ESO207 (Data Structures and Algorithms) Lecture Notes Set 8

Shashank K Mehta

1 Remarks about Divide and Conquer Technique and Memoization

Let us bring our attention to an inherent problem with divide and conquer. To explain it let us visualize the entire computation using a rooted tree, called *sub-instance Tree*. Each node of this tree represents an instance of the problem which was created during the computation. In particular, the root represents the instance of the problem that we wish to solve. If during the computation an instance $P(i)$ is reduced to $P(j_1), P(j_2), \ldots$, then in the tree the nodes of $P(j_1), P(j_2), \ldots$ are made to be the child nodes of the node of $P(i)$. So the nodes associated with the base instances are leaf nodes of the tree.

In the sub-instance tree the size of the instance associated with a node is strictly lesser than that of its parent. Hence the instance of a node is strictly smaller than the instance associated with any of its ancestors. This ensures that any instance must be different from any of its ancestors. But it is possible that the instances associated with two nodes, not ancestor-descendent, may be same. In this situation we end up solving the same instance more than once. This obviously is inefficient.

Consider the example of the computation if Fibonacci numbers. Fibonacci numbers are defined as follows:

$$f(0) = 1 f(1) = 1 f(n) = f(n-1) + f(n-2) \ \forall \ n > 1$$

The problem is to compute $f(n)$ where n is the input. This inductive definition has a natural divide and conquer based solution, see Algorithm 1.

```
Input: A non-negative integer n
Output: n-th Fibonacci number f(n)
if n ≤ 1 then
    Return 1;
else
    a := F(n − 1);
    b := F(n − 2);
    Return a + b;
end
```

Algorithm 1: Fibonacci number $F(n)$

Consider an instance $F(8)$. We notice that in the tree associated with $F(8)$, there are two nodes for $F(6)$, three nodes for $F(5)$, four nodes for $F(4)$, so on. It is clearly evident that this program is not an efficient solution. One way to overcome this wasteful repetitive computation is to save

the solution of each instance in memory when it is solved (for the first time). The algorithm may be modified in which before making a recursive call for an instance, we check the memory if that instance is already solved. If indeed its solution has already been computed, then we can read it out from the memory and not make the call. Otherwise we make a recursive call and at the end of the call we save the solution in the memory for future use if any. This method of using memory to avoid repeated computation is called *memoization*.

Is memoization always more efficient? Memoization comes with its own problems. There are two issues with it. One is the cost of memory to store the solutions of the sub-instances and the other is the time to store and retrieve the solutions of a sub-instances from the memory. We can always define a total order over the sub-instances, by mapping instance-parameters to integers. This will allow to store the solutions of sub-instances in a data structure such as balanced binary search tree. Let $I(\mathbf{k})$ denote the set of sub-instances generated in solving the instance $P(\mathbf{k})$ and $t(\mathbf{k})$ be the total number of sub-instance calls made in the algorithm, i.e., the number of nodes in the sub-instance tree of $P(\mathbf{k})$, except the root. Then the memory requirement will $O(|I|)$ and the cost of storing or searching/retrieving a solution in the data structure will be $O(f.\log\ |I|)$ where f denotes the cost of computing the mapping function. This will add $O(t.f.\log\ |I|)$ time to the time complexity for storing and retrieving the solutions. This cost may make memoization undesirable in some cases.

There may be an alternative approach to memoization in which a trade off between time and space costs could be done. If we choose a data structure such as an array, then the access cost can be reduced from $O(\log\ |I|)$ to $O(1)$. But the memory cost may increase as many slots may remain empty. To explain let us consider a hypothetical example. To simplify the discussion suppose each instance is identified by a single integer and I is $\{1, 5, 15, 16, 20, 31\}$. Such a set does not have a structure to store in an array. So we might define a superset $I' = \{5i | i = 0, 1, 2, 3, 4, 5, 6\} \cup \{5i + 1 | i = 0, 1, 2, 3, 4, 5, 6\}$ which can be stored in an array. This will allow us to access the storage in $O(1)$ time. While the time complexity may be acceptable, the space cost may increase enormously in some cases making memoization undesirable once again. Thus memoization may be a good idea to adopt in some divide and conquer algorithms but not always. In the next section we will describe the bottom up implementation of Reduce and Solve technique.

2 Dynamic Programming

In the last section we discussed the divide and conquer method which is the top down approach to reduce and solve strategy. Now we will discuss the bottom up approach for this strategy. This approach is suitable when the set of sub-instances has some properties.

