CSCI 2300
Introduction to Algorithms: HW6

Sean Hyde
RIN: 662096071

Due: October 24th, 2024

1 (6.3)

1.1 Subproblem

To begin solving this dynamic programming problem, we must define the subproblem that will allow us to solve for all cases efficiently. The subproblem can be defined as follows:

- j: The location we are currently considering as we iterate through the available list of potential restaurant sites.
- i: The last index where a possible location could exist such that the distance between m_j and m_i is at least k. Essentially: $m_j - m_i \geq k$.

We then define our subproblem based on two cases:

- The case where the profit at location j is included.
- The case where the profit at location j is not included.

Thus, we define MTP_j (Maximum Total Profit up to location j) as follows:

$$\text{MTP}_j = \max\{P_j + \text{MTP}_i, \text{MTP}_{j-1}\}$$

where P_j represents the profit at location j and MTP_i represents the maximum profit up to the last possible location i that satisfies the k-mile constraint.

This solution assumes that we have computed MTP for all locations before j. Our algorithm will solve for the MTP at each location iteratively, allowing us to build up the solution for all possible locations n along the highway.

1.2 Algorithm

For each location n on the QVH, we calculate the MTP at each location by applying our subproblem. This will allow us to iteratively compute the maximum total profit at each location, resulting in the maximum possible profit achievable by the end of the highway.

1.3 Runtime

The runtime for this algorithm is $O(n^2)$. this is because we must firs titerate through each node n in the QVH. This results in a $O(n)$ time as we are iterating linearly. We then must iterate to find the first location i that satisfies the k mile constraint resulting in $O(n * n)$ time, which is $O(n^2)$.

2 (6.4)

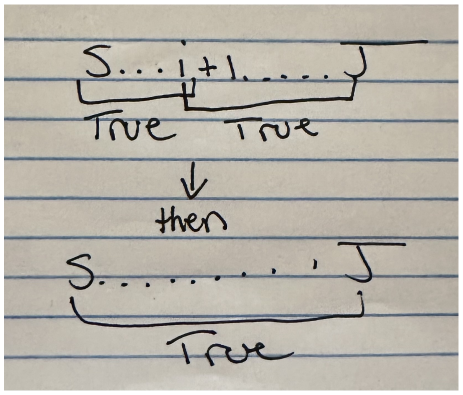
2.1 Subproblem

To begin with solving this problem we must start by defining the subproblem to be used in the algorithm.

- j: The character we are looking at.
- i: A character determined where $i < j$.

I'll define the algorithm as such: $isWord[j]$. We must first check for a value $i|i < j$. Once that is found, we run $isWord[i]$. If this is true, we know that there is a valid word from $S[1...i]$. We can then determine if there is a valid word from $dict(S[i+1...j])$. We do this since we know that $S[i]$ is valid, so we want to look at the immediate next character and test if there is a valid word from $S[i+1...j]$. If this returns true then we can say that $isWord[j]$ is true. If all of these are false then we say that $isWord[j]$ is false.

For example:



The final subproblem can be defined as follows:

$$\text{isWord}[j] = \begin{cases} \text{True} & \text{if } i < j \text{ such that isWord}[i] = \text{True and dict}(s[i+1\dots j]) = \text{True} \\ \text{False} & \text{otherwise.} \end{cases}$$

2.2 Algorithm

To implement the subproblem into the algorithm we simply run the subproblem on all characters within $S[.]$ and run the subproblem at each character j. Essentially a for loop through all characters in $S[.]$ with the subproblem.

2.3 Runtime

The runtime of this algorithm is $O(n^2)$ as we must iterate through each character in $S[.]$ and for each iteration iterate through and find a position $i < j$. Resulting in an $O(n^2)$ runtime.

2.4 (b)

Supposing we want to output the corresponding sequence of words we can slightly alter the algorithm to keep track of each index where a valid word is found. We can define a words array as $words[]$. From here, if $isWord[i]$ return true, we append the following to our words array: $words[1 \ldots i]$. We then do the same for dict(s[i+1 $\ldots$j]). We can then string concatenate the entries in the word array to output the corresponding sequence of words.

3 (3)

3.1 Subproblem

We must define a subproblem for our algorithm to use.

- j: the piece of yarn we are looking at
- i: A piece of yarn cut from j where $1 \leq i < j$

The subproblem can be defined in two different cases:

- We do not make any cuts to the piece of yarn j, and just take the P_j as the best value.
- We make a cut in j called i, and determine the max amount of money by cutting yarn j.

For MaxMoney[j]:

- Case 1 can best be described as just taking the P_j as the maximum amount of money for the piece of yarn j as we made no cuts to it.
- Case 2 can best be described as taking the P_i, the segment we have cut + $MaxMoney[j-i]$.

We then want to take the maximum of these two cases as this will be the maximum amount of profit that can be made by segmenting the piece of yarn j.

The subproblem can be defined as such:

$$\text{MaxMoney}[j] = \max\left(p_j,\ \max_{1\leq i<j}\left(p_i + \text{MaxMoney}[j-i]\right)\right)$$

3.2 Algorithm

We will iterate through each potential segmentation of the piece of yarn. For example if a piece of yarn with length n = 4, will have 4 possible price points for each integer-length segmentation. At each segmentation we will run our subproblem to find the maximum amount of profit that can be made by segmenting the yarn into integer-length pieces

3.3 Runtime

The algorithm will result in a $O(n^2)$ runtime since we must iterate through each integer-length segmentation and within each segmentation iterate through j possible cuts resulting in a runtime of $O(n^2)$.