CSE 101 - ALGORITHMS - SUMMER 2000

Lecture Notes 5

Monday, July 17, 2000.

4.1.3 Longest Common Subsequence

The problem presented in this section generalizes the longest increasing subsequence problem presented in the previous section. It is defined as follows:

LONGEST-COMMON-SUBSEQUENCE

INPUT: Two arrays, $A[1, ..., n]$ and $B[1, ..., m]$.

OUTPUT: A longest common subsequence of A and B.

It indeed generalizes the longest increasing subsequence problem: we can sort A in B using any sorting algorithm; then a longest common subsequence of A and B is exactly a longest increasing subsequence of A.

Step 1) If we let $\text{LCS}(A, B, i, j)$ denote the longest common subsequence of $A[i, ..., n]$ and $B[j, ..., m]$ (the output of the problem would be $\text{LCS}(A, B, 1, 1)$), then it can be recurrently defined as follows:

$\text{LCS}(A, B, i, j)$

if $i > n$ or $j > m$ **then return** $\langle\rangle$

if $A[i] = B[j]$ **then return** $\langle A[i], \text{LCS}(A, B, i + 1, j + 1)\rangle$

return $longest(\text{LCS}(A, B, i + 1, j), \text{LCS}(A, B, i, j + 1))$

where $longest$ is a function that returns the longest sequence. Like in the previous example, this code can be slightly improved by letting $\text{LCS}(A, B, i, j)$ return the length of the longest common subsequence, while auxiliary information is stored in a global table to help us output the longest subsequence. Let $next[1, ..., n][1, ..., m]$ be a two-dimenensonal table whose elements take values in a three element set $\{next_A, next_B, next_{A,B}\}$. The intuition behind $next[i, j] = next_A$ is that in order to output the longest common subsequence of $A[i, ..., n]$ and $B[j, ..., m]$ one should look at the first element of the longest common subsequence of $A[i + 1, ..., n]$ and $B[j, ..., m]$; a similar intuition is behind $next[i, j] = next_B$; the intuition for $next[i, j] = next_{A,B}$ is that one should look for the first element in a longest subsequence of $A[i + 1, ..., n]$ and $B[j + 1, ..., m]$. We can now redefine LCS:

$\text{LCS}(A, B, i, j)$

if $i > n$ or $j > m$ **then return** 0

if $A[i] = B[j]$ **then** $\{next[i, j] \leftarrow next_{A,B}$; **return** $1 + \text{LCS}(A, B, i + 1, j + 1)$

$length_1 \leftarrow \text{LCS}(A, B, i + 1, j)$; $length_2 \leftarrow \text{LCS}(A, B, i, j + 1)$

if $length_1 > length_2$ **then** $\{next[i, j] \leftarrow next_A$; **return** $length_1\}$

$next[i, j] \leftarrow next_B$; **return** $length_2$

Then a solution algorithm for the longest common subsequence problem would be:

LONGEST-COMMON-SUBSEQUENCE(A, B)

$\text{LCS}(A, B, 1, 1)$

OUTPUT-SEQUENCE$(A, next, 1, 1)$

where OUTPUT-SEQUENCE is recurrently defined as follows:

OUTPUT-SEQUENCE$(A, next, i, j)$

if $i > n$ or $j > m$ **then return**

case $next[i, j]$ **of**

$next_{A,B}$: **output** $A[i]$; OUTPUT-SEQUENCE$(A, next, i + 1, j + 1)$

$next_A$: OUTPUT-SEQUENCE$(A, next, i + 1, j)$

$next_B$: OUTPUT-SEQUENCE$(A, next, i, j + 1)$

Exercise 4.1 Argue that the procedure $\text{LCS}(A, B, i, j)$ runs in exponential time.

Step 2) Memoizing the natural, but inefficient, algorithm above, we immediately obtain the following:

LCS(A, B, i, j)
if $l[i,j] \neq \infty$ **then return** $l[i,j]$
if $i > n$ or $j > m$ **then return** 0
if $A[i] = B[j]$ **then** $\{next[i,j] \leftarrow next_{A,B}$; **return** $l[i,j] \leftarrow 1 + \text{LCS}(A, B, i+1, j+1)$
$length_1 \leftarrow \text{LCS}(A, B, i+1, j)$; $length_2 \leftarrow \text{LCS}(A, B, i, j+1)$
if $length_1 > length_2$ **then** $\{next[i,j] \leftarrow next_A$; **return** $l[i,j] \leftarrow length_1\}$
$next[i,j] \leftarrow next_B$; **return** $l[i,j] \leftarrow length_2$

where $l[1,...,n][1,...,m]$ is a table of natural numbers initially filled with ∞. The procedures LONGEST-COMMON-SUBSEQUENCE and OUTPUT-SEQUENCE remain unchanged.

