CS5800: Algorithms — Spring ’21 — Virgil Pavlu

Homework 6
Submit via Gradescope

Name: Shri Datta Madhira (NUID: 001557772)
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(40 points + Extra Credit 40 points)***

Jars on a ladder problem. Given a ladder of n rungs and k identical glass jars, one has to design an experiment of dropping jars from certain rungs, in order to find the highest rung (HS) on the ladder from which a jar doesn't break if dropped.

Idea: With only one jar (k=1), we can't risk breaking the jar without getting an answer.So we start from the lowest rung on the ladder, and move up. When the jar breaks, the previous rung is the answer; if we are unlucky, we have to do all n rungs, thus n trials.Now lets think of k=log(n): with log(n) or more jars, we have enough jars to do binary search, even if jars are broken at every rung. So in this case we need log(n) trials.Note that we can't do binary search with less than log(n) jars, as we risk breaking all jars before arriving at an answer in the worst case.

Your task is to calculate *q = MinT(n, k)* = the minimum number of dropping trials any such experiment has to make, to solve the problem even in the worst/unluckiest case (i.e., not running out of jars to drop before arriving at an answer). *MinT* stands for Minimum number of Trials.

A(5 points). Explain the optimal solution structure and write a recursion for MinT(n, k).
B(5 points). Write the alternative/dual recursion for MaxR(k, q) = the Highest Ladder Size n doable with k jars and maximum q trials. Explain how MinT(n, k) can be computed from the table MaxR(k, q). MaxR stands for the Maximum number of Rungs.
C(10 points). For one of these two recursions (not both, take your pick) write the bottom-up non-recursive computation pseudocode. *Hint: the recursion MinT(n, k) is a bit more difficult and takes more computation steps, but once the table is computed, the rest is easier on points E-F below. The recursion in MaxR(q, k) is perhaps easier, but trickier afterwards: make sure you compute all cells necessary to get MinT(n, k) — see point B.*
D(10 points). Redo the computation this time top-down recursive, using memoization.
E(10 points). Trace the solution. While computing bottom-up, use an auxiliary structure that can be used to determine the optimal sequence of drops for a given input n, k. The procedure TRACE(n, k) should output the ladder rungs to drop jars, considering the dynamic outcomes of previous drops. *Hint: its recursive. Somewhere in the procedure there should be an if statement like "if the trial at rung x breaks the jar... else ..."*
F(extra credit, 20 points). Output the entire decision tree from part E) using JSON to express the tree, for the following test cases : (n=9,k=2); (n=11,k=3); (n=10000,k=9). Turn in your program that produces the optimum decision tree for given n and k using JSON as described below. Turn in a zip folder that contains: (1) all files required by your program (2) instructions how to run your program (3) the three decision trees in files t-9-2.txt, t-11-3.txt and t-10000-9.txt (4) the answers to all other questions of this homework.

To represent an algorithm, i.e., an experimental plan, for finding the highest safe rung for fixed n,k, and q we use a restricted programming language that is powerful enough to express what we need. We use the programming language of binary decision trees which satisfy the rules of a binary search tree. The nodes represent questions such as 7 (representing the question: does the jar break at rung 7?). The edges represent yes/no answers. We use the following simple syntax for decision trees based on JSON. The reason we use JSON notation is that you can get parsers from the web and it is a widely used notation. A decision tree is either a leaf or a compound decision tree represented by an array with exactly 3 elements

```
// h = highest safe rung or leaf
```

```
{ "decision_tree" : [1,{"h":0},[2,{"h":1},[3,{"h":2},{"h":3}]]] }
```

The grammar and object structure would be in an EBNF-like notation:

```
DTH = "{" "\"decision_tree\"" ":" <dt> DT.
DT = Compound | Leaf.
Compound = "[" <q> int "," <yes> DT "," <no> DT "]".
Leaf = "{" "\"h\" " ":" <leaf> int "}".
```

This approach is useful for many algorithmic problems: define a simple computational model in which to define the algorithm. The decision trees must satisfy certain rules to be correct.

