FirstName LastName
StudentID acct@ucsc.edu

CMPS 102 — Winter 2019 – Homework 4

Three problems, 28 points, due 11:50 pm Wednesday March 6th, read the *Homework Guidelines*

1. (10 pts) Let B(n) be the number of different binary search trees containing the n keys 1, 2, . . . , n. For this problem you will develop a dynamic programming algorithm that, given n, calculates B(n). Note that B(1) = 1 since there is only one one-node binary tree. For two nodes B(2) = 2 (either node can be root, and there is only one binary search tree consistent with each choice).

   (a) (1 pt) Calculate by hand B(3).

   $B(3)$ is 5: there are 2 trees each 1 and 3 as the roots, and just one tree with 2 at the root. (Note that there are $6 = 3!$ orders in which the elements could be inserted, the insertion orders 2,3,1 and 2,1,3 give rise to the same BST.)

   (b) (1 pt) How many n = 6 key binary trees are there with keys $\{1, 2, 3, 4, 5, 6\}$ and key 3 at root (so the left subtree has two nodes, and the right one has 3 nodes) are there?

   There are 2 BSTs with just two elements, and from the first part, there are 5 BSTs with 3 elements. Each pair of choices for the left and right subtrees gives a different 6-key BST. Therefore there are $2 \cdot 5 = 10$ BSTs with $n = 6$ keys.

   (c) (3 pts) Construct a recurrence for B(n), I expect something with a sum. Include boundary conditions in your recurrence; what is a convenient boundary value for B(0)? Briefly justify (formal proof not required) that your recurrence computes the correct value.

   Clarification: this part is worth 4 points.

   Let $B(n)$ be the number of Binary Search Trees with n keys (call them 1 to n in order). Each key $i \in \{1, \ldots, n\}$ could be the root.

   First, some small values: $B(1) = 1$, $B(2) = 2$ and $B(3) = 5$.

   Assume first that some key i is the root, so there are $i - 1$ keys in the left subtree and $n - i$ keys in the right subtree. Thus there are $B(i - 1)$ possible left subtrees and $B(n - i)$ possible right subtrees, and each combination gives a different BST with i at the root. So when key i is the root there are $B(i - 1) \cdot B(n - i)$ different BSTs.

   One rough edge is when $i = 1$ or $i = n$. In these case we have 0 subtrees on one side or the other. In order to keep our multiplication rule (i.e. that there are $B(i-1) \cdot B(n-i)$ possible BSTs with key i at the root) in these cases we should define $B(0) = 1$, i.e. there is one binary search tree with no keys (the empty tree).

   Now, all BSTs on $n > 0$ keys have one of the keys at the root, and no two different BSTs have different roots. Therefore the number of BSTs on $n > 0$ keys is:

$$B(n) = \sum_{1 \le i \le n} B(i-1) \cdot B(n-i) \qquad \text{if } n \ge 1$$
$$B(0) = 1.$$

   (Note that this gives the right value for the small cases $n = 1, 2, 3$ above.)

(d) (4 pts) Give an iterative (bottom up) algorithm based on the recurrence that computes B(n) by filling in a table.

I will call my table B. The following fills in values up to n.

```
1: B(0) ← 1
2: for j = 1, ..., n do                          ▷ Calculate each B(j)
3:     B(j) = 0
4:     for r = 1, ..., j do                      ▷ add up over choices of root
5:         B(j) = B(j) + B(r − 1) · B(j − r)
6:     end for
7: end for
```

(e) (1 pt) What is the running time of your iterative algorithm as a function of n (use asymptotic notation)?

The running time is $\Theta(n^2)$, and $O(n^2)$ is also acceptable.

Fine point: The running time is dominated by the total number of iterations of the "for r" loop. There are n iterations of the "for j" loop, each with at most n iterations of the "for r" loop, so the running time is $O(n^2)$. Since the last $n/2$ of the "for j" loop iterations each have $n/2$ iterations of the "for r" loop, the total running time is $\Omega((n/2)^2) = \Omega(n^2)$. Therefore the total running time is $\Theta(n^2)$

2. (8 pts) Assume that you have a list of n home maintenance/repair tasks (numbered from 1 to n) that must be done *in list order* on your house. You can either do each task i yourself at a positive cost (that includes your time and effort) of $c[i]$. Alternatively, you could hire a handyman who will do the next 4 tasks on your list for the fixed cost h (regardless of how much time and effort those 4 tasks would cost you). You are to create a dynamic programming algorithm that finds a minimum cost way of completing the tasks. The inputs to the problem are h and the array of costs $c[1], \ldots, c[n]$.

(a) (3 pts) First, find a justify a recurrence (with boundary conditions) giving the optimal cost for completing the tasks.

Clarification: The handyman must be hired for exactly 4 tasks, so cannot be hired if fewer tasks remain. (Note that students may answer the harder version where the handyman can be hired even if fewer than 4 tasks remain).

