Prof. Sergio A. Alvarez
21 Campanella Way, room 569
Computer Science Department
Boston College
Chestnut Hill, MA 02467 USA

http://www.cs.bc.edu/∼alvarez/
alvarez@cs.bc.edu
voice: (617) 552-4333
fax: (617) 552-6790

CS383, Algorithms
Spring 2009
HW8 Solutions

1. Consider the dynamic programming algorithm for the computation of the Fibonacci populations $F(n)$ that we discussed in class, shown in Algorithm 1.

Algorithm 1: Fibonacci numbers via dynamic programming.
Input: Integer $n \geq 0$.
Output: The Fibonacci rabbit population $F(n)$ in the n-th generation.

```
dpFib(n)
(1)     allocate table(0...n)
(2)     table(0) = 0, table(1) = 1
(3)     foreach i = 2...n
(4)         table(i) = table(i − 1) + table(i − 2)
(5)     return table(n)
```

(a) Determine the simplest possible big Θ expression for the running time of Algorithm 1 in terms of n, assuming that addition of two d-digit operands takes time $\Theta(d)$. Explain in detail. You may assume that the Fibonacci numbers $F(n)$ grow at the asymptotic rate $\Theta(b^n)$ for some $b > 1$.

Solution

Since $F(n) = \Theta(b^n)$, there are positive, finite constants A and B with $Ab^n \leq F(n) \leq Bb^n$ for all $n \geq 0$. Therefore, the number of digits needed to represent $F(n)$ is between $\log A + n \log b$ and $\log B + n \log b$. More succintly, the number of digits in $F(n)$ is $\Theta(n)$. Thus, the time needed to compute the sum $F(i-1) + F(i-2)$ on line 4 is $\Theta(i)$, and there are positive, finite constants A' and B' such that the time for the sum is between $A'i$ and $B'i$. Adding over all values of i, we see that the total time for the for loop is

$$\Theta\left(\sum_{i=2}^{n} i\right) = \Theta(n^2)$$

This is the dominant time in the entire algorithm, and so the net running time is $\Theta(n^2)$.

(b) Based on your analysis in 1a, is the dynamic programming solution in Algorithm 1 actually faster asymptotically than straightforward recursive implementation of the Fibonacci recurrence relation, or does the increasing time needed for the additions of increasingly large operands swamp the runtime to the point that the two algorithms are asymptotically equivalent? Explain.

Solution

Dynamic programming is still *much* faster than recursion, as the former runs in quadratic time and the latter requires exponential time as we described in class.

2. Consider driving from one street corner to another in a city covered by a grid of perpendicular streets. Any *monotonic* path from the source location to the destination is allowed; that is, each corner along the path should be closer to the destination than the preceding one. Paths should proceed along existing streets, without cutting across a block diagonally. You will design a dynamic programming algorithm that counts how many different such paths exist.

   (a) Let $P(n, m)$ denote the total number of monotonic paths between two street corners that are separated by n North-South "avenues" and m East-West "streets". For example, $P(1, 1)$ is the number of monotonic paths between two street corners located at diagonally opposite points of a city block; therefore, $P(1, 1)$ equals 2. Derive a recurrence relation for $P(n, m)$ by considering all street corners that can occur on the path one block away from the destination. Explain your thinking.

   Solution

   Any monotonic path that reaches (n, m) will have visited either $(n - 1, m)$ or $(n, m - 1)$ one block earlier. Conversely, any monotonic path that reaches either one of the latter two points will lead to a distinct monotonic path that reaches (n, m) one block later, simply by adding to it the final block to (n, m). Therefore:

   $$P(n.m) = P(n - 1, m) + P(n, m - 1)$$

   (b) Based on the recurrence relation for $P(n, m)$ that you derived above, sketch the domain of dependence of a given point $(n0, m0)$ in the (n, m) plane for that recurrence relation. Explain.

   Solution

   The domain of dependence is the rectangle with opposite corners at the points (n, m) and $(0, 0)$.

   (c) Describe suitable boundary cases for the recurrence relation for $P(n, m)$. Clearly state what the value of $P(n, m)$ is for each boundary point (n, m). Explain, and indicate the locations of the boundary points in your sketch above.

   Solution

   There is only one monotonic path from $(0, 0)$ to any point of one of the forms $(i, 0)$ and $(0, j)$. Therefore:

   $$P(i, 0) = 1 = P(0, j) \quad \text{for } 0 \leq i \leq n,\ 0 \leq j \leq m$$

   (d) Using your work in the preceding parts of this task, write detailed pseudocode for a dynamic programming algorithm that computes the number of different monotonic paths between two street corners in a city as described above.

Solution

See Algorithm 2.

Algorithm 2: Dynamic Programming Paths Counting

Input: Positive integers n, m.

