Intro to Algorithms, COMP-160, Homework #7

Benjamin Tanen, 03/17/2016

1. You and your friend are presented with a row of n buckets, each containing some amount of candy. Suppose that each amount is some real number. You both agree to take turns as follows: first you will keep either the first or the last bucket. Then your friend will keep the first or last of what remains. This will continue until you have partitioned the buckets.

   (a) **Provide a recursive formulation that returns the maximum amount of candy that you can acquire. This assumes that both you and your friend are playing as best as possible to maximize.**

   For this problem, consider the function $c(x)$ which returns the amount of candy in bucket x. This will be used in our recursive function $R(i, j)$, which returns the total candy obtained from playing the game on buckets i through j. The maximum amount of candy obtained for playing over n buckets would result from $R(1, n)$.

   We formulate the recursive relation of $R(i, j)$ by considering the different cases of the game. The first two are simple / obvious.

   i. Case 1 is if there is only one bucket ($i = j$). For this, we will obviously pick the one remaining bucket and get all of its candy $c(i) = c(j)$

   ii. Case 2 is where we only have two buckets left ($j - i = 1$). For this, we will obviously just pick the bucket with more candy

   iii. Case 3 is the interesting, non-base case where we have 3 or more buckets. We know that we want to maximize our candy and our friend wants to minimize our candy (thus maximizing his or her own). From this, we can think that in each move, we will maximize our total candy followed directly by our friend trying to minimize our candy. Thus, we can to pick from the two minimized options that our friend could pick. This then comes to another move by us, expressed by some $R(x, y)$. Since we know that we can take either the front or back and our friend can take either the front or back of what remains, there are four possible outputs to occur from our consecutive moves. Thus, we recursively consider four subproblems for each move to slowly reduce the number of buckets until we hit a base case.

   Using these, we can form the full relation for $R(i, j)$.

$$R(i,j) = \begin{cases} c(i) & \text{if } i = j \\ \max(c(i), c(j)) & \text{if } j - i = 1 \\ \max\left\{ \begin{array}{l} \min\left\{ \begin{array}{c} R(i+1, j-1) \\ R(i+2, j) \end{array} \right\} \\ \min\left\{ \begin{array}{c} R(i+1, j-1) \\ R(i, j-2) \end{array} \right\} \end{array} \right\} & \text{otherwise} \end{cases}$$

(b) **Analyze the time complexity of finding the maximum part (a), via plain recursion and then via dynamic programming.**

For the time complexity of plain recursion, we know that each recursion will branch into four subproblems (see the recursion relation above). From here, this splitting will repeat for each of our $\frac{n}{2}$ moves. If we think about this recursion in a tree, we would get a tree with height $\frac{n}{2}$, branching four each time. Since there is no way to halt this problem partially through, we know that our recursion tree will be a full tree. Therefore, we get a complexity of $\Theta(4^{\frac{n}{2}}) = \Theta(2^n)$.

For the time complexity of dynamic programming, we will utilize our memoization table (of size a-by-b where $a = n$ and $b = n$, the range of i and j respectively). We first calculate the base subproblems (when $i = j$, $j - i = 1$) and fill those spaces in our table, each taking constant time. From here, we can solve any problem that is dependent already solved subproblems. Thus, if we solve them in the right order, we can calculate any problem in constant time $\Theta(1)$. Since we only need to calculate half of the subproblems (when $i \leq j$), we can calculate all of the necessary subproblems in our table in $\Theta(1)\Theta(\frac{n^2}{2}) = \Theta(n^2)$ which is a drastic improvement over the plain recursion method.

(c) **How long does it take to figure out what your first move should be? What about the second move?**

From part b, we can see that the dynamic programming approach is significantly better than the plain recursion method, so we will use this. Using dynamic programming, in order to solve any particular problem we must solve its subproblems. Thus, in order to solve the problem of the first move, we must solve every other subproblem (every subsequent move), which would require us to completely fill out our memorization table. This takes $\Theta(n^2)$ time, so it takes the full $\Theta(n^2)$ time to pick the first move.

For the second move, we have two situations to consider. If we are trying to determine our second move before determining any other move, we would need to fully generate the table for all subproblems under move 2. This would take $\Theta(n^2)$ like the first move.

If we however want to determine the second move after we've already determined the first move, we would already have a completed memoization table. Therefore, in order to pick the best move would simple require querying our table at the four possible routes. This would take $4\Theta(1) = \Theta(1)$ time.

From this we can see that once we have calculated the ideal first move, determining any subsequent move takes constant time with dynamic programming.