Consider any optimal solution S to some n-task instance of the problem. Define $M(j)$ to be the minimum cost required to do the first j tasks. Since S is optimal, the cost of S is $M(n)$.

Optimal solution S either has the owner doing the last task, or the handyman doing the last task.

First consider the case that $n < 4$, since there are not enough tasks for the handyman, the owner must do all of them with cost the sum of the $c(i)$'s. (Note that if $n = 0$ the sum of the $c(i)$'s is also 0.)

Now assume that $n \geq 4$, and examine who does the last task.

Case 1: The owner does the last task. In this case the cost of the optimal solution is $c(n)$ plus the cost of solution S's way of doing all $n - 1$ previous tasks.

Claim: this must be the minimum cost for doing the $n - 1$ previous tasks.

Proof: Assume to the contrary that the previous $n-1$ tasks can be done more cheaply than they are done in S. Modifying S to do the remaining tasks in this cheaper way results in a solution S' that is cheaper than S, contradicting the optimality of S.

The cost of S is thus $M(n) = M(n-1) + c(n)$ in this case.

case 2: The handyman does the last task.

If the handyman does last task, the handyman must do the last 4 tasks. Since S is optimal, it must also contain a minimum-cost way of doing the remaining $n-4$ tasks. Otherwise one could hire the handyman for the last four tasks and use a cheaper way of doing the other $n-4$ tasks to contradict the optimality of S. This implies the cost of S is $M(n) = h + M(n-4)$ in this case.

Combining cases 1 and 2 gives:

$$M(n) = \min\{c(n) + M(n-1); h + M(n-4)\}$$

with the boundary condition that for $0 \leq n < 4$

$$M(n) = \sum_{1 \leq i \leq n} c(i)$$

Thus $M(0) = 0$.

Students may solve an earlier version of the problem where the handyman is allowed to be used for fewer than 4 tasks at the end of the sequence for full credit. There are a couple of ways to modify the solution if the handyman is allowed to finish the tasks even if there are less than 4 remaining. The difficulty is that the original problem and the subproblems are no longer the same: the original problem for tasks 1 through n can use the handyman on the last 2 or 3 tasks while the subproblems for tasks 1 through $j < n$ cannot.

One way is to first prove that no optimal solution has the owner doing a task j followed by the handyman doing all the remaining tasks (from $j+1$ to n) unless the number of remaining tasks is a multiple of 4. This can be proven by contradiction similar to the correctness of a greedy algorithm. The recurrence could then assume that the handyman only does tasks in groups of n, but an additional step at the end would be needed to consider the case where the handyman is hired to do all of the tasks.

Another way to handle this is to re-define the subproblems, letting $M[i]$ be the minimum cost to do tasks from i to n (rather than from 1 to i). Now the subproblems all end with the partial-completion flexibility and the recurrence would be derived by considering how an optimal schedule could do the first task (rather than the last one). This will result in a recurrence like: $M[j] = \min\{c[j] + M[j+1]; h + M[j+4]\}$ with the boundary conditions something like $M[j] = 0$ if $j > n$. In the iterative algorithm, the $M[]$ array would then be filled in from the back (i.e. $M[n]$ first and $M[1]$ last).

A third way is padding the original sequence of tasks with three "dummy" cost-0 tasks so there are now $n+3$ tasks. This padding will not change the cost of the optimal solution, but the dummy tasks give a way to assign the handyman exactly 4 tasks at the end.

(b) (1 pts) Give an $O(n)$-time recursive algorithm with memoization for calculating the value of the recurrence.

```
Initialize M(0) through M(3) as described above
return CALCM(n)

function CALCM(j)
    if M(j) initialized then
        return M(j)
    else                                          ▷ j must be at least 4
        M(j) = min{c(j) + CALCM(j − 1); h + CALCM(n − 4)}
        return M(j)
    end if
end function
```

(c) (1 pt) Give an $O(n)$-time bottom-up algorithm for filling in the array

```
Initialize M(0) through M(3) as described above
for j = 4, . . . , n do
    M(j) = min{c(j) + M(j − 1); h + M(n − 4)}
end for
```

(d) (2 pts) Describe how to determine which tasks to do yourself, and which tasks to hire the handyman for in an optimal solution.

Trace back through the calculated array $M()$ to find when the handyman is used.

```
1: function TRACEBACK(j)
2:     if j=0 then
3:         return
4:     else if j < 4 then
5:         print owner does first j tasks
6:         return
7:     else
8:         if M(j) = c(j) + M(j − 1) then
9:             TRACEBACK(j − 1)
10:            print owner does task j
11:        else
12:            TRACEBACK(j − 4)
13:            print hire handyman for tasks j − 3 to j
14:        end if
15:    end if
16: end function
```

(Note: this order of the **print** statements and TRACEBACK calls causes the information to printed in task order. This is not necessary for the assignment.)