2.1 Elements of Dynamic Programming

In addition to the basic requirement of reduce and solve, following conditions must be satisfied in order to apply dynamic programming.

1. The size of the set of sub-instances of $P(\mathbf{k})$, $|I(\mathbf{k})|$, should be small. Usually it must be a polynomial of its parameters.

2. The members of $I(\mathbf{k})$ should have a structure such that their solutions can be stored in a table (array of some dimensions) T such that the solution of the instance associated with $T[k_1, k_2, \dots]$ can be computed by a bounded number of solutions located an "lower" indices. It is implied that the solutions of "larger" instances are located at larger indices. In short $I(\mathbf{k})$ has a structure so looking just at the indices we know which solutions are required to solve the instance of a given index.

Now the algorithm. First, the algorithm must solve the base cases of $I(\mathbf{k})$ located at the "lowest" indices. Then iteratively compute the solutions of the non-base cases, from smaller indices to the larger indices, and stop when entire table of solution is filled. Finally output the solution of $P(bfk)$. In this approach we solve all the instances of $I(\mathbf{k})$ even if some of them may not be essential in solving $P(bfk)$.

The bottom-up approach makes better sense when there is a large gap between $|I'|$ and t, the number of recursive calls made in the top-down approach. While top-down approach solves t instances (number of nodes in the sub-instances tree), the bottom-up approach solves $|I|$ instances. For example in the Fibonacci number problem $t(n) \geq f(n+2) \approx (1.618)^{n+2}/\sqrt{5}$ while $|I(n)| = n-2$ because the sub-instances for $F(n)$ are $F(n-1), F(n-2), \ldots, F(2)$. Clearly in such situation bottom-up is the way to go.

Here are a few examples of dynamic programming.

2.2 Example 1

Let $A = a_1, a_2, \ldots, a_n$ be a sequence. Then $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ is a subsequence of A if $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. Given two sequences A and B. Then a sequence C is called a common subsequence of A and B if it is a subsequence of A as well as of B. For example, if $A = c, b, c, b, a, c, d, d, a, b, b, a, d, c, a, b$ and $B = e, a, a, c, b, f, f, c, e, c, a, b, b, a, a, e, e, f, b, a$ then sequence c, c, b, a, a is a common subsequence of A and B.

Longest Common Subsequence Problem Given two sequences $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_m$, find the longest possible common subsequence of A and B.

2.2.1 Sub-Instance Space

Clearly the lengths of the input sequences constitute the size-parameters. Let us define the sub-instances of $P(n, m)$ to be $P(i, j)$ for $0 \leq i \leq n$ and $0 \leq j \leq m$, where instance $P(i, j)$ asks for the longest common subsequence of $A_i = a_1, a_2, \ldots, a_i$ and $B_j = b_1, b_2, \ldots, b_j$. Let $S(i, j)$ denote its solution. The base cases are $P(0, j)$ and $P(i, 0)$ and the solution in each base case is the empty sequence. Next we will describe how to compute the solution of a non-basic instance, $P(i, j)$ for $i \geq 1, j \geq 1$, using the solutions of the smaller instances in I. The natural data structure to store the solutions of I is a 2-dimensional array S where we store $S(i, j)$ in $S[i, j]$. Now we will show that its has the desired structure between $S(i, j)$ and the sub-instances required to compute it.

2.2.2 Combining the Solutions of Sub-Instances

Lemma 1 *(i) If $a_i = b_j = x$, then $S(i, j) = S(i-1, j-1) \cdot x$.*
(ii) If $a_i \neq b_j$, then $S(i, j)$ is the longer of $S(i-1, j)$ and $S(i, j-1)$.

Proof L ■et $S(i, j) = y_1, y_2, \ldots, y_r$.

(i) If $y_r \neq x$, then $(y_1, \ldots, y_r)$ must also be a common subsequence of A_{i-1} and B_{i-1}. Hence $y_1, y_2, \ldots, y_r, x$ must be a common subsequence of A_i and B_j. But that is impossible since $y_1, y_2, \ldots, y_r$ is a longest common subsequence. Hence we must have $y_r = x$. Then $y_1, y_2, \ldots, y_{r-1}$ must be a common subsequence of A_{i-1} and B_{j-1}. If $y_1, y_2, \ldots, y_{r-1}$ is not the longest common subsequence of A_{i-1} and B_{j-1}, then we can compute a common subsequence of A_i and B_j by appending x to the longest common subsequence of A_{i-1} and B_{j-1}. This will give a solution of length greater than r. But that is impossible since $S(i, j)$ has length r. Hence $y_1, y_2, \ldots, y_{r-1}$ must be a longest common subsequence of A_{i-1} and B_{j-1}. Thus any solution of $P(i, j)$ is a solution of $P(i-1, j-1)$ appended by x.