A decision tree t in DT for HSR(n,k,q) must satisfy the following properties:
(a) the BST (Binary Search Tree Property): For any left subtree: the root is one larger than the largest node in the subtree and for any right subtree the root is equal to the smallest (i.e., leftmost) node in the subtree.
(b) there are at most k yes from the root to any leaf.
(c) the longest root-leaf path has q edges.
(d) each rung 1..n-1 appears exactly once as internal node of the tree.
(e) each rung 0..n-1 appears exactly once as a leaf.

If all properties are satisfied, we say the predicate correct(t,n,k,q) holds. HSR(n,k,q) returns a decision tree t so that correct(t,n,k,q).

To test your trees, you can download the JSON HSR-validator from the url:
`https://github.com/czxttkl/ValidateJsonDecisionTree`

G(extra credit, 20 points). Solve a variant of this problem for q= MinT(n,k) that optimizes the average case instead of the worst case: now we are not concerned with the worst case q, but with the average q. Will make the assumption that all cases are equally likely (the probability of the answer being a particular rung is the same for all rungs). You will have to redo points A,C,E specifically for this variant.

Solution:

A.

$$MinT(n,k) = \begin{cases} i & \text{if MinT(i,j)} \geq \text{n} \\ 1 + MinT(i-1, j-1) + MinT(i-1, j) & \text{for i,j} \leq \text{n,k} \end{cases}$$

B.

$$MaxR(k,q) = \Big\{ 1 + MaxR(i-1, j-1) + MaxR(i-1, j) \quad \text{for i,j} \leq \text{k,q}$$

With a maximum of 'q' trails, we can observe that we can cover 1+2+3+...+(q-1)+q rungs. This is nothing but the sum of first 'q' natural numbers. Therefore,

$$MaxR(k,q) = \frac{q*(q+1)}{2}$$

So, the minimum number of dropping trails(MinT(n,k), let us assume it is 'q' for now) required can be calculated by observing that the sum of first 'k' natural numbers in a 'n' number sequence is always ≥ 'n' itself. Applying it to the above equation, we get

$$MaxR(k,q) \geq n \Rightarrow \frac{q*(q+1)}{2} \geq n \Rightarrow \frac{q^2+q}{2} \geq n \Rightarrow q^2+q \geq 2n \Rightarrow q^2+q-2n \geq 0$$

Using the formula $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ on the above equation we get (We ignore the negative cases because they are not applicable),

$$q \; or \; MinT(n,k) = \frac{-1+\sqrt{1^2-4(1)(-2n)}}{2(1)}$$

$$MinT(n,k) = \frac{-1+\sqrt{1+8n}}{2}$$

We will get the value of 'n' from MinT(n,k). We can substitute that value and get the value of MinT.

C.

Algorithm 1: Algorithm for jars on a ladder problem.

```
Function JAR-LADDER-MinT(n,k):
    m[0 ⋯ n+1, 0 ⋯ k+1] = 0
    For i = 1 to n+1
        For j = 1 to k+1
            m[i,j] = 1 + m[i-1,j-1] + m[i-1,j]
            If m[i][j] ≥ n :
                Return i,m
```

D.

Algorithm 2: Algorithm for jars on a ladder problem.

```
m[0 ··· n+1, 0 ··· k+1] = 0
Function JAR-LADDER-MinT(n,k,m):
    If n == 1 or n == 0 :
        Return n
    If k == 1 :
        Return n
    If m[n,k] ≠ 0 :
        Return m[n,k]
    minT = float(”inf”)
    For i = 1 to n+1
        temp = 1 + JAR-LADDER-MinT(i-1,k-1,m)+JAR-LADDER-MinT(i-1,k,m)
        If temp < minT :
            minT = temp
    m[n][k] = minT
    Return minT
```

E.

Algorithm 3: Algorithm to Trace solution for jars on a ladder.

```
Function TRACE(n,k,m,i):
    If i == 0 :
        Return
    TRACE(n,k,m,i-1)
    print(”Floor:”, m[i][k])
```

F.

G.

2. ***(20 points)*** *Problem 15-2. Hint: try to use the LCS problem as a procedure.*

Solution:

Step 1: Characterizing the problem

This problem exhibits optimal substructure. To get the longest palindrome sequence for 'n' characters in the string, we have to find the longest palindrome sequence for the 'n-1' characters before that. This solution is optimal because if there is another optimal sub-problem solution for the current main solution, we can just plug in the new solution and get an even better main solution, which contradicts the optimality of the original optimal solution.

Step 2: Recurrence Objective

$$c[i,j] = \begin{cases} 0 & \text{if i = 0 or j = 0,} \\ c[i-1,j-1]+1 & \text{if i,j > 0 and } x_i = y_j, \\ max\{c[i,j-1], c[i-1,j]\} & \text{if i,j > 0 and } x_i \neq y_j \end{cases}$$

Step 3: Algorithm for the problem

Algorithm 4: Algorithm to find longest palindrome sequence.

```
Function LCS-PALINDROME(X):
    Y = X.reverse
    n = X.length
    For i = 1 to n
        c[i,0] = 0
    For j = 0 to n
        c[0,j] = 0
    For i = 1 to n
        For j = 1 to n
            If $x_i == y_j$ :
                c[i,j] = c[i-1, j-1] + 1
                b[i,j] = (i-1, j-1)
            ElseIf c[i-1,j] ≥ c[i,j-1] :
                c[i,j] = c[i-1,j]
                b[i,j] = (i-1, j)
            Else
                c[i,j] = c[i,j-1]
                b[i,j] = (i, j-1)
    Return b,c
```

Step 4: Tracing the solution

Algorithm 5: Algorithm to Trace solution for longest palindrome sequence.

```
Function SOL-TRACE(b,X,i,j):
    If i == 0 or j == 0 :
        Return
    If b[i,j] == (i-1,j-1) :
        SOL-TRACE(b,X,i-1,j-1)
        print $x_i$
    ElseIf b[i,j] == (i-1,j) :
        SOL-TRACE(b,X,i-1,j)
    Else
        SOL-TRACE(b,X,i,j-1)
```

Step 5: Time and Space Complexities

Time Complexity: It takes $O(m+n)$ time.
Space Complexity: $\Theta(n^2)$ to store 'b' and 'c' matrices.

3. ***(30 points)*** *Exercise 15.4-6.*

Solution:

Step 1: From the recursive code I wrote in the last assignment, I realized that this problem can be solved easily using dynamic programming by tabulating the results we got from the sub-sequences solved before. This problem exhibits dynamic programming because the optimal solution for the main problem depends on the optimal solutions for the sub-problems, i.e., we get the maximum desired length of sub-sequence only when we get the maximum length at each sub-problem. If there is a solution optimal than the solution we have for any sub-problem, then we can plug in that solution and get an even better solution which contradicts the argument.

In the algorithm below, we will initiate a new array 'm' with the first element of the main array. Then, we iterate from 1 through n, processing each element of the array. When we receive an element, we will compare it to the last inserted element into the new array. If the current element is larger, we append it to the new array. Otherwise, we start from mid point of the new array and compare until start becomes greater than end, at which point we stop and attach the current element in that position in the new array. Finally, the length of the longest sub-sequence is equal to the length of the new array.

Step 2: Recurrence Relation

$$m[i] = \begin{cases} append(a[i]) & \text{if a[i]>m[-1]} \\ replace(a[i], m) & \text{if a[i]<m[0]} \\ find(a[i]) & \text{where a[i] > m[-1]} \end{cases}$$

Step 3:

Algorithm 6: Algorithm to find the longest increasing sub-sequence

```
Function LONGEST-SUB-SEQUENCE-nlgn(array):
    m = [array[0]]
    For i = 1 to array.length
        If array[i] > m[-1] :
            m.append(array[i])
        ElseIf array[i] ≤ m[-1] :
            m[0] = array[i]
        Else
            start, end = 0, array.length - 1
            While start ≤ end
                mid = ceil((start+end)/2)
                If array[i] < m[mid] :
                    end = mid - 1
                Else
                    start = mid + 1
            m[start] = array[i]
    Return m
```

Step 4: Tracing the Solution

Algorithm 7: Algorithm to trace the solution for longest increasing sub-sequence

```
Function TRACE-SOLUTION-LIS-nlgn($m$):
    print("The desired sub-sequence is:", m);
    print("The length of the desired sub-sequence is:", m.length);
```

Step 5: Discussing the asymptotic complexities
Time Complexity: It has 1 for loop to iterate through the given array of elements, and $\lg n$ for searching for suitable element in the array. So it takes $O(n \lg n)$ time.
Space Complexity: $O(n)$ for creating the array 'm'.

4. ***(Extra Credit 20 points)***

Suppose that you are the curator of a large zoo. You have just received a grant of \$$m$ to purchase new animals at an auction you will be attending in Africa. There will be an essentially unlimited supply of n different types of animals, where each animal of type i has an associated cost c_i. In order to obtain the best possible selection of animals for your zoo, you have assigned a value v_i to each animal type, where v_i may be quite different than c_i. (For instance, even though panda bears are quite rare and thus expensive, if your zoo already has quite a few panda bears, you might associate a relatively low value to them.) Using a business model, you have determined that the best selection of animals will correspond to that selection which maximizes your perceived profit (total value minus total cost); in other words, you wish to maximize the sum of the profits associated with the animals purchased. Devise an efficient algorithm to select your purchases in this manner. You may assume that m is a positive integer and that c_i and v_i are positive integers for all i. Be sure to analyze the running time and space requirements of your algorithm. **Solution:**

5. ***(20 points)*** *Problem 15-4.*

Solution:

Step 1: Characterizing the problem

This problem exhibits optimal substructure. In order to get the minimum number of extra spaces over all the lines, we have to get the minimum number of extra spaces for each and every line. That means, the optimal solution for the current line depends on the solutions for all the lines that are printed before that. This solution is optimal because if there is another optimal sub-problem solution for the current main solution, we can just plug in the new solution and get an even better main solution, which contradicts the optimality of the original optimal solution.

Step 2: Recurrence Objective

$$cost[i,j] = \begin{cases} (\text{M-j+i-}\sum_{k=i}^{j} l_k)^3 & \text{if i} \leq \text{j,} \\ (\text{M-1+l}_i)^3 & \text{if i == j,} \\ 0 & \text{if j == n (last word, that means last line).} \end{cases}$$

$$m[i,j] = \begin{cases} cost[i,i] & \text{if i == j,} \\ m[i-1,j] + cost[i,j] & \text{if i} \leq \text{j, j - i} \leq \text{M, and j} \leq \text{n.} \end{cases}$$

Step 3: Algorithm for the problem

Algorithm 8: Algorithm to print neatly.

```
Function PRINT-NEATLY(n, M):
    For i = 1 to n
        length = len[i]
        cost[i,i] = $(M - i + length)^3$
        For j = i+1 to n
            length += len[j]
            If j == n-1 :
                cost[i,j] = 0
            cost[i,j] = $(M - j + i - length)^3$
    For j = 1 to n
        m[j] = float("inf")
        For i = 1 to j
            If $j - i \leq M$ :
                temp = m[i-1] + cost[i,j]
                If temp < m[j] :
                    m[j] = temp,  s[j] = i
    Return m,s
```

Step 4: Tracing the solution

Algorithm 9: Algorithm to Trace solution for print neatly.

```
Function SOL-TRACE($m$,$s$,$n$):
    totalCost = 0
    If $n$ == $1$ :
        path = []
        path.append(s[1])
        Return path, m[1]
    path, cost = SOL-TRACE(m,s,s[n])
    totalCost = cost + m[n]
    path.append(s[n])
    Return path, totalCost
```

Step 5: Time and Space Complexities

Time Complexity: It takes $O(n^2)$ time.
Space Complexity: $\Theta(n^2)$ to store 'cost', 'm,' and 's' matrices.