3. (10 pts) Assume you are planning a canoe trip down a river. The river has n trading posts numbered 1 to n going downstream. You will start your trip at trading post number 1 and end at trading post number n. Let $R(i, j)$ be the cost of renting a canoe at trading post i and returning it at trading post j, where $j > i$. Assume that you always want to go down river, so the costs if $j \leq i$ are irrelevant. Find the cheapest sequence of rentals that allow you to complete your trip. Aim for an algorithm running in $O(n^2)$ time.

For example, if $n = 4$ and the costs are:

R(i,j)	j		
i	2	3	4
1	15	25	35
2	$--$	12	16
3	$--$	$--$	5

then the cheapest sequence of canoe rentals to travel the river would be to rent from 1 to 3, and then from 3 to 4 for a cost of $25 + 5 = 30$.

On the other hand, if the costs were:

R(i,j)	j		
i	2	3	4
1	20	15	30
2	$--$	5	10
3	$--$	$--$	20

then taking one canoe all the way from 1 to 4 and renting 1 to 2 and then 2 to 4 are the cheapest solutions (both cost 30). (Note that renting from 1 to 3 is cheaper than going 1 to 2, but the 1 to 3 rental is not in any of the cheapest 1 to 4 solutions.)

Let $C(k)$ be the cost of the cheapest sequence of canoe rentals starting from trading post 1 and returning the last canoe rented at trading post k.

(a) (3 pts) Assume an optimal sequence of rentals changes canoes at some trading post j. What subproblems are also solved optimally by (parts of) this rental sequence? Prove your answer (probably using a proof by contradiction)

Let S be the sequence of trading post rentals taken by an optimal solution The sequence of rentals taken from post 1 to post j in the S is an optimal sequence of rentals from post 1 to post j. Also, the sequence of rentals taken from post j to post n in S is also an optimal sequence of rentals from post j to post n.

Let $n > 1$ and suppose we have found an optimal sequence S taking us from 1 to n. Let j be any post at which canoes were exchanged, where $1 < j < n$. In other words our optimal rental sequence S consists of a sequence S_1 from 1 to j combined with a sequence S_2 from j to n.

Claim 1. *The subsequence S_1 of canoe rentals starting at 1 and ending at j is a cheapest way to get from post 1 to j.*

Proof. We prove this by contradiction. Assume that there is a cheaper way T to get from post 1 to post j. Then the sequence of rentals T followed by S_2 takes us from post 1 to post n at a cheaper cost than the sequence $S = S_1S_2$. But this contradicts the optimality of S.

Claim 2. *The subsequence S_2 of canoe rentals starting at 1 and ending at j is a cheapest way to get from post j to n.*

Proof. We prove this by contradiction (similarly to the above). Assume that there is a cheaper way T to get from post j to post n. Then the sequence of rentals S_1 followed by T takes us from post 1 to post n at a cheaper cost than the sequence $S = S_1S_2$. But this contradicts the optimality of S.

There are several ways to phrase this property leading to different dynamic programming algorithms. A common one will be to use the claim only for the last trading post j where an exchange of canoes was made. We use this variant for the rest of the problem.

(b) (2 pts) Derive a recurrence for $C(k)$ in terms of $C(j)$ values where $j < k$.

Any optimal sequence of rentals to post k makes a last rental at some trading post $j < k$. By taking the minimum over these possibilities we get:

$$C(k) = \begin{cases} 0 & \text{if } k = 1 \\ \min_{1 \le j < k}(C[j] + R[j,k]) & \text{if } 1 < k \le n \end{cases}$$

(c) (4 pts) Give a bottom-up iterative algorithm for computing the $C(j)$ values.

```
1: Initialize C[1] ← 0
2: for k ← 2, ..., n do                          ▷ calculate C(k)
3:     min ← R[1, k]                             ▷ store best value so far in min
4:     b[k] ← 1                                  ▷ b[k] stores best last rental to get to k
5:     for i ← 2, ..., k − 1 do                  ▷ consider i as last rental before k
6:         if C[i] + R[i, k] < min then          ▷ new best value so far
7:             min ← C[i] + R[i, k]
```

```
            b[k] ← i
        end if
    end for
    C[k] ← min
end for
```

Note: the $b[]$ array is for the next part.

(d) (1 pt) Finally, show how keeping a little (i.e. $O(n)$) additional information allows the a cheapest sequence of canoe rentals to be printed out in $O(n)$ time.

By keeping a best last rental to get to k (as with the $b[k]$ values computed in the above algorithm) one can quickly recover the entire optimal sequence of canoe rentals by calling TRACEBACK(n):

```
function TRACEBACK(j)
    if j=1 then
        return
    else
        TRACEBACK(b[j])
        print rent from b[j] to j
    end if
end function
```

For part (c), it may be helpful to first construct a recursive algorithm for computing the $C(k)$ values.