CSCI 570 - HW 5

Solutions

1. You are considering opening a series of electrical vehicle charging stations along Pacific Coast Highway (PCH). There are n possible locations along the highway, and the distance from the start to location k is $d_k \geq 0$, where $k = 1, 2, \ldots, n$. You may assume that $d_i < d_k$ for $i < k$. There are important constraints:

a) at each location k you can open only one charging station with the expected profit p_k.

b) you must open at least one charging station along the whole highway.

c) any two stations should be at least M miles apart.

Design a dynamic programming algorithm to maximize the profit and analyze its running time.

Solution:

Let OPT(k) be the maximum expected profit which you can obtain from locations 1, 2, ..., k.

OPT(k) = max{OPT(k − 1), p_k + OPT(f(k))} where f(k) finds the largest index j such that $d_j \leq d_k - M$, when such j doesn't exist, f(k) = 0.

Base cases: OPT(0) = 0, OPT(1) = p_1

Pseudocode:

```
OPT(0) = 0; OPT(1) = p_1;
for k = 2, · · · , n do
        f(k) finds the largest index j such that d_j ≤ d_k − M, when j doesn't exist, f(k) = 0.
        OPT(k) = max{OPT(k − 1), p_k + OPT(f(k))}
end
return Opt(n)
```

Algorithm solves n subproblems; each subproblem requires finding an index f(k) which can be done in time O(log n) by binary search. Hence, the running time is O(n log n).

2. Given a non-empty string *str* and a dictionary containing a list of unique words, design a dynamic programming algorithm to determine if *str* can be segmented into a sequence of dictionary words. If *str* ="algorithmdesign" and your dictionary contains "algorithm" and "design". Your algorithm should answer Yes as *str* can be segmented as "algorithmdesign". You may assume that a dictionary lookup can be done in O(1) time.

Solution:

Let $s_{i,k}$ denote the substring $s_i s_{i+1} \ldots s_k$. Let OPT(k) denote whether the substring $s_{1,k}$ can be segmented using the words in the dictionary, namely OPT(k) = 1 if the segmentation is possible and 0 otherwise. A segmentation of this substring $s_{1,k}$ is possible if only the last word (say $s_{i+1} \ldots s_k$) is in the dictionary and the remaining substring $s_{1,i}$ can be segmented. Therefore, we have equation:

OPT(k) = max OPT(i) where $0 < i < k$ and $s_{i+1,k}$ is a word in the dictionary

Pseudocode:

```
OPT(0) = 1;
for k = 2, · · · , n do
        for i such that 0 < i < k
                if s_{i+1,k} is a word in the dictionary
                OPT(k) = max OPT(i)
end
return OPT(n).
```

We can begin solving the above recurrence with the initial condition that OPT(0) = 1 and then go on to compute OPT(k) for k = 1, 2, ..., n. The answer corresponding to OPT(n) is the solution and can be computed in $O(n^2)$ time.

3. Given *n* balloons, indexed from 0 to *n* - 1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon *i* you will get *nums*[left] · *nums*[*i*] · *nums*[right] coins. Here left and right are adjacent indices of *i*. After the burst, the left and right then becomes adjacent. You may assume *nums*[-1] = *nums*[*n*] = 1 and they are not real therefore you cannot burst them. Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.

Solution:
The problem is similar to the chain matrix multiplication.

Let OPT (l, r) be the maximum number of coins you can obtain from balloons l, l+1, ..., r−1, r.

The key observation is that to obtain the optimal number of coins for balloon from l to r, we choose which balloon is the last one to burst. Assume that balloon k is the last one you burst, then you must first burst all balloons from l to k − 1 and all the balloons from k + 1 to r which are two sub problems. Therefore, we have the following recurrence relation:

$OPT(l, r) = \max \{OPT(l, k-1) + OPT(k+1, r) + nums[l-1] * nums[k] * nums[r+1]\}$ where $l < k < r$

Pseudocode:

```
OPT(l, 0) = 0;
for len = 2, · · · , n do
  for l = 0, · · · , n-len do
    r = left + len;
    for k = l + 1, · · · , r do
      OPT(l, r) = max{OPT(l, k−1) + OPT(k+1, r) + nums[l−1]∗nums[k]∗nums[r+1]}
return OPT(0,n)
```

Three for-loops result in $O(n^3)$ runtime.

4. A rope has length of *n* units, where *n* is integer. You are asked to cut the rope (at least once) into different smaller pieces of integer lengths so that the product of lengths of those new smaller ropes is maximized. Design a dynamic programming algorithm and analyze its running time. Explain how you would find the optimal set of cutting positions.

Solution:

Based on the assumption that you have to make at least one cut, an observation can be made: for a rope with length i, if the optimal cut includes a cut at length j, $1 \leq j \leq i - 1$, then at least one of the following 4 cases must happen for the left part (with length j) and right part (with length i − j) after the cutting:

- no cut on the left part, optimal cut on the right part;
- optimal cut on the left part, no cut on the right part;
- optimal cut on the left part, optimal cut on the right part;
- no cut on the left part, no cut on the right part;

Define OPT(i) as the maximum product of the lengths of smaller ropes after cutting the rope of length i. Based on above observation, we have the following recursive relation for length i > 2:

$OPT(i) = \max(j \times OPT(i-j), OPT(j) \times (i-j), OPT(j) \times OPT(i-j), j \times (i-j))$

where $1 \leq j \leq \lfloor i/2 \rfloor$

j only need to takes values up to $\lfloor i/2 \rfloor$ because of the symmetry of the rope; if j = 1, there is no possible cut on the part with length 1, we simply define OPT(1) = 1.

Base case: OPT (2) = 1, because it has to be cut at least once and this is the only one way to cut.

Pseudocode:

```
OPT(1) = 1; OPT(2) = 1;
for i = 3, · · · , n do
        OPT (i) = max(j ×OPT (i−j),OPT (j)×(i−j), OPT(j) × OPT(i-j), j × (i-j))
        where 1≤ j ≤ i/2
end
return OPT(n);
```

To find a set of cutting positions we need to record j and the Action (”cut” or ”no cut”) on the corresponding part that achieve the maximum, denoted as {j, Action(i, left), Action(i, right)}. Track back through the array OPT, by recursively checking the records starting at i = n.

Each iteration takes O(n) time to compute; there are O(n) iterations; Thus, the overall complexity is $O(n^2)$.