CSCI 570 Homework 3

Due Sunday Mar. 08 (by 23:30)

For all divide-and-conquer algorithms follow these steps:

1. Describe the steps of your algorithm in plain English.
2. Write a recurrence equation for the runtime complexity.
3. Solve the equation by the master theorem.

For all dynamic programming algorithms follow these steps:

1. Define (in plain English) subproblems to be solved.
2. Write the recurrence relation for subproblems.
3. Write pseudo-code to compute the optimal value
4. Compute its runtimecomplexity in terms of the input size.

1. Suppose we define a new kind of directed graph in which positive weights are assigned to the vertices but not to the edges. If the length of a path is defined by the total weight of all nodes on the path, describe an algorithm that finds the shortest path between two given points A and B within this graph.

   Solution:

   We have to reduce a vertex weighted graph G into an edge weighted graph G1. Create a new graph G1 as follows. Split each vertex v in G into two vertices vin and vout, with an edge weight between them equals to the vertex weight in G1. And the original edges from G have weights as 0 in G1. Edges coming to v in G will come to vin in G1. Edges leaving v in G will leave vout in G1. Run Dijkstra's algorithm. In the shortest path tree T1 for G1, an edge between vin and vout means that vertex v is in the shortest path tree T for G. Collapse all such edges (vin, vout) in T1 by merging vin and vout. This will create the shortest path tree T.

   Rubric: (10 points)

   - (10 points) The algorithm is stated clearly. And the algorithm is correct.

2. For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.
   - $T(n) = 3T(n/2) + n^2$
   - $T(n) = 4T(n/2) + n^2$
   - $T(n) = T(n/2) + 2^n$
   - $T(n) = 2^n T(n/2) + n^n$
   - $T(n) = 16T(n/4) + n$
   - $T(n) = 2T(n/2) + nlogn$
   - $T(n) = 2T(n/4) + n^{0.51}$
   - $T(n) = 0.5T(n/2) + 1/n$
   - $T(n) = 16T(n/4) + n!$
   - $T(n) = 10T(n/3) + n^2$

   Solution:

   - $T(n) = 3T(n/2) + n^2$ — case 3: $T(n) = \Theta(n^2)$

- $T(n) = 4T(n/2) + n^2$ — case 2: $T(n) = \Theta(n^2 logn)$
- $T(n) = T(n/2) + 2^n$ — case 3: $T(n) = \Theta(2^n)$
- $T(n) = 2^n T(n/2) + n^n$ — Does not apply, a is not constant
- $T(n) = 16T(n/4) + n$ — case 1: $T(n) = \Theta(n^2)$
- $T(n) = 2T(n/2) + nlogn$ — case 2: $T(n) = \Theta(nlog^2 n)$
- $T(n) = 2T(n/4) + n^{0.51}$ — case 3: $T(n) = \Theta(n^{0.51})$
- $T(n) = 0.5T(n/2) + 1/n$ — Does not apply ($a < 1$)
- $T(n) = 16T(n/4) + n!$. — case 3: $T(n) = \Theta(n!)$
- $T(n) = 10T(n/3) + n^2$. — case 1: $T(n) = \Theta(n^{log_3 10})$

Rubric: (10 points)

- (10 points) 1 point for each item. One for getting the case right and one for the runtime/justification of case

3. Suppose that we are given a sorted array of distinct integers $A[1, ..., n]$ and we want to decide whether there is an index i for which $A[i] = i$. Describe an efficient divide-and-conquer algorithm that solves this problem and explain the time complexity.

Solution:

If the array has just one integer, then we check whether $A[1] = 1$ with one comparison. Otherwise divide the list into two parts, the first half and the second half, as equally as possible. Consider the largest element $A[m]$ of the left half. We compare $A[m]$ with m. If $A[m] = m$, the answer is yes and we are done. If $A[m] > m$, then we can throw away the right half and continue recursively in the left half. Indeed, then for every integer $k \geq 0$ using the fact that the integers are distinct and sorted, $A[m + k] \geq A[m] + k > m + k$. If $A[m] < m$, then we can throw away the left half and continue recursively in the right half. Indeed, then for every integer $k \geq 0$ using the fact that the integers are distinct and sorted $A[m - k] \leq A[m] - k < m - k$. Thus for the number of comparisons we get the following recursion, $T(n) = T(n/2)+O(1), T(1) = O(1)$. According to Master Theorem, the time complexity is $\Theta(\log n)$.

