values of k and s are guaranteed to be less than or equal to n, so the total runtime is O(n + n + n), which is just O(n).

b)

This problem is the same setup as part a, only there are ‘+’ and ‘-’ signs instead of ‘+’ and ‘\*’ signs. Since two ‘-’ signs can cancel out and become a positive, working out this problem requires recursion. However, performing all the addition operations (aka putting them all in parenthesis) before performing any subtraction, just like in the last problem, is guaranteed to be part of the solution, as maximizing the solution requires the largest magnitude numbers. This section of the algorithm takes O(n) time.

Heart of the Solution:

What

* dp[i][j] = the maximum value for the subequation from integer i to integer j

How

* Every dp[i][i] = integers[i] because the maximum value of the equation from integer i to itself is just that integer
* If k == i, then dp[i][j] = max(dp[i][j], dp[i][k] - dp[k + 1][j])
* Else, dp[i][j] = max(dp[i][j], dp[i][k] - dp[k + 1][j], dp[i][k] + dp[k + 1][j])

Where

* The answer is in dp[1][n]

The algorithm loops through three nested loops in a very similar fashion to the matrix multiplication minimum operations.

Taking in the input takes O(n) time, setting up the initial dynamic programming array takes O(n2) time, setting all the diagonals takes O(n) time, and performing the meat of the algorithm takes O(n3) time. All in all, this is O(n + n2 + n + n3), which is O(n3).

Problem 3)

Heart of the Solution:

What

* longest[i] = the size of the longest subsequence from 0 to i
* count[ii] = the number of occurrences of the subsequence from 0 to i

How

* If X[j] < X[i] and longest[j] + 1 > longest[i]
  + If the number at j is less than the number at i and the subsequence is longer than the longest subsequence so far, set the new longest subsequence equal to the current subsequence
  + This is the same algorithm for counting the longest subsequence normally
* If X[j] < X[i] and longest[j] + 1 > longest[i], but if if X[j] < X[i] and longest[j] + 1 == longest[i], then count[i] += count[j]
  + When a new longest subsequence is found, set the count equal to the count of sequences of that length
  + If the current subsequence length is equal to the longest so far, increase the total count by the count found for the current subsequence
  + Basically, find the count for the current subsequence, and if it turns out that subsequence is the same length as the longest, add this current count to the total count for subsequences of the longest length

Where

* The answer is the sum of all count[i] where longest[i] is equal to the length of the max subsequence, so the sum of the count of all longest subsequences found

Taking in the input takes O(n) time, setting up the two initial dynamic programming arrays takes O(n) time each, and then looping through the main algorithm where the outer loop is from 0 to n and the inner loop is from 0 to i is O(n2). Therefore, the total runtime is O(n + n + n + n2), which is just O(n2).

Problem 4)

Heart of the Solution:

What

* dp[i][j] is the cost of converting the first i letters of string X to the first j letters of string Y

How

* dp[i][j] = min(dp[i - 1][j] + 3, dp[i][j - 1] + 4, dp[i - 2][j - 1] + 5)
* Deleting the current character in string X adds 3 to the cost of changing i - 1 characters into j, because it’s like removing 1 character from string X that we don’t have to worry about anymore
* Inserting a character into string X adds 4 to the cost of changing i characters into j - 1 characters, because it’s like skipping that j character for the cost of 4
* Replacing the current and last character in string X with another character is like performing 2 deletes and 1 insert, so the i is moved back 2 and the j is moved back 1
* Also, dp[i][j] = dp[i - 1][j - 1] if the character in string X is already the same as the one in string Y at the same location
* Finally, dp[i][j] = min(dp[i - 1][j] + 3, dp[i][j - 1] + 4) if i < 2 because that means there aren’t at least 2 characters to replace in string X, so it’s an invalid command. Without this if, the return cost was far too low.

Where

* The answer is in dp[n + 1][m + 1]
* There’s an extra row and column because the first row is for when string X is empty and the first column is for when string Y is empty

Taking in the inputs is O(m + n) and creating the initial dp array is O(mn). The outer loop of the actual algorithm loops n + 1 times and the inner loop loops m + 1 times, meaning it runs in O((m + 1)(n + 1)), which is just O(mn). The other operations inside the loops are constant time, so the whole algorithm is O(mn).