Output: The number of distinct monotonic paths from $(0, 0)$ to (n, m).

```
dpPaths($n, m$)
(1)   allocate $P[0...n, 0...m]$
(2)   foreach $i = 0...n$
(3)      $P[i, 0] = 1$
(4)   foreach $j = 0...m$
(5)      $P[0, j] = 1$
(6)   foreach $j = 1...m$
(7)      foreach $i = 1...n$
(8)         $P[i, j] = P[i-1, j] + P[i, j-1]$
(9)   return $P[n, m]$
```

(e) Determine the big Θ running time of your dynamic programming algorithm for computing $P(n, m)$. Explain your analysis in detail.

Solution

Assuming that each addition takes time $\Theta(1)$, the overall running time is dominated by the update time (loop), which is $\Theta(nm)$.

3. In this task you will design an efficient dynamic programming algorithm for the maximum consecutive subsequence sum task (MSS) from HW6. Recall that the input is an array of n real numbers, each of which can be positive, zero, or negative. The output is the maximum sum that can be obtained by adding some set of consecutive elements of the input array. Consecutive here means that the elements occupy some segment $i, i+1, ...i+k$ of consecutive positions of the array, not that the elements' values are related in any way. *In the present task, only the maximum sum needs to be returned, not the index range of the original array over which that sum is attained.*

   (a) Define a hierarchy of subproblems $P(1)...P(n)$ that will allow the solution of the MSS task for the original input $a[1...n]$ to be computed. *It is of crucial importance to define the subproblems in the right way in order to allow the solution of the target problem to be obtained readily in terms of the solutions of smaller subproblems in the hierarchy.* Provide a concise definition of $P(i)$ in terms of i.

   Solution

   We define subproblem $P(i)$ as that of determining the maximum consecutive subsequence sum over the i-th prefix $a[1...i]$ of the input array, *considering only index ranges that end at (and include) the index i.* We define for each i between 1 and n the corresponding quantity S_i to be the value of the objective function evaluated on the i-th subproblem, namely the maximum sum that is possible by adding the values $a[j]$ over some subset of consecutive indices j ending at i. Note that the solution to the original problem is the maximum of the n values $S_1 ... S_n$, since the index subset that leads to the largest sum could end anywhere in the range $1...n$.

(b) Derive a recurrence relation that expresses the solution of the MSS task for $a[1...n]$ in terms of the solutions of smaller subproblems that belong to the subproblem hierarchy from 3a. Specify suitable boundary conditions for the recurrence. Explain.

Solution

Suppose that S_{i-1} is known. How can S_i be determined? There are two types of consecutive index sets ending at i: i by itself, and some set ending at $i-1$ together with i. However, S_{i-1} gives the sum over the best set of the second type. Hence, only two values need to be compared for index sets that end at i: the value $a[i]$ corresponding to i by itself, and the value $S_{i-1} + a[i]$ corresponding to the largest sum over a set of the second type. Which of these two values is larger? The answer just depends on the sign of S_{i-1}. If $S_{i-1} > 0$, then the sum $S_{i-1} + a[i]$ will be larger than $a[i]$; otherwise, $a[i]$ will be at least as large as $S_{i-1} + a[i]$. This yields the recurrence relation that we need:

$$S_i = \begin{cases} S_{i-1} + a[i], & \text{if } S_{i-1} > 0 \\ a[i], & \text{if } S_{i-1} \leq 0 \end{cases}$$

This assumes that $i > 1$.

Next, we determine appropriate boundary conditions. For $i = 1$, we have the value $S_1 = a[1]$, the only possible sum over a subset of indices ending at $i = 1$.

We identify the domain of dependence of S_i by examining the recurrence relation. Clearly, S_i depends on S_{i-1}, which in turn depends on S_{i-2}, and so on. Hence, the full domain of dependence for S_i includes all indices j between 1 and $i-1$. The space of subproblems here is linear, indexed by i.

(c) Write detailed pseudocode for an $O(n)$ dynamic programming solution to the MSS task, using your work from 3a and 3b.

Solution

We just need to decide on an update order for S_i for which the domain of dependence of S_i will have been completely updated before attempting to update S_i. In the present context the only possibility is to start at $i = 1$ and increment i by one at each step.

Combining all of the above ingredients, we now obtain Algorithm 3.

(d) What is the running time of your algorithm from 3c in terms of n? Provide the simplest possible big Θ expression. Explain your analysis.

Solution

$\Theta(n)$ due to the update loop and max computation at the end, as we discussed in class. This assumes that each sum requires time $\Theta(1)$.

Algorithm 3: Maximum Consecutive Subset Sum
Input: Array $a[1...n]$ of real numbers.
Output: Maximum sum $\sum_{i=i_1}^{i_2} a[i]$ over a segment $i = i_1...i_2$ of consecutive indices.

```
MAXCSS($a$)
(1)    $S[1] = a[1]$
(2)    foreach $i = 2$ to $n$
(3)        if $S[i-1] > 0$
(4)            $S[i] = S[i-1] + a[i]$
(5)        else
(6)            $S[i] = a[i]$
(7)    return $\max(S[1]...S[n])$
```