Rubric: (15 points)

- (10 points) The algorithm is stated clearly.
  - 10 points if the time complexity of the algorithm is equal or less than $\Theta(\log n)$. And it is using divide-and-conquer algorithm.
  - 5 points if the time complexity of the algorithm is is larger than $\Theta(\log n)$. And it is using divide-and-conquer algorithm.
  - 0 points if the algorithm is not correct or the algorithm is not using divide-and-conquer.
- (3 points) The recurrence equation for run time complexity is correct.
- (2 points) Solving the equation by master theorem and the time complexity is $\Theta(\log n)$.

4. We know that binary search on a sorted array of size n takes $O(\log n)$ time. Design a similar divide-and-conquer algorithm for searching in a sorted singly linked list of size n. Describe the steps of your algorithm in plain English. Write a recurrence equation for the runtime complexity. Solve the equation by the master theorem.

Solution:

If the array has just one integer, then we easily find this value. Otherwise firstly we find the middle element of the list, this can be done in O(n). And we split the list into left list and right list. Compare this middle element with the search element. If they are equal and we are done. If the middle element is larger than the search element, we throw away the right half list and continue recursively on the left sublist. Otherwise we continue recursively on the left sublist. We have $T(n) = T(n/2) + O(n), T(n) = \Theta(n)$ according to Master Theorem.

Rubric: (15 points)

- (10 points) The algorithm is stated clearly.
  - 10 points if the algorithm is using divide-and-conquer algorithm and it is correct.
  - 0 points if the algorithm is not correct or the algorithm is not using divide-and-conquer.
- (3 points) The recurrence equation for run time complexity is correct.
- (2 points) Solving the equation by master theorem and the time complexity is $\Theta(n)$.

5. 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 $nums[i-1] \times nums[i] \times nums[i+1]$ 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.

   Example. If you have the $nums = [3, 1, 5, 8]$. The optimal solution would be 167, where you burst balloons in the order of 1, 5, 3 and 8. The array $nums$ after each step is:

$$[3, 1, 5, 8] \rightarrow [3, 5, 8] \rightarrow [3, 8] \rightarrow [8] \rightarrow []$$

   And the number of coins you get is $3 \times 1 \times 5 + 3 \times 5 \times 8 + 1 \times 3 \times 8 + 1 \times 8 \times 1 = 167$.

Solution:

Algorithm 1 Balloon Burst Max Coins

```
1: function MaxCoins(nums, n) ▷ num is the array representing balloons, n is the length of nums
2:     Create a table dp((n+2)*(n+2)) with all 0s in the table.
3:     nums = [1]+nums+[1]
4:     for length from 1 to n do
5:         for left from 1 to n-length+1 do
6:             right = left + length-1
7:             for k from left to right do
8:                 dp[left][right] = max{ dp[left][right], nums[left-1]*nums[k]*nums[right+1] + dp[left][k-1] + dp[k+1][right]}
       return dp[1][n]
```

Let dp[i][j] be the maximum coins gained from bursting all the balloons between index i(left) and j(right) in the original array, i and j are included. dp is a two-dimensional array.

The base case is when left==right, then return 0. There are no balloons in the middle to burst.

To get the maximum value we can get for bursting all the balloons between [i,j], we just loop through each balloon between these two indices [i,j] and make them to be the last balloon to be burst and choose the one with maximum coins gained. Let the index of last balloon to be burst is k between [i,j], dp[i][j] =

max { dp[i][j], nums[i-1]*nums[k]*nums[j+1]+dp[i][k-1]+dp[k+1][j] }. nums[i-1]*nums[k]*nums[j+1] is the coined gained when k is the last balloon to be burst. dp[i][k-1] is the maximum coins gained on the left list, and dp[k+1][j] is the maximum coins gained on the right list. For each index k between i and j($i \leq k \leq j$), we need to find and choose a value of k(last balloon to be burst) with maximum coins gained, and update dp array. In the end, we want to find out dp[0][n-1], which is the maximum value we can get when we burst all the balloons between [0, n-1].

