FirstName LastName
StudentID acct@ucsc.edu

# CMPS 102 — Spring 2020 – Homework 4

Five problems, due Saturday May 16 (on canvas) and May 25 (11:59 PM) on Gradescope.
Due class size, it is likely that only a subset of the problems will be graded.
**Draft version**: a final version and solution template will be posted in a couple of days.

Before you begin the assignment, please read the following carefully.

- Read the Homework Guidelines.
- Every part of each question begins on a new page. Do not change this.
- This does not mean that you should write a full page for every question. Your answers should be short and precise. Lengthy and wordy answers will lose points.
- Do not change the format of this document. Simply type your answers as directed.
- You are **not** allowed to work in teams.

I have read and agree to the collaboration policy. – Isai Lopez Rodas, ilopezro@ucsc.edu
Collaborators: None

---

1. (1 pt) Take the pre-quiz on Canvas by 11 PM on Saturday May 16th. This will be short quiz designed to test if you have read and understood the other problems. The quiz should be available by Friday afternoon.

2. (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 use a dynamic-programming like methodology (mostly memoization) to create an algorithm that, given $n$, calculates $B(n)$ efficiently. Note that $B(1) = 1$ since there is only one one-node binary tree. For two nodes $B(2) = 2$ (either node can be the root, and there is only one binary search tree consistent with each choice).

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

   **Solution.**

   *When the root is 1, there are two different trees that can be made. You can insert 1, 3, 2 OR 1, 2, 3 and it will create 2 seperate trees. When the root is 2, there is only one possible tree than can be made. The inserts can happen 2, 1, 3 OR 1, 3, 1. However, since the 3 > 1, 1 will always be inserted to the left and 3 will always be inserted to the right regardless of the order they come in. When the root is 3, there are 2 possible trees that can be made. The inserts can happen: 3, 1, 2 OR 3, 2, 1. This would create two seperate trees. Therefore the total number of possibilites is equal to 2 + 1 + 2 = 5 = B(3).*

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

**Solution.**

*Since the root of the tree is going to be key 3, there are 2 nodes on the left and 3 nodes on the right. Given what we know from the problem and what we found out in part $(a)$, we know that B(3) = 5 and B(2) = 2. Since the left subtree has 2 possible nodes and the right subtree has 3 possible subtrees, we can multiply $B(2) \times B(3)$ to find the total possibilies, which is 10.*

(c) (4 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.

**Solution.**

*Going back to the example brought up in part (c), in order to find a recurrence I will walk through the problem of finding B(4). I will start by finding the possible trees for B(4):*

i. *When the root is 1: That means that the right side of the tree will be populated by 3 randomly inserted nodes. We know that* ***B(3) = 5****.*
ii. *When the root is 2: The right side will only be able to fit one node 1 = B(1). The right side will have to randomly insert 2 nodes or B(2). Since those are the only possibilities, we can multiply them together, thus* ***B(2)*** $\times$ ***B(1) = 2****.*
iii. *When root is 3: The right side will only be able to fit one node: 1 = B(1). The left subtree has to randomly insert 2 nodes, which means again that the total possibilites for this is* ***B(2)*** $\times$ ***B(1) = 2****.*
iv. *When root is 4: We have to randomly insert 3 nodes on the left subtree since all nodes are smaller than 4. Therfore there are* ***B(3) = 5*** *total possibilites.*

*When we add up all the numbers in bold, we get that total number of binary trees for B(4) = 5 + 2 + 2+ 5 = 14.*

*I saw a pattern here. At the edges when root node is 1 or $n$, you multiply $B(0) \times B(n-1)$. In this case, I made $B(0) = 1$ since there is an empty tree when there are no nodes. The middle nodes are calculated by multiplying $B(n-i) \times B(i-1)$. At the end of calculating these values, we sum everything up. Therefore I came up with the following summation for the recurrence of $B(n)$:*

$$B(n) = \sum_{i=1}^{n} B(n-i) \times B(i-1)$$

(d) (3 pts) Give an iterative (bottom up) algorithm based on the recurrence that computes $B(n)$ by filling in a table. It may help to first create a recursive function with memoization from your recurrence above.

