CS 270 Homework 7

Neel Gupta

March 8, 2023

Problem 1. Certain languages like Chinese and Japanese are written without spaces between the words. Therefore, software to work with these written languages must address the word segmentation problem. How could we automate this process?

A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative "quality" of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters $x = x_1x_2...x_k$, will return a number $quality(x)$. This number can be either positive or negative; larger numbers correspond to more plausible worfds.

Given a long strong of letters $y = y_1y_2...y_n$, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The $totalquality$ of a segmentation block is determined by adding up the qualities of each of its blocks.

Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality.

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

 i. base cases and their values

 ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. For a given string y, find the maximum quality that can be obtained through partioning at various letters in the first i letters.

b. Let mem be an array for conducting dynamic programming on. Then $mem[i]$ will be the maximum capacity of the first i out of n characters in the string. At every level, we have two choices depending on qualities.

- Add the ith letter to the last word, then $mem[i] = quality(\text{last word} + y_i)$

- Do not include the *i*th letter in the last word, so start a new word, then $mem[i] = quality(\text{last word}) = mem[i-1]$

Therefore, the recurrence relation can be expressed as

$$mem[i] = \max(mem[j-1], quality(1 \dots y_i + y_j))$$

c.

```
procedure MOSTQUALITY($y$)
    Intitialize $mem[0] \leftarrow 0$
    for $i = 1, 2, ..., n$ do
        for $j = 1, ..., i$ do
            if $mem[i] < mem[j-1] + quality(1 \dots y_x + y_j)$ then
                $mem[i] \leftarrow mem[j-1] + quality(1 \dots y_x + y_j)$
            end if
        end for
    end for
    return $mem[n]$
end procedure
```

d. The base case is a string of size 0 would have 0 quality. The answer can be found in $mem[n][W]$.

e. The time complexity of the algorithm is $\Theta(n)$, where n is the number of types of letters in the string. For each of the n iterations, the first inner loop does $O(1)$ work. Thus, the total runtime is $\Theta(n)$. The total space complexity of this algorithm is $\Theta(n)$ for the DP table.

Problem 2. You are given an integer array $a_1, a_2, ..., a_n$, find the contiguous subarray (containing at least one number) which has the largest sum and only return its sum. The optimal subarray is not required to return or compute. Taking $a = [5, 4, -1, 7, 8]$ as an example: the subarray [5] is considered as a valid subarray with sum 5, though it only has one single element; the subarray $[5, 4, -1, 7, 8]$ achieves the largest sum 23; on the other hand, [5,4,7,8] is not a valid subarray as the numbers 4 and 7 are not contiguous.

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

   i. base cases and their values
   ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. The subproblems would be to find the maximum contiguous subarray from index i to index j.

b. We can initialize an n-by-n matrix dp of pairs which hold the current value and whether that pair came from a contiguous sequence. Therefore, we have 2 options at every iteration.

- Take jth item and add it to our optimal solution, so $dp[i][j] = a[j] + dp[i][j-1]$ if $dp[i][j-1]$ comes from a contiguous sequence itself.
- Lose the jth item, so jth item is not in our optimal solution even though it's in our possible range, then $dp[i][j] = dp[i][j-1]$, but we have to note that we are coming from a non contiguous sequence, so we can't start one there next time and must start anew.

Therefore, the recurrence relation can be written as

$$dp[i][j] = \max(dp[i][j-1], a[j] + dp[i][j-1])$$

consider contiguity with boolean variable checks along each entry in the matrix.

c.

```
procedure MAXSUBARRAY($a$, $n$)
    $dp$ := matrix of size $n \times n$ storing $(int, bool)$
    for $len = 1, 2, ..., n$ do
        for $i = 1, 2, ..., n - len$ do
            $j \leftarrow i + len - 1$
            if $dp[i][j-1].bool$ = true and $a[j] + dp[i][j-1] \geq dp[i][j-1]$
then
                $dp[i][j] = (a[j] + dp[i][j-1]$, true$)$
            else if $i = j$ then
                $dp[i][j] = (a[i]$, true$)$
            else
                $dp[i][j] = (dp[i][j-1]$, false$)$
            end if
        end for
    end for
    return $dp[0][n]$
end procedure
```

d. The base case is that single items by themselves have the value of that item and also are contiguous since they are one object. The final answer can be found in the value at $dp[0][n]$.

e. The total runtime is $\Theta(n^2)$ since there are up to n iterations at each iteration. The total space complexity is also $\Theta(n^2)$ for the table.

