CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 6

Name: Xuran Feng
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. ***(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).
The problem has optimal subproblem structure. If dropping the jar from a certain rung jth and the jar doesn't break, the problem's optimal solution must be above the jth rung and we need to find the solution to subproblem MinT(n-j, k); else if the jar breaks, the problem's optimal solution must be below the jth rung and we need to find the solution to subproblem Min(j-1,k-1). In both cases, the original problem shrinks to a smaller subproblem.

```
RECURSION-MinT(n,k):
    if(k=1) return n
    if(n=0) return 0
    ans=∞
    for j=1 to j=n:
        temp=max(RECURSION-MinT(n-j,k),RECURSION-MinT(j-1,k-1))+1
        if(temp<ans):
            ans=temp
```

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.
Drop a jar from current rung, if the jar gets broken, meaning there are MaxR(k-1,q-1) below this jung; else there are MaxR(k,q-1) rungs above this jung. MaxR(k,q)=1+MaxR(k-1,q-1)+MaxR(k,q-1). Iterate the last row of MaxR(k,q) where there are k jars, return the minimum q of which MaxR(k,q)=n as MinT(n,k).

```
RECURSION-MaxR(k,q):
    if(k=1) return 1
    if(q=0) return 0
    return 1+RECURSION-MaxR(k-1,q-1)+RECURSION-MaxR(k,q-1)
```

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.*

Choose MinT(n,k): C[n,k]=1+$min_{1\leq j\leq n}$\{max(C[j-1,k-1], C[n-j,k])\}

```
DP-MinT(n,k):
    for i=1 to i=n:
    C[i,0]=0
    C[i,1]=i
    for j=1 to j=k:
    C[0,j]=0
    for j=2 to j=k:
        for i=1 to i=n:
        tmp=∞
        for w=1 to w=i:
            if(1+max(C[w-1,k-1], C[n-w,k])<tmp):
                tmp=1+max(C[w-1,k-1], C[n-w,k])
        C[i,j]=tmp
        S[i,j]=arguments of tmp
```

D(10 points). Redo the computation this time top-down recursive, using memoization.

```
RECURSION-MinT(n,k)-MEMO:
    if(k=1) return n
    if(n=0) return 0
    if(n,k) exists in memo[][]:
        return memo[n,k]
    ans=∞
    for j=1 to j=n:
        temp=max(RECURSION-MinT-MEMO(n-j,k),RECURSION-MinT-MEMO(j-1,k-1))+1
        if(temp<ans):
            ans=temp
```

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 ..."*

Use first method to trace:

```
TRACE(n,k):
    if(S[n,k]!=∞): return
    TRACE(first element of S[n,k], second element of S[n,k])
    output(n)
```

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:

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

Solution:
Reverse the whole input string and find the LCS of the original string and the reversed string. Reverse the input string costs $\theta(n)$ time by using a left pointer and a right pointer and move them together towards the middle. Call LCS procedure from textbook and find LCS of two strings uses $\theta(n^2)$ time so total running time is $\theta(n^2)$.

```
LPS(String s):
    Sting reverse=REVERSE(s)
    return LCS(s, reverse)
REVERSE(s):
    leftPointer=0, rightPointer=s.length
    while(leftPointer<rightPointer):
        swap element at leftPointer of s with element at rightPointer of s
        leftPointer++, rightPointer- -
```

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

Solution:
Can use binary search to speed up the running time but need extra $O(n)$ space to maintain the candidate subsequence.

```
LIS-O(nlogn)
let A[] be the given n inputs
let C[] store candidate subsequence where index i denoting the last element of candidate whose length is i
C[1]=A[1]
let Prev[] store the index of which the last valid element of C gets transferred from
for i=2 to i=n
    position=BINARY-SEARCH-POSITION(C, A[i])
    C[position]=A[i]
    Prev[i]=the index of second last element of C that is corresponding to A
//roll back from the last valid element of C to adjust
length=C.length of all valid elements
prev-index=second last element of C while(length¿0):
    C[length]=prev-index
    prev-index=Prev[prev-index]
    length- -
return elements of A corresponding to C
```

```
BINARY-SEARCH-POSITION(Array, element):
left=1, right=array.length of all valid elements
while(left<right):
    middle=(left+right)/2
    if(Array[middle]<element): right=middle
    else left=middle+1
return left
```

The detailed analysis is written in the last problem of HW4. Everytime the binary search to find the proper position costs O(logn) and iterate the whole array will give a total O(nlogn) time complexity.

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:

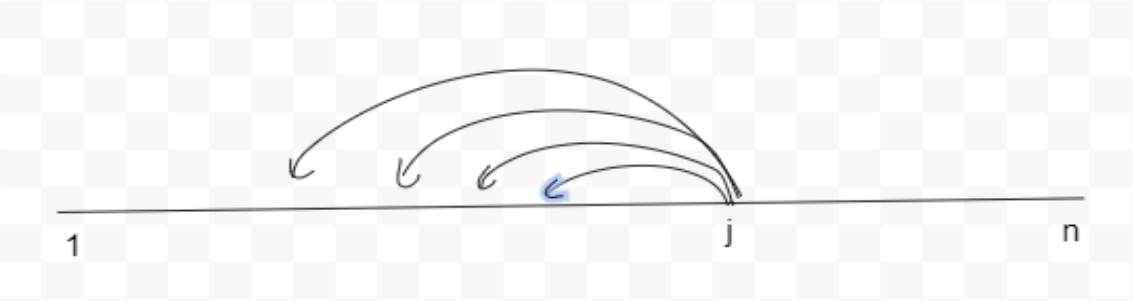
The problem has optimal subproblem structure. The optimal solution to the n words comes from the optimal solution of $n-1$ words if the last word is not considered now, once we get the optimal solution to the $n-1$ words and combining with last word's optimal solution, we get a global optimal solution.

The possible number of spaces $Space[i,j]$ at the end of each line that contains word i to word j is $M-j+i-\sum_{k=i}^{j} l_k$. $Space[i,j]$ could be negative and nonnegative.

We first define the cost from word i to word j as $Cost[i,j]=\infty$ if word i to word j cannot fit into a line, $Cost[i,j]=0$ if word $j=n$ and $Space[i,j]$ is nonnegative because last line has no cost but only valid when word i to word j can fit into a line, $Cost[i,j]=(Space[i,j])^3$ for a normal case when word i to word j can fit into a line and it is not the last line.

The dependency relation between subproblems is $C[j]=min_{1\le i\le j}\{C[i-1]+Cost[i,j]\}$. $C[j]$ represents the total cost from word 1 to word j. We need to iterate all words before j

(inclusive) to pick up the minimum cost. The bottom-up order is from left to right because $C[j]$ depends on $C[i-1]$ where $i \leq j$.
The visual of dependency is below:



Pseudocode:

```
PRINT-NEATLY(n, M):
let S[] track the trace result
C[0]=0
for j=1 to j=n:
    C[j]=∞
    for i=1 to i=j:
        if(C[i-1]+Cost[i,j]<C[j]):
            C[j]=C[i-1]+Cost[i,j]
            S[j]=i
return S[]
```

S[] tracks where the breakpoint is, for example, if n=20 and S[20]=18, meaning $word_{18}$ to $word_{20}$ should be in the same line; if S[17]=15, meaning $word_{15}$ to $word_{17}$ should be in the same line...

```
Print(S[],n):
    if(n=1) return
    Print(S[], S[n])
    output(S[n])
```

The total running time is $\theta(n^2)$ and space complexity is also $\theta(n^2)$ because Space[] and Cost[] are both 2D arrays.

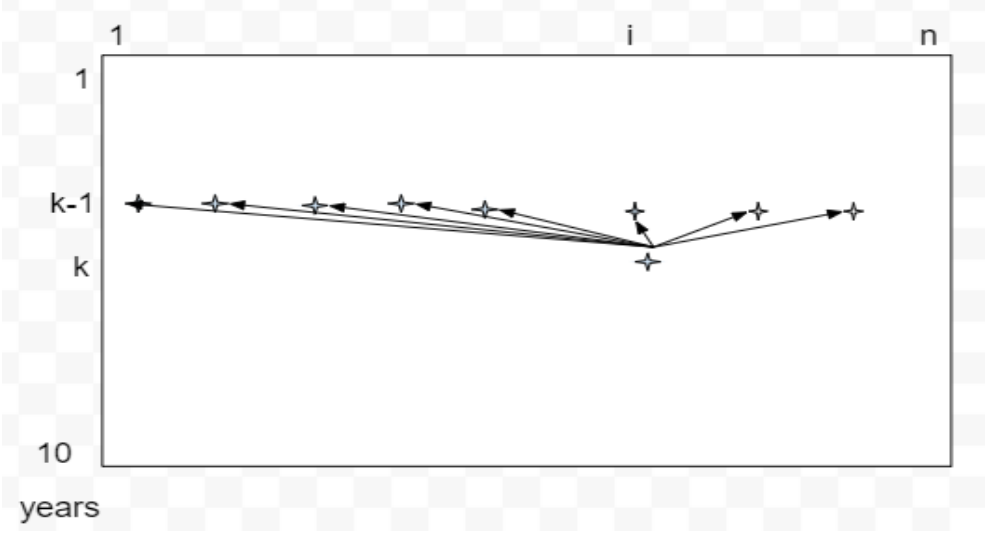
7. *(20 points)* *Problem 15-10.*

Solution:

a. We can use two investments i, j at one year k as an example to illustrate. If we invest d_1 into i at year k and invest d_2 into j at year k, the total return r_1 is $d_1 \cdot r_{ik} + d_2 \cdot r_{jk}$, if we invest d_1 and d_2 into a single investment, then the total return r_2 is $\max\{(d_1 + d_2) \cdot r_{ik}, (d_1 + d_2) \cdot r_{jk}\}$; if $r_{ik} \geq r_{jk}$, $r_2 = (d_1 + d_2) \cdot r_{ik} \geq r_1$, else $r_2 = (d_1 + d_2) \cdot r_{jk} \geq r_1$. In both cases, investing into single investment is at least as better compared to spreading money into different investments. In other words, we can always choose a best single investment each year to invest all the money into it and the problem converts to which investment should be considered best for each year.

b. Before considering 10 years, we can consider the 9th year, for a single investment at 10th year, we can iterate all investments's return at 9th year and have each of them multiply

10th year's return rate minus f_1 or f_2, at last we just choose the investment with highest return at 10th year and the investment strategy for each year will also be recorded.
c. The algorithm is dynamic programming. The optimal subproblem structure is showed in part b. The subproblem dependency is represented as $C[i,k] = max_{1\le j\le n, j\neq i}\{C[j,k-1] * r_{ik} - f_2, C[i,k-1] * r_{ik} - f_1\}$ because of the compound return. C[i,k] represents the return investment i gets at year k.
The visual is below:



Pseudocode:
let Y[] track the best investment for each year
for i=1 to i=n:
    C[i,1]=d*$r_{i1} - f_1$
for k=2 to k=10:
    for i=1 to i=n:
        C[i,k]=0
        for j=1 to j=n:
            if(j=i and $C[i,k-1] * r_{ik} - f_1 > C[i,k]$):
                $C[i,k] = C[i,k-1] * r_{ik} - f_1$
                S[i,k]=i
            else if(j!=i and $C[j,k-1] * r_{ik} - f_2 > C[i,k]$):
                $C[i,k] = C[j,k-1] * r_{ik} - f_2$
                S[i,k]=j
    Y[k]=max argument of S[i,k] whose C[i,k] is the greatest
return Y[]

Print(Y[]):
    for k=1 to k=10:
        output Y[k]
The running time is $\theta(n^2)$ since outest loop 10 is a constant. d. Because now subproblems are limited by each other and we lose the independence between subproblems. For example, if we invest $d1$ into investment i at year k and $d2$ into investment j at year k,

next year $k+1$, the optimal strategy is to transfer $d1+d2$ to single investment i, however we still need to consider how much quota left for i to get invested, meaning we cannot implement the best strategy at year $k+1$ without considering the strategy at year k. So there is no optimal substructure that exists.