(ii) In case $a_i \neq b_j$, either $y_r \neq a_i$ or $y_r \neq b_j$. So $y_1, y_2, \ldots, y_r$ must either be a common subsequence of A_{i-1} and B_j or of A_i and B_{j-1}. Thus $S(i,j)$ must be the larger of $S(i-1,j)$ and $S(i,j-1)$.

This result shows that for computing any $S(i,j)$, other than base cases, we need $S(i-1,j-1), S(i-1,j)$ and $S(i,j-1)$. Hence we first fill in the base cases $S(i,0)$ and $S(0,j)$ for all i,j. Subsequently we will fill from Row-1 to Row-n, each from the first column to the last. This way we will find the solutions $S(i-1,j-1), S(i,j-1), S(i-1,j)$ already filled when we approach to compute $S(i,j)$.

2.2.3 Algorithm

See Algorithm 2. In the design of the dynamic programming algorithm we store only the length of the solution in $S[i,j]$. To recover the actual subsequence we will also save the information about how the solution was computed in an array $D[]$. If $S[i,j] = S[i-1,j-1] \cdot A[i]$, then we save "nw" in $D[i,j]$ indicating that solution of the north-west cell (i.e., cell $i-1, j-1$) followed by $A[i]$. If $S[i,j] = S[i-1,j]$, then we save "n" and if $S[i,j] = S[i,j-1]$, then save "w" in $D[i,j]$.

2.2.4 Analysis

To fill in each entry of the solution tables only $O(1)$ time is required. Hence the time complexity of the first part is $O(n.m)$ because tables are of size $n \times m$. The second part, to recover the subsequence, has time complexity $O(\max\{n,m\})$. Hence the total time complexity is $O(n.m)$.

2.3 Example 2

The standard matrix multiplication algorithm computes (i,j)-th entry of the product by computing the dot-product (inner-product) of i-th row of the left matrix and j-th column of the right matrix. Thus computation cost of $A \cdot B$, where A is $p_1 \times p_2$ and B is $p_2 \times p_3$, is $O(p_1.p_2.p_3)$. If we have to compute $A \cdot B \cdot C$, then due to associativity property of the matrix multiplication we may compute it as $(A \cdot B) \cdot C$ or as $A \cdot (B \cdot C)$. While both orders of computation will give the same result due to associativity, their costs may be different. If sizes of A, B and C are $p_1 \times p_2, p_2 \times p_3$ and $p_3 \times p_4$ respectively, then order $(A \cdot B) \cdot C$ costs $O(p_1.p_2.p_3 + p_1.p_3.p_4)$ and order $A \cdot (B \cdot C)$ costs $O(p_2.p_3.p_4 + p_1.p_2.p_4)$.

Matrix-Chain Multiplication Problem Given a chain of matrices $M_1 \cdot M_2 \cdots M_k$ where M_i is of size $m_i \times m_{i+1}$ for all i. Determine the order of multiplication which has minimum time complexity.

2.3.1 Sub-Instance Space

The objective of this problem is to compute the optimal cost and optimal order without actually computing the product. The input for this problem is $k+1$ integers $m_1, m_2, \ldots, m_{k+1}$. The problem size is $k+1$ (we are not measuring the size in terms of bits because the basic unit of the cost will be the number of elementary integer arithmetic and logic operations.

In this problem we define I to be the set of sub-instances $P(i,r)$ with $1 \leq r \leq k$ and $1 \leq i \leq k-r+1$, which refers to the optimal cost of multiplying the chain $M_i \cdot M_{i+1} \cdot M_{i+r-1}$. Given instance is $P(1,k)$ which is in I as required. Parameter r will indicate the size of the instance. So the base cases are the chains of length 1 and 2, i.e., $P(1,1), P(2,1), \ldots, P(k,1), P(1,2), P(2,2), \ldots, P(k-1,2)$.