Problem 3. You are given an array of positive numbers $a_1, a_2, \ldots, a_n$. For a subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_t}$ of array a, if it is an increasing sequence of numbers, its happiness score is given by

$$\sum_{k=1}^{t} k \times a[i_k]$$

. Otherwise, the happiness score of this array is 0. Pleasae design an efficient algorithm to only return the highest happiness score over all the subsequences

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

   i. base cases and their values

   ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. The subproblems would be to find the maximum happiness from ith element to the jth element.

b. Consider an n-by-n matrix dp which stores tuples in the following form (value, maximum, k) for every entry. At every new number we have two options: take a new maximum which will increase the happiness value or skip taking an option, so the happiness will stay the same. We will never take decreasing values since they will give us no happiness.

If the new number is greater than the current max, $dp[i][j] += a[j] * k$, and we would update the current max, increment k, and note the value. Otherwise, $dp[i][j] = dp[i][j-1]$ where value, maximum, and k remain the same as before.

c.

```
procedure MAXHAPPINESS($a$, $n$)
    $dp :=$ matrix of size $n \times n$ storing $(int, int, int)$
    for $len = 1, 2, ..., n$ do
        for $i = 1, 2, ..., n - len$ do
            $j \leftarrow i + len - 1$
            if $dp[i][j-1].max < a[j]$ then
                $dp[i][j].max \leftarrow a[j]$
                $dp[i][j].k \leftarrow dp[i][j-1].k + 1$
                $dp[i][j].value += a[j] * dp[i][j].k$
            else if $i = j$ then
                $dp[i][j] = (a[i], a[i], 1)$
            else
                $dp[i][j] = dp[i][j-1]$
            end if
        end for
    end for
    return $dp[0][n].max$
end procedure
```

d. For the base case of only having 1 balloon, the diagonal of the matrix will get filled with one element subarrays when $i = j$. The answer can be found in $dp[0][n].max$ since this will be the value for an array starting at 0 and ending at n with the maximum possible happiness.

e. The overall runtime is $\Theta(n^2)$ since we can have up to n iterations for each subset and there are up to n subsets. The overall space complexity is $\Theta(n^2)$ for the DP table.

Problem 4. You've started a hobby of retail investing into stocks using a mobile app, RogerGood. You magically gained the power to see N days into the future and you can see the price of one particular stock. Given an array of prices of this particular stock, where $prices[i]$ is the price of the given stock on the ith day, find the maximum profit you can achieve through various buy/sell actions. RogerGood also has a fixed fee per transaction. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction (one pay once per pair of buy and sell). Assume you can own at most one unit of stock.

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

   i. base cases and their values

   ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. The subproblems to be solved are finding the maximum profit for the ith day.

b. At every day, we can either take the profit from the previous day or buy, own, and sell all within the day if we know that there was a profit from owning it today. If we buy a stock on a certain day, we lose all the profit we made on a previous day since the price of the stock is continuous and to buy means we previously sold at a more optimal price. Therefore, we can choose to include the ith day's gains if $prices[i] - prices[i-1]$ is greater than yesterday's profit. Therefore, we get the following recurrence relation.

If $i \neq 0, maxProfit = \max(prices[i] - prices[i-1], maxProfit)$. Otherwise, $maxProfit = 0$.

c.

```
procedure MAXPROFIT($prices$)
    $maxProfit \leftarrow 0$
    $n \leftarrow$ len($prices$)
    if $n < 2$ then
        return 0
    end if
    for $i = 1, ..., n$ do
        $maxProfit = \max(prices[i] - prices[i-1], maxProfit)$
    end for
    return $maxProfit$
end procedure
```

d. The base case is that if we don't have two days to trade the stock with, we cannot make any profit, so we return 0 with not enough prices. The final answer is found in $maxProfit$.

e. The time complexity of the solution is $\Theta(n)$ since we have n iterations which have constant work each. The space complexity of the solution would be $\Theta(1)$ since we don't memoize, but rather tabulate in a bottom-up fashion.