**Solution.**

*I will store all keys inside a dictionary. The lookup for a key is $\mathcal{O}(1)$ and we can dynamically allocate memory to the dictionary. The dictionary* dict *is a global variable. I will also store the initial keys 0 and 1 inside the dictionary and set their values = 1.*

```
procedure FindTotalTreesRec(n)                          ▷ n is the key
    if dict[n] exists then          ▷ Check to see if key $n$ is already in dictionary
        return dict[n]                ▷ will return value of $n$ if already calculated
    end if
    sum ← 0
    for $i = 1 \to n$ do
        num1 ← FindTotalTrees(n-1)
        num2 ← FindTotalTrees(n-i)
        sum +=( num1 × num2 )
    end for
    dict[n] ← sum        ▷ setting n = sum in dictionary before returning to memoize
    return sum
end procedure
```

*This is my recursive approach to the problem. Below, I will implement an iterative bottom up solution. Instead of using a global dictionary, I will not use a dictionary that is only defined within the scope of the function* FindTotalTreesIter(). *I will still initialize keys 0 and 1 to the values of 1. I chose a dictionary because you can dynamically allocate memory instead of worrying about not creating a big enough array.*

```
procedure FindTotalTreesIter(n)                         ▷ n is the key
    dict ← {0:1, 1:1}
    if $n < 2$ then
        return 1
    end if
    for $i = 2 \to n$ do
        for $j = 0 \to i$ do
            dict[i] += dict[j] * dict[i-j-1]
        end for
    end for
    return dict[n]
end procedure
```

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

**Solution.**

*In the worst case that $n$ has not been calculated yet, the algorithm has to run $\mathcal{O}(n^2)$. It has to run the entire $i = 1 \rightarrow n$ and recursively call for each part of the summation.*

3. (8 pts) Assume that you have a list of $n$ home maintenance/repair tasks (numbered from 1 to $n$) that must be done in numeric 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 for the fixed cost $h$ (regardless of how much time and effort those 4 tasks would cost you). The handyman always does 4 tasks, so cannot be used if fewer than four tasks remain. 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 and justify a recurrence (with boundary conditions) giving the the optimal cost for completing the tasks. Use $M(j)$ for the minimum cost required to do the first $j$ tasks.

**Solution.**

$$M(j) = \begin{cases} 0 & j = 0 \\ \sum_{i=1}^{j} c[i] & j <= 3 \\ \min(c[j-1] + c[j], c[j-4] + h) & otherwise \end{cases}$$

*This recurrence provides the optimal cost because it covers 3 cases assuming that j will always be non negative.*

i. *When j = 0: it will simply return 0 because there is no work to do*

ii. *When j $< 0 \leq$ 3: in this case, you still cannot hire a handyman so you must do all the work yourself. You have to sum up the amount of work you have to do for the j jobs.*

iii. *When j $\geq$ 4: you look at what the $j - 1^{th}$ job cost you and add the current job's cost and find the minimum between that and hiring a handyman, in which case we will calculate how much it cost for the previous i-4 jobs and add the handyman's fixed price. If you find the minimum between those two costs, you are guaranteed to find the optimal solution to which jobs we should perform or which jobs we should hand off to the handyman.*

(b) (2 pts) Give an $O(n)$-time recursive algorithm with memoization for calculating the $M(j)$ values.

**Solution.**

*Assume Costs array is a global variable*

```
procedure FINDOPTCOST(x)                                        ▷ x is representative of j
    Costs[x] ← [0] = 0                                          ▷ arr length x, with index 0 = 0
    if $x = 0$ then
        return 0
    else if $x < 4$ then
        Costs[x] = c[x] + FindOptCost(x-1)
        return Costs[x]
    else                                                        ▷ $x \geq 4$
        Costs[x] = $\min(FindOptCost(x-1) + c[x], FindOptCost(x-4) + h)$
        return Costs[x]
    end if
end procedure
```

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

**Solution.**

```
procedure FINDOPTCOST(x)                                    ▷ x is representative of j
    Costs[x] $\leftarrow$ [0] = 0                           ▷ arr length x, with index 0 = 0
    if $x = 0$ then
        return 0
    else if $x < 4$ then
        for $i = 1 \rightarrow x$ do
            Costs[i] = Costs[i-1] + c[i]
            return Costs[x]
        end for
    else                                                    ▷ $x \geq 4$
        Costs[1] $\leftarrow$ c[1]
        Costs[2] $\leftarrow$ c[2]
        Costs[3] $\leftarrow$ c[3]
        for $i = 4 \rightarrow x$ do
            Costs[i] = $\min(Costs[i-1] + c[i], Costs[i-4] + h)$
        end for
        return Costs[x]
    end if
end procedure
```

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

**Solution.**

*In order to determine which jobs will be done by the handyman and by myself, there should be some sort of dictionary or array that records the jobs in order that they were chose. For example, in the edge case where number of job is less than 3, the dictionary would be populated by 1, 2, 3 because that was the order in which the jobs were performed. If they are in increments of one, they are done by me. However, it gets a little more complicated when the number of jobs is greater than 3. The way I think it would work, is if the array gets populated with what number job was chosen. So if we have a total of 6 jobs and it was more optimal for me to do the first job, the handyman the next 4, and me the last one. I will put in the array 1, 2, 6. Since* $2 + 1 \neq 6$, *we can conclude that I did the first job, handyman did the next 4 jobs, and I did the last job (since 6 = total number of jobs).*

(optional challenge for the interested students, not to be turned in: solve the variant of the problem where the handyman can be used at cost $h$ to finish the last one, two, or three tasks on the list)

4. (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)

**Solution.**

*The subproblems being solved is finding the most cost efficient route between the previous trade post i to j and now solving to see which is the most optimal solution from j to a post k.*

*Assuming that j does not choose the next most optimal canoe trading post.*

*j has the option to go to next trading post k or $k'$. If the algorithm chooses to take the trading post $k$, the algorithm will continue to iterate until it finishes at the final trade post; however, this would not be a valid option because the algorithm has to choose the minimum value between trade post $k$ and $k'$. Therefore, we have reached a contradition.*

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

**Solution.**

*I will develop a similar recurrence as the one found in the lecture slides for the Knapsack problem.*

$$C(k) = \begin{cases} 0 & k = 1 \\ \min_{j<k}(C(j) + R(j,k)) & otherwise \end{cases}$$

(c) (4 pts) Give a bottom-up iterative algorithm for computing the $C(j)$ values. This algorithm may also compute "signpost" values for use in the following part.

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

**Solution.**

```
procedure COST(k)                                        ▷ k is target end post
    arrCosts[k+1]                                        ▷ Array used for memoization
    arrCosts[1] = 0                                      ▷ since R(1,1) = 0
    for $i = 2 \to k$ do
        arrCosts[i] = $R(1, i)$ = arrCosts[0]            ▷ storing the min value at index 0
        for $j = 1 \to i - 1$ do
            currMinCost = arrCosts[j] + $R(j, i)$
            if currMinCost $<$ C[i] then
                arrCosts[0] = currMinCost                ▷ set array at index 0 equal to smallest value
            end if
        end for
    end for
    return arrCosts[0]                                   ▷ this contains the smallest value
end procedure
```

(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.

**Solution.**

*The way I handled the printing of the result is by keeping an array of path the canoes must go in order to minimize the cost of the rentals. So if the path must go from 1-2 and 2-4, I added 2 and 4 to the array and iterate through that array to find the cost AND print the solution. This array takes an additional $\mathcal{O}(n)$ of auxillary space, but it does also allow the algorithm to print out result in the same time because its a single iteration of the array and a $\mathcal{O}(1)$ lookup for the cost.*

5. (10 pts) Consider the problem of detecting similar papers in a history course. If the assignment was on a particular topic, then one would expect that many of the key terms related to the topic would appear in most of the essays, but perhaps not in the same order. Let the sequence similarity of two essays be the length of the longest sequence of words such that both essays contain the sequence of words in the same order.

   For example, if one paper was "four score and seven years ago our fathers brought forth upon this continent, a new nation conceived in liberty, and dedicated to the proposition that all men are created equal." and another paper was "four years and seven days ago, fathers helped created a new nation with liberty and justice for all on the continent."

   One sequence I found that has many of the words in both essays in right order is: "four and seven ago fathers a new nation and all" (10 words). Replacing "all" with "the" gives another 10 words occurring in both sequences in order. Although "continent" and "years" also occur in both papers, they cannot be added to this sequence while preserving the sequence's order of occurrence.

   For this problem you are to create a dynamic programming algorithm that takes a papers $P$ and $Q$ as a lists/arrays of words $n$ and $m$ words respectively (possibly with repeats) and determines a longest sequence of words that occur in both papers in the same order.

(a) (4 pts) First focus on the length of the longest sequence of words occurring in both papers in order. Derive a recurrence for this length by considering what can happen with the last words in the papers. What are the boundary conditions for your recurrence? Give a brief rational why the recurrence gives the right values.

**Solution.**

(b) (3 pts) Give a bottom-up dynamic programming algorithm based on your recurrence that calculates the value of the optimal solution (i.e. the length of the longest sequence of words occurring in both lists in the same order). This algorithm should fill in a table.

**Solution.**

(c) (1 pt) What is the running time of your dynamic programming algorithm? (use asymptotic notation, assume that any two words can be compared in $O(1)$ time)

**Solution.**

(d) (2 pts) Describe how to output an actual longest sequence of words occurring in both lists in the same order. What is the (asymptotic) worst-case running time of your output method (assuming the table from part (b) has already been calculated).

**Solution.**