The optimal cost of $P(i,r)$ will be stored in a table S (here 2-D array) in location $S[i,r]$. It will be the optimal cost of computing $M_i \cdot M_{i+1} \cdot M_{i+r-1}$. We will also store the position of the

```
Input: n, A = [a_1, a_2, ..., a_n], m, B = [b_1, b_2, ..., b_m]
Output: Longest common subsequence of A and B
Initialize empty n × m arrays S[] and D[];
for k := 1 to n do
    S[k, 0] := 0;
end
for k := 1 to m do
    S[0, k] := 0;
end
for i := 1 to n do
    for j := 1 to m do
        if A[i] = B[j] then
            S[i, j] := S[i − 1, j − 1] + 1;
            D[i, j] := "NW";
        else
            if S[i − 1, j] ≥ S[i, j − 1] then
                S[i, j] := S[i − 1, j];
                D[i, j] := "N";
            else
                S[i, j] := S[i, j − 1];
                D[i, j] := "W";
            end
        end
    end
end
/* Symbols "N" and "W" indicate north and west location in table respectively.
   */
/* Construction of the solution subsequence                                  */
i := n;
j := m;
Initialize string variable C to empty string;
while i > 0 AND j > 0 do
    if D[i, j] = "NW" then
        i := i − 1;
        j := j − 1;
        C := A[i] · C;
    else
        if D[i, j] = "N" then
            i := i − 1;
        else
            j := j − 1;
        end
    end
end
Return C;
```

Algorithm 2: Longest Common Subsequence

outermost parentheses in another table D. For instance, if the the outer-most parentheses in the optimal order for $P(i,r)$ are as $(M_i \cdots M_{i+l-1})(M_{i+l} \cdots M_{i+r-1})$, then we will set $D[i,r] = l$. The information in table D will be used to recover the optimal multiplication order. In the base cases, $S[i,1] = 0$ and $S[i,2] = m_i m_{i+1} m_{i+2}$, for all i. $D[i,1]$ and $D[i,2]$ are undefined.

To compute $S[i,r]$ for $r > 2$ (non-base cases) we will consider each possibility. If we place the outermost parentheses as $(M_i \cdots M_{i+l-1})(M_{i+l} \cdots M_{i+r-1})$, then the cost will be $S[i,l] + S[i+l, r-l] + m_i m_{i+l} m_{i+r}$. Hence to determine $S[i,r]$ we will find the minimum of $S[i,l] + S[i+l, r-l] + m_i m_{i+l} m_{i+r}$ over $l = 2, 3, \ldots, r-1$. Observe that in the computation of $S[i,r]$, the solutions of the smaller instances required for the computation are all present in I. The computation order will be such that after filling in the solutions of the base cases we will compute the solutions of all chains of length $r = 3$, then that of all chains of length $r = 4$, so on.

2.3.2 Algorithm

The algorithm is described in Algorithm 3.

Input: $m_1, m_2, \ldots, m_{k+1}$
Output: A parenthesized expression of matrix chain giving a unique multiplication order
for $i := 1$ *to* k **do**
  $S[i,1] := 0$;
end
for $i := 1$ *to* $k-1$ **do**
  $S[i,2] := m_i m_{i+1} m_{i+2}$;
end
for $r := 3$ *to* k **do**
  for $i := 1$ *to* $k - r + 1$ **do**
    $S[i,r] := \infty$;
    for $l := 1$ *to* $r - 1$ **do**
      $x := S[i,l] + S[i+l, r-l] + m_i.m_{i+l}.m_{i+r}$;
      if $x < S[i,r]$ **then**
        $S[i,r] := x$;
        $D[i,r] := l$;
      end
    end
  end
end
$PrintExpression(1, k, D)$;

Algorithm 3: Computation of the optimal multiplication order for a matrix chain

2.3.3 Analysis

The time complexity of the algorithm is $O(k^3)$ due to the three nested For-loops.

2.4 Example 3

Decorative Gardening Problem n Ashok-trees were planted in a straight line, $T_1, T_2, \ldots, T_n$. Presently their respective heights are $l_1, l_2, \ldots, l_n$. It is desired that the trees be cut to heights

```
if r = 1 then
    Print M_i;
    Return;
end
if r = 2 then
    Print M_i M_{i+1} ;
    Return;
end
if D[i, r] = 1 then
    Print M_i;
    Print "(" ;
    PrintExpression(i + 1, r − 1, D);
    Print ")" ;
else
    if D[i, r] = r − 1 then
        Print "(";
        PrintExpression(i, r − 1, D);
        Print ")";
        Print M_{i+r−1};
    else
        Print "(" ;
        PrintExpression(i, l, D);
        Print")(";
        PrintExpression(i + l, r − l, D);
        Print ")";
    end
end
```

Algorithm 4: PrintExpression(i, r, D)

$l'_1, l'_2, \ldots, l'_n$ such that for any $a < b$ either $l'_a = 0$ or $l'_a \geq l'_b$. Determine $l'_1, \ldots, l'_n$ satisfying this condition such that $\sum_i l'_i$ is maximum.

Let $S = \{s_1, s_2, \ldots, s_p\}$ be the set of the initial heights of the trees, i.e., $l_i \in S$ for every i. So $p \leq n$. Without loss of generality assume that $s_1 < s_2 < \cdots < s_p$. Here is a useful fact.

Claim 1 *Let $\{l'_j | 1 \leq j \leq n\}$ be any optimal solution. Then each $l'_i \in S \cup \{0\}$.*

Proof Assume the claim is not true. Let $l'_j \in S \cup \{0\}$ for all $j < i$ and $l'_i \notin S \cup \{0\}$. Let $s_t < l'_i < s_{t+1}$. Clearly $s_{t+1} \leq l_i$. Consider the solution $l'_1, l'_2, \ldots, l'_{i-1}, s_{t+1}, l'_{i+1} \ldots, l'_n$. We have $l'_{i-1} \geq l'_i$ and $l'_{i-1} \in S$ therefore $l'_{i-1} \geq s_{t+1}$. On the other side, $s_{t+1} > l'_i \geq l'_{i+1}$. Hence this is a valid solution. The total length of the trees in this solution is greater than the total length of the trees in the optimal solution by $s_{t+1} - l'_i > 0$. This is absurd.

From this claim we know that in the solution the heights of the final trees can be restricted to the set $S' = S \cup \{0\}$. Next let us define I.

2.4.1 Sub-instance Space

In this problem we define I to be the set of problems $P(i, j)$, where $i \in [n]$ and $s_j \in S$. $P(i, j)$ is defined as follows: Given trees $T_i, T_{i+1}, \ldots, T_n$ of initial heights $l_i, l_{i+1}, \ldots, l_n$, find heights $l'_i, l'_{i+1}, \ldots, l'_n$ to which they be cut such that (i) for any indices a, b such that $i \leq a < b \leq n$, either $l'_a = 0$ or $l'_a \geq l'_b$, (ii) either $l'_a = 0$ for all $i \leq a \leq n$ or the tallest surviving tree has height s_j. Find a solution subject to these conditions such that $\sum_{j=i}^{n} l'_j$ is maximum.

Parameter $i \in [n]$ denotes the size of the instance $P(i, j)$. Larger the i smaller the instance and the base cases are $P(n, j)$ $\forall j$. In the solution table we store $S(i, j) = \sum_{j=i}^{n} l'_j$ where $l'_i, l'_{i+1}, \ldots, l'_n$ is the optimal solution for $P(i, j)$. It is easy to see that if $l_a < s_j$ for all $i \leq a \leq n$, then $S(i, j) = 0$. The original problem does not correspond to any sub-instance but its solution can be computed from the table S as follows. The solution of the given problem is $\max_{j=1}^{p}\{S(1, j)\}$.

The solutions of the base cases are given by

$$S(n, j) = \begin{cases} 0 & \text{if } l_n < s_j \\ s_j & \text{if } l_n \geq s_j \end{cases}$$

Claim 2 *For $i < n$, (i) if $l_i \geq s_k$, then $l'_i = s_k$ in the solution of $P(i, j)$ and $S(i, j) = s_k + \max_{j'=1}^{j} S(i + 1, j')$, (ii) if $l'_i < s_k$, then $l'_i = 0$ and $S(i, j) = S(i + 1, j)$.*

The non-basic instances can be computed as follows. For $i < n$ and $j \in S$

$$S(i, j) = \begin{cases} S(i + 1, j) & \text{if } l_i < s_j, \\ s_j + \max_{j'=1}^{j}\{S(i + 1, j')\} & \text{if } l_i \geq s_j \end{cases}$$

It shows that the solutions of the non-basic sub-instances can be computed from the lower sub-instances in I. Table S only stores the total height of the final trees in the optimal solution. We also use another table $Next$ to recover the height of each tree in the solution. In $Next(i, j)$ we store the height of the tallest tree in $T_{i+1}, \ldots, T(n)$ after cutting. So $Next(i, j) = j$ if $l_i < s_j$ otherwise $Next(i, j) = j''$ where $S(i + 1, j'') = \max_{j'=1}^{j}\{S(i + 1, j')\}$.