The pseudo-code is shown as above. In the pseudo-code, we added nums[-1] and nums[n] to nums. So the return will be dp[1][n] due to the index change. The time complexity is $O(n^3)$.

Rubric: (15 points)

- (5 points) The algorithm is stated clearly.
  - 5 points if the algorithm is using dynamic programming algorithm and it is correct.
  - 0 points if the algorithm is totally not correct or the algorithm is not using dynamic programming.
- (3 points) The recurrence relation for sub-problems is correct.
- (5 points) The pseudo-code is clearly stated and correct.
- (2 points) The run time complexity is correct.

6. 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 dome in $O(1)$ time.

Solution:

The idea is simple, we consider each prefix and search it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix). If the recursive call for suffix returns true, we return true, otherwise we try next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.

Why Dynamic Programming? The above problem exhibits overlapping sub-problems

Let $s_{i,k}$ denote the substring $s_i s_{i+1} ... s_k$. 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.

Efficient implementation:

We build the OPT array using matched-index array that holds the indexes for which OPT[i] is true. When the algorithm is done, last element of OPT shows whether the whole input string can be segmented or not.

```
s = input string
n = length of s
OPT = array of length n holding possibility of segmentation up to index i
matched-index = array holding the indexes for which OPT[i] is true
if n == 0
  return True
initialize OPT array with 0
initialize matched-index with -1

for i from 0 to n
  msize = matched-index.size
  flag = 0

  for j from msize-1 to 0
    sb = s.substring(from: matched-index[j] + 1, length:i - matched-index[j])
    if dictionary contains sb
      f = 1
      break

  if f == 1
    OPT[i] = 1
    push i on matched-index

return OPT[n-1]
```

Rubric: (20 points)

- (10 points) clear description of the algorithm
- (3 points) Objective function: OPT(k) = denote whether the substring $s_{1,k}$ can be segmented using the words in the dictionary
- (3 points) Computation of OPT(i) for all i
- (1 point) The fact that last element of OPT contains the solution (can either be explicitly mentioned or inferred from the solution)

- (3 points) Runtime complexity analysis, result $O(n^3)$ in this case since we need O(n) to get substring.

7. A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as ai. Here, ai could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community requires that the tourism company do their boat tour business on a contiguous sequence of the river segments (i.e., if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money). The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design a dynamic programming algorithm to achieve this goal and analyze its runtime.

Solution:

This is a framing of the well known Largest Sum Contiguous Subarray problem.

Dynamic Programming Solution MS = array that holds maximum sum ending at each index

a = input (profits)
iarray = array holding beginning index for each MS
MS(0) = 0;

for j = 1, ... , a.length do
  $MS[j]$ = Math.max($MS[j-1] + a[j-1], a[j-1]$);
  if $MS[j-1] + a[j-1] > a[j-1]$
    $iarray[j] = iarray[j-1]$
  else
    $iarray[j] = j - 1$

$result = MS[0]$
$finali = 0$
$finalj = 0$

for j = 1, ... , a.length do
  if $result < MS[j]$
    $result = MS[j]$
    $finali = iarray[j]$

$finalj = j$

return $result, i, j$

Rubric: (15 points)

- (4 points) Objective function: MS(i) = maximum sum ending at index i
- (5) Recursive formulation: to calculate the solution for any element at index "i" has two options
  - (3 points) EITHER added to the solution found till "i-1"th index
  - (2 points) OR start a new sum from the index "i".
- (3 point) recording the beginning of each MS (iarray in pseudo code)
- (2 points) handling the final-i , final-j indices
- (1 point) Computing runtime complexity = $O(n)$.