Problem session 11

Dibran Dokter 1047390

November 28, 2019

11

11.1

To solve this problem we can split it up into smaller problems and solve it recursively. Namely, where should we place the first shop for a maximum profit, and the second etc. We start out with 0 shops, and thus a profit of 0. After this we want to find the location that will give us the greatest profit. We do this by finding the maximal profit for all the different locations past k.

This gives the following recurrence equations:

$$Profit(0) = 0$$

$$Profit(n) = max(Profit(n-1) + P_i, Profit(n-1))$$

Where i $\geq$ Pos(n-1) + k.
Namely the next possible position we can place a shop.
Using the function Pos(n) = the number of km's where we placed shop m_n.

These equations hold because we find the highest profit for every shop using the recurrence equation. After placing the last shop we have calculated the maximal profit.

This algorithm gives a linear time complexity because for placing every shop we need to look at most at all the other possible options which does not depend on any placed shops. Thus we walk through the list of shops only once.

11.2

a)
We do this by creating an array M that has the number of paths for every value. So the amount of paths for k = 0..k.
We fill this array by starting in the top left and adding a value either from the bottom or the right. Then we add the number possible ways to get that value to the array on the correct index.
After we have walked through the entire array we have a list with all the possible k values we can get and the number of paths to get them. Then we return the number in the k'th index of the array.

This idea is contained in the following recurrence equation:

$$Paths(0, 0, k) = 0$$

$$Paths(i, j, 0) = 0$$

$$Paths(i+1, j, k) = M[k] \; in \; M[v[i, j] + v[i+1, j] = M[i, j] + 1$$

$$Paths(i, j+1, k) = M[k] \; in \; M[v[i,j] + v[i,j+1] = M[i,j] + 1$$

$$Paths(i, j, k) = M[i+1, j, k] + M[i, j+1, k]$$

Thus, when we calculate the number of paths for a certain matrix we need to fill the M array for all the possible values that can be found in the matrix with the number of paths for that value.

At the end we can find the solution from the matrix by reading the value in $M[k]$.
b)

```
1   FindCPaths(Mat[i,j], k) // Mat is the matrix and k is C
2   {
3       M[k] = {0}; // Create the matrix with size k and fill it with zeroes.
4       i, j = 0;
5       for(i = 0; i++; i < Mat.rows) // Loop through the rows.
6       {
7           for(j = 0; j++; j < Mat.cols) // Loop through the columns.
8           {
9               M[Mat[i,j] + Mat[i+1,j] = M[i,j] + 1; // 3rd recurrence
                    equation.
10              M[Mat[i,j] + Mat[i,j+1] = M[i,j] + 1; // 4th recurrence
                    eqation.
11          }
12      return M[k];
13  }
```

This algorithm gives a time complexity of $\mathcal{O}(n^2)$ since we have two for loops that go through the rows and columns.
c)

When using a top-down implementation we can both use memoization to reuse the values we have already computed, thus improving the performance. And we don't need to calculate all the indices. Since if we subtract a value and it becomes negative we know that that path is not a C-Path. In this way we reduce the number of paths we need to check.

11.3

To do this we create a list of words and integers that represent the position of the letter after the word. Using this list we can find the following recursion equations to find all the words.

$$words(0) = (\text{""}, 0)$$

$$words(n) = ifTrue(d(s[words(n-1) \; .. \; k]))$$

Where ifTrue adds the value to words if the word is in the dictionary.

Where $k = words(n-1) + 1 \leq k < n$
Thus, we try to find a word of length 1 to add, then a word of length 2 to add, etc. When we find a word, we add it to words and set the value of words(n) to the next letter after the word we found.

If we are either unable to find a word in the rest of the string or there are letters left after we have found all the words then the string cannot be constructed from the dictionary, otherwise we have found the words

that make up the string.

This algorithm has the problem that it will always find the shortest word it can find in the dictionary, so if the word s="there" it will find "the" and not "there". However we assume that the dictionary only has the correct versions of the word in it.

Using this algorithm gives the time complexity $\mathcal{O}(n)$ because we search for a word for every character in the string, this means that in the worst case we have only single letter words, thus we need to search the dictionary n times, which leads to a complexity of $\mathcal{O}(n)$.