Exercise 4.2 Argue that the new procedure LCS(A, B, i, j) runs in $O(nm)$ time.

Step 3) Analyzing the natural recursive pseudocode in Step 1, we can formulate the following optimum condition in terms of l, the table introduced in Step 2:

$$l[i,j] = \begin{cases} 1 + l[i+1, j+1] & \text{when } A[i] = B[j], \\ \max\{l[i+1,j], l[i, j+1]\} & \text{otherwise.} \end{cases}$$

For a more compact presentation, suppose that $l[1,...,n+1][1,...,m+1]$ has the dimension $(n+1) \times (m+1)$ and is initially filled with 0. Then the following is a bottom-up solution for the longest common subsequence problem:

LONGEST-COMMON-SUBSEQUENCE(A, B)
for $i \leftarrow n$ **downto** 1 do
  for $j \leftarrow m$ **downto** 1 **do**
    if $A[i] = A[j]$ **then** $\{l[i,j] \leftarrow 1 + l[i+1, j+1];\ next[i,j] \leftarrow next_{A,B}\}$
    else if $l[i+1,j] > l[i, j+1]$ **then** $\{l[i,j] \leftarrow l[i+1,j];\ next[i,j] \leftarrow next_A\}$
        else $\{l[i,j] \leftarrow l[i,j+1];\ next[i,j] \leftarrow next_B\}$
OUTPUT-SEQUENCE$(A, 1, 1)$

which clearly runs in $\Theta(nm)$ time.

Exercise 4.3 The table $l[1,...,n+1][1,...,m+1]$ used in Step 3) is not necessary to be completely filled with 0s. Which elements must be 0?

4.1.4 Matrix-Chain Multiplication

The following simple pseudocode algorithm calculates the product of two matrixes:

MATRIX-PRODUCT(A, B)
if $columns(A) \neq rows(B)$ **then return** "Error"
for $i \leftarrow 1$ **to** $rows(A)$ **do**
  for $k \leftarrow 1$ **to** $columns(B)$ **do**
    $C[i,k] \leftarrow 0$
    for $j \leftarrow 1$ **to** $columns(A)$ **do**
      $C[i,k] \leftarrow C[i,k] + A[i,j] \cdot B[j,k]$
return $C[1,...,rows(A)][1,...,columns(B)]$

Multiplications are the most expensive operations in the code above. On most computer systems, a multiplication typically takes more then an order of magnitude execution time than other ordinary instructions, such as additions, jumps, etc. The number of multiplications executed by the code above is $rows(A) \times columns(A) \times columns(B)$ and we assume that it cannot be minimized[1].

But what's the number of multiplications needed to calculate the product of a chain of matrixes, $A_1 \times A_2 \times \cdots \times A_n$? Well, it depends upon *how* the matrixes are multiplied. Since the matrix multiplication operation is associative, the result of the product is the same, regardless of the order in which we do the multiplications. To be more concrete, suppose that we want to multiply 5 matrixes, A_1, A_2, A_3, A_4, A_5, of dimensions $(100, 10)$, $(10, 100)$, $(100, 10)$, $(10, 100)$, and $(100, 10)$, respectively. If we do the multiplications in the straightforward order, that is, $(((A_1 \times A_2) \times A_3) \times A_4) \times A_5$, then it can be easily seen that the total number of multiplications is $400,000$. On the other hand, if the matrixes are first grouped like $A_1 \times ((A_2 \times A_3) \times (A_4 \times A_5))$, then the total number of multiplications executed to calculate their product is $31,000$, which is quite an improvment! Since matrix multiplications are very common operations in many areas of computing and mathematics, such as graphics, neural networks, numerical anlysis, etc., efficient algorithms to do these operations are of great practical interest.

A *chain of matrixes* is a sequence $\mathcal{C} = \langle A_1, A_2, ..., A_n \rangle$ of matrixes whose dimensions are compatible, that is, $columns(A_i) = rows(A_{i+1})$ for each $1 \leq i \leq n-1$. A *vector of dimensions* $D = \langle d_0, d_1, d_2, ..., d_n \rