CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 6

Name: Ketaki Nitin Kolhatkar
Collaborators: Madhusudan Malhar Deshpande

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. ***(no credit)*** *Write up a complete DP solution for Optimal BST*

2. ***(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`

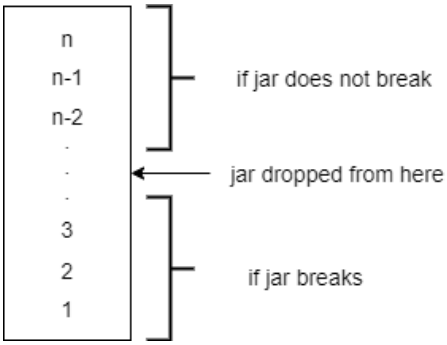
G(extra credit, 20 points). Solve a variant of this problem for q= MinT(n,k) that optimizes the average case instead of the worst case: now we are not concerned with the worst case q, but with the average q. Will make the assumption that all cases are equally likely (the probability of the answer being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

Solution:

(A) We drop the jar from a random place on the ladder. The jar will either break or it will not.

1. If the jar breaks, we don't need to check at the steps above, we need to start checking below with one less jar.
2. If the jar does not break, we don't need to check the at the steps below, we need to start checking at the top with all the jars.

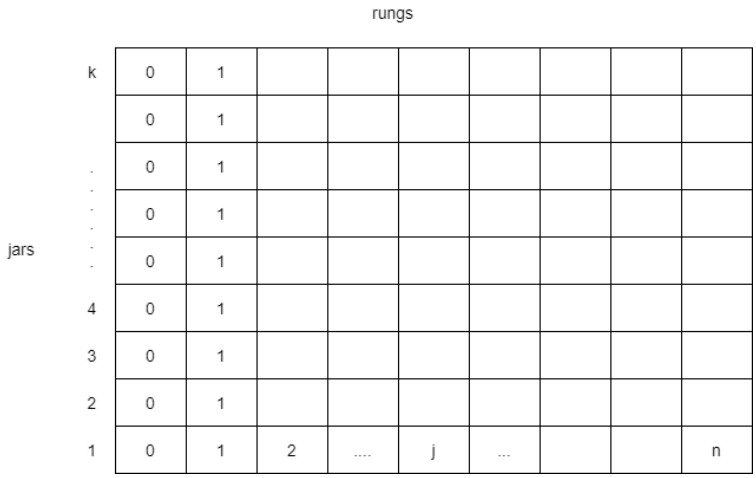
Let T(n,k) be a function that determines the Jar's breaking point in minimum number of



trials, in the worst case. Thus, the recursion is: T(n,k) = 1 + MinMaxT(i-1,k-1) ; T(n-i,k), where i goes from 1 to n. In this expression - MaxT(i-1,k-1) ; T(n-i,k) - 'Max' ensures that always worst case is chosen. And taking 'Min' of overall function gives us the optimal number of trials.

(B)

(C)



Let Let C[i,j] be a 2D array of dimensions k x n, where k = Jars and n = Rungs. C[i,j] will hold

the minimum number of trials required for finding the threshold in the worst case, for i Jars and j rungs.

```
T(n,k)
1. Initialize C[] of dimensions k*n
2. for i in range(1, k):
3.      C[i][0] = 0 //No trials required for 0 rungs
4.      C[i][1] = 1 //Only 1 trial required for 1 rungs
5. for j from 1 to n:
6.      C[1][j] = j //In case of 1 Jar, every rung has to be tried
7. for i from 2 to k:
8.      for j from 2 to n:
9.      C[i][j] = Infinity
10.     for p from 1 to j:
11.         if C[i][j] < 1 + max(C[i-1][p-1], C[i][j-p]):
12.             C[i][j] = 1 + max(C[i-1][p-1], C[i][j-p])
13. return C
C[k][n] holds the result for the minimum trials required in the worst case
```

Time complexity of the algorithm is $O(n * k^2)$, array C takes up $O(n * k)$

```
(D)Top-Down Recursion optimized using memoization:
T(n,k) // n = rungs and k = jars
1. Initialize C of dimensions n*k with all values = -1
2. if T(n,k) is not equal to -1:
3.      return C[n][k]
4. if n == 1 or n == 0:  required
5.      return n
5. if k == 1:
6.      return k
7. min = infinity
8. for p from 1 to n:
9.      Z = max(T(n-1,p-1), T(n-p,p)) //Recursive Call
10.     if Z < min: //add to C so result can be used in future calls
11.     min = Z
12. C[n][k] = min + 1
```

(E) Let C[i,j] and S[i,j] be a 2D array of dimensions k x n, where k = Jars and n = Rungs. C[i,j] will hold the minimum number of trials required for finding the threshold in the worst case, for i Jars and j rungs. S[i,j] will hold the rung where the Jar should be dropped from, when jars are equal to i and rungs are equal to j. We can use the recursion formula in the 'Question a' to compute the results for each rung. The recursion formula is: T(n,k) = 1 + MinMaxT(i-1,k-1) ; T(n-i,k)

```
T(n,k)
1. Initialize C[] and S[] of dimensions k*n
```

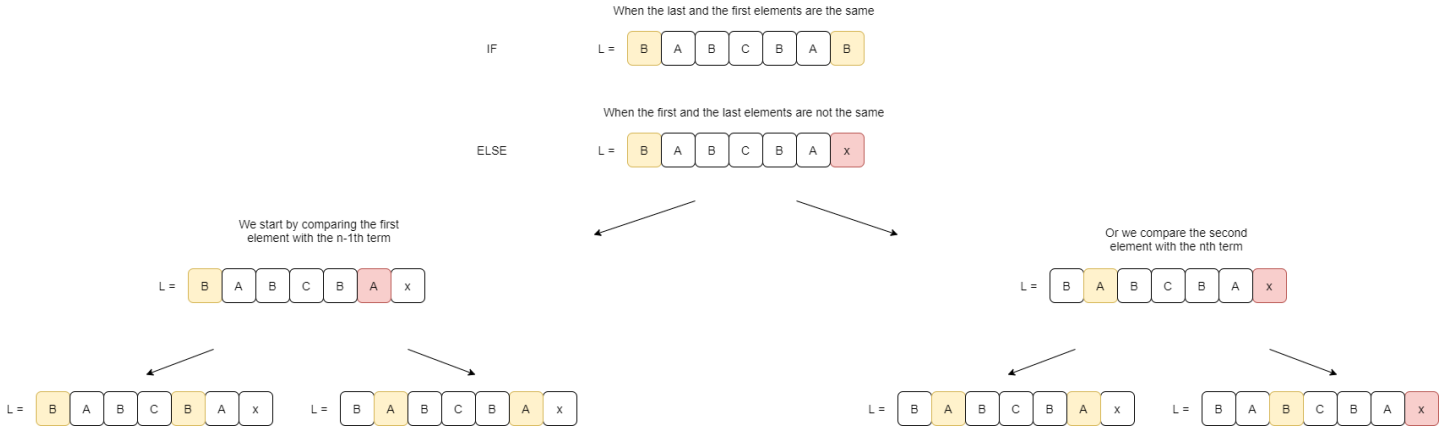
```
2. for i in range(1,k):
3.      C[i][0] = 0 //No trials required for 0 rungs
4.      C[i][1] = 1 //Only 1 trial required for 1 rungs
5. for j from 1 to n:
6.      C[1][j] = j //In case of 1 Jar, every rung has to be tried
7. for i from 2 to k:
8.      for j from 2 to n:
9.          C[i][j] = Infinity
10.         for p from 1 to j:
11.             if C[i][j] < 1 + max(C[i-1][p-1], C[i][j-p]):
12.                 C[i][j] = 1 + max(C[i-1][p-1], C[i][j-p])
13.                 S[i][j] = p
14. return C, S
C[k][n] holds the result for the minimum trials required in the worst case
S[k][n] holds the rung number from where the jar should be thrown to get
minimum trial count in the worst case, when rungs = n and jars = k.
Final Trace Solution:
Trace(S[n,k])
1. p = S[n,k] //throw from pth rung
2. print(p)
3. if Jar breaks:
4. Trace(S[p-1,k-1])
5. else:
6. Trace(S[p+1,k])
```

3. ***(20 points)*** *Problem 15-2. Hint: try to use the LCS problem as a procedure.*

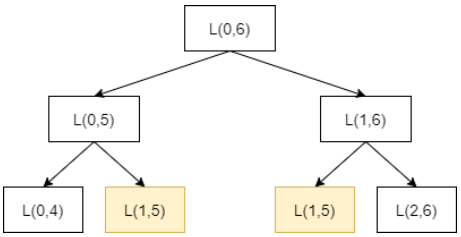
Solution:

Consider the string L = ”BABCBAB”.

1. Characterize the structure of the optimal solution:



2. Define the dynamic recurrence:
When we iteratively run through the algorithm, we find that there are sequences that are repeated and calculated more than once. For example, L(1,5) is repeated twice. Hence, we solve the problem dynamically.



3. Compute the value bottom up:
We start checking the string from the front end and the back end together. For example, we start by comparing the 1st element 'B' with the last element 'B'. If we get the characters same, we return the length of the string between the same characters and add 2 for adding the two characters. If we find that the two characters are not the same, then we compare the 1st character with n-1th character. And we also compare the 2nd character with the nth character. We do this iteratively until we find two characters same. We make this 7x7 matrix where we write the

```
LongestCommonPalindrome{a}
    p = reverse of q
    n = q.length
    Array C[0...n][0..n]
    Array B[0...n][0..n]
    for i=0 to n:
        C[i,0] = 0
    for j=1 to n:
        C[0,j] = 0
    for i=1 to n:
        for j=1 to n:
            if (q[i] == p[i]):
                C[i,j] = C[i-1,j-1] + 1
                B[i,j] = 'diag'
            else if ( C[i,j-1] > C[i-1,j] )
                C[i,j] = C[i,j-1]
                B[i,j] = 'left'
            else
                C[i,j] = C[i-1,j]
                B[i,j] = 'up'
return c,b
```

	0	1	2	3	4	5	6
0	1	1	3	3	3	5	7
1		1	1	1	3	5	5
2			1	1	3	3	3
3				1	1	1	3
4					1	1	3
5						1	1
6							1

The time complexity of the algorithm is $O(n^2)$.

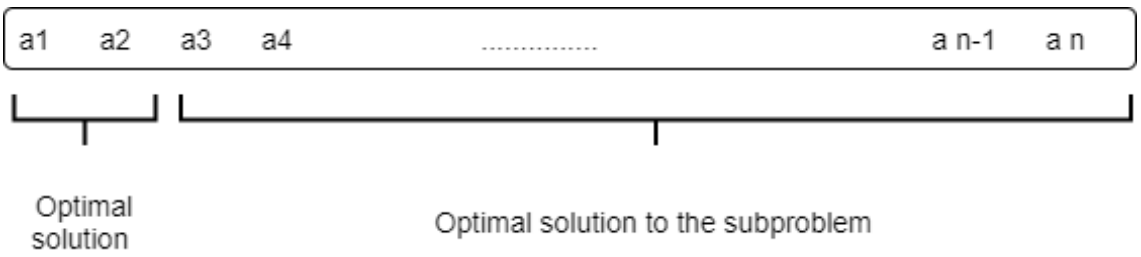
4. Trace the solution:

```
  The longest common palindrome in a string q can be solved as:
\begin{verbatim}
    LongestCommonPalindromeTraceback(b, q, i, j)
        if i == 0 or j == 0:
            return
        if b[i,j] == 'diag':
            print-LongestCommonPalindrome(b, q, i - 1, j - 1)
            print q[i]
        else if b[i, j] == 'left':
            print-LongestCommonPalindrome(b, q, i, j - 1)
        else:
            print-LongestCommonPalindrome(b, q, i - 1, j)
```

4. ***(30 points)*** *Exercise 15.4-6.*

Solution:

1. Characterizing the optimal solution :



2. Defining the dynamic recurrence: search for k, C[n] = C[k] + 1, C[k]: largest subsequence. Dependency is on the next biggest element.

3. Algorithm:

```
LIS(A[1:n])
1. Let S[i] be the an empty array
2. S[1] = A[1]
3. for i in range(2,n):
4.      if A[i] > S[i-1]:
5.      Add A[i] to array S[]
6. else:
7.      j = BinarySearch(S[1:i-1], A[i])
8. C[n] = length(S)
```

Time complexity of this algorithm is O(nlogn).

4. Trace Solution: S[] gives us the longest increasing subsequence, no need to trace the solution seperately.

5. ***(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:

6. ***(20 points)*** *Problem 15-4.*

Solution:

1. Characterizing the optimal solution: We need to fit the maximum number o characters in a given line with the number of extra spaces in the line.

2. Define the dynamic recurrence: The recurrence is C[1,j] = min(C[1, i-1]+p[i,j]), i[1:j]
Let C[1,j] = cost of arranging words 1 to j optimally, penalty, p[i,j] = extras $[i, j]^3$

3. Algorithm

```
PRINT_NEATLY(n,M,l)
```

```
1. //Calculate Extras and store in look up table
2. for i in range(1, n):
3.      e[i,i] = M - l_i
4.      for j from i+1 to n:
5.      e[i,j] = e[i,j-1] - l_j - 1
6.      if e[i,j] < 0:
7.      p[i,j] = infinity //words to don't fit on the line
8.  elseif e[i,j] >= 0 and j == n:
9.      p[i,j] = 0 //last line does not have any penalty
10. else p[i,j] = (e[i,j])^3
11. //Calculate Objective function C
12. Let C[] be the array storing minimum penalty of printing words from 1 to j
and S[] be the array storing the index of word starting the line that ends
with word j
13. C[0] = 0
14. for j in range(1, n):
15.     C[j] = infinity
16.     for i in range(1, j):
17.         if C[1,i-1] + p[i,j] < C[1,j]:
18.         C[1,j] = C[1,i-1] + p[i,j]
19.         S[j] = i
20. return C[], S[]
```

The time complexity of the algorithm is $O(n^2)$.

4. Trace Solution:

```
1. Let i = S[j]
2. If i > 1:
3.      k = BreakLines(S[], i-1) + 1
4. else:
5.      k = 1
6.      print(k,i,j)
```

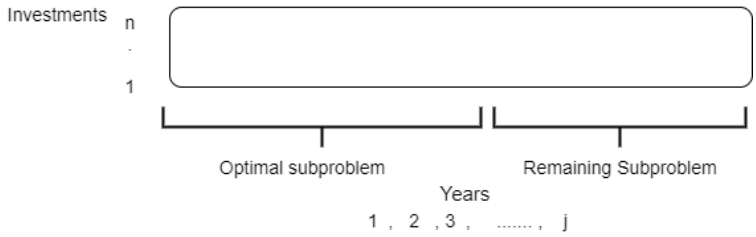
7. ***(20 points)*** *Problem 15-10.*

Solution:

(a) Exchange argument helps us prove this question. Consider a case where we invest P dollar is the 1st investment and Q dollars in the 2nd investment forthe first n years.
Then ri1 + ri1 + ri3 + ... + rik ¡ rj1 + rj2 + rj3 ... + rjk, we can then put the money from investment 1 to investment 2. By repeating these steps for different investments, we could get a solution close to the optimal solution by putting all the money in one investment.

(b) The problem exhibits optimal substructure since we have two options at the end of each year, either we continue the investment with a fee of f1 or move it to another investment with a fee

of f2. Hence, what we do in the next year is independent of the previous years. It does not need the prior knowledge of the problem.



(c)
1. Characterize the structure of the optimal solution: Assume S[] = [i1, i2, i3, ...i10] to be the optimal solution after 10 years.

2. Define the dynamic recurrence: Search investment in the 1st Year -
$C[1:n] = MaxC[i,1] + C[i,2:n]f1; C[q,1] + C[i,2:n]f2$
C[i,1] = return from investment i in year 1
C[q,1] = return from investment q (investment with highest return in a given year). We start solving from the right and go towards left owing to the dependencies.

3. Algorithm:

```
1. S[i] =  array indicating investment made in each year
2. C[j] =  array indicating total returns made from year i to 10
3. for j in range(10,1):
4.  q = 1
5.  for i in range(1,n):
6.      if rij > rqk:
7.      q = i
8.  if C[j+1] + [Return with same investment in prev year] - f1 > C[j+1] + [Return with best i
9.      C[j] = C[j+1] + [Return with same investment in prev year] - f1
10. S[j] = S[j+1]
10.     else:
11.         C[j] = C[j+1] + [Return with best investment q in prev year] - f2
12. return S[j]
```

The time complexity is $O(n^2)$.

(d) There is no limit up to which we we have to invest the money. Hence, we may have to shift the money to some other investment based on the money we have saved over a few years. Using the algorithm previously used, we would have a different starting money the next year. And there are infinite number of possibilities of having the starting money. Hence, the problem does not show optimal substructure property anymore.