2.4.2 Algorithm

The algorithm is given in Algorithm 5.

```
Input: n, l_1, ..., l_n
Output: Optimal heights l'_1, ..., l'_n
Compute s_1, ..., s_p;
/* Assigning the basic solutions                                    */
for j = 1 to p do
    if l_n < s_j then
        S[n, j] := 0;
    else
        S[n, j] := s_j;
    end
end
/* Computing non-basic S[i, j]                                      */
for i := n − 1 to 1 do
    for j := 1 to p do
        if l_i < s_j then
            S[i, j] := S[i + 1, j];
            Next[i, j] := 0;
        else
            x := 0;
            for k := 1 to j do
                if x < S[i + 1, k] then
                    x := S[i + 1, k];
                    Next[i, j] := k;
                end
            end
            S[i, j] := s_i + x;
        end
    end
end
/* Computation of the solution                                      */
x := 0;
for k := 1 to p do
    if x < S[1, k] then
        x := S[1, k];
        k_0 := k;
    end
end
/* Generation of the sequence l'_i for i = 1, 2, ...                */
Print "Total value of the optimal solution = " S[1, k_0];
Print "The length of the trees in the optimal solution are";
k := k_0;
for i := 1 to n do
    if l_i < s_k then
        Print 0;
    else
        Print s_k;
        k := Next(i, k);
    end
end
```

Algorithm 5: Decorative Gardening

2.4.3 Analysis

The time complexity is $c.np^2$ due to three nested For-loops. Since $p \leq n$, the time complexity can be stated as $O(n^3)$.

2.5 An Insight into Sub-structure Optimality and Conditional Optimality

The central principle on which reduce and solve technique works is as follows. Given an instance $P(n)$ of an optimization problem, there exists a collection of sub-instances $P(n_1), P(n_2), \ldots$ where $n_1, n_2, \ldots$ are smaller than n such that the **optimal** solutions of these sub-instances can help compute the optimal solution of $P(n)$. Such a situation requires an "independence" of the solution of the sub-instances from the main instance $P(n)$ in the sense that the optimal solution of $P(n)$ does not depend on the details of solutions of $P(n_1), P(n_2), \ldots$. It only depends on the final values of these solutions and better the solution of these $P(n_i)$ the better the solution of $P(n)$. To make this point clear let us review some of the examples we have considered above.

In the closest pair of points in a plane we split the point set into two sets and in the first step we want to know what is the distance between the points of a closest pair in each set. We do no need to know which pair is closest in each case. Besides the solutions of the two sets are independent of each other. In the Matrix Chain Multiplication we want the best cost to multiply $M_i \cdots M_j$ and that of $M_{j+1} \cdots M_k$. Once again we never cared for how to multiply them to get the best solution. Besides the two disjoint sequences can be solved independently. In every case we do need the best solution of the sub-instances.

Now let us consider the problem of Decorative Gardening. Suppose we split the trees into the sequences $T_1, \ldots, T_i$ and $T_{i+1}, \ldots, T_n$. The question is, can we solve them independently for the best / optimal solution and then combine them? These sub-instances are not independent but we can see a conditional independence. If we fix the final height of the first non-zero tree in $T_{i+1}, \ldots, T_n$ is fixed to s_j, then we can solve for the first sequence independently. This conditional independence gives us a way to deal with the problem.

In our approach we defined a variant of the problem for the sub-instances. For the tree sequence $T_i, T_{i+1}, \ldots, T_n$ and a number s_j we defined the sub-instance $P(i, j)$ where we want the best solution where the first non-zero height is s_j. We showed that there are only a small number of options of the non-zero final heights. So we compute the solutions of $P(i, j)$ for all the options of j. With these solutions at our disposal we do not need to know the details of the final heights of $T_i, T_{i+1}, \ldots$ and we can focus on the the trees $T_1, \ldots, T_{i-1}$.

Apply these ideas to solve the following exercise.

Exercise Given a sequence of non-negative numbers $b_1, b_2, \ldots, b_n$. Find a sequence of non-negative numbers $a_1, \ldots, a_n$ so that $\sum_{i=1}^{n-1} |a_{i+1} - a_i|$ is maximum subject to the conditions $a_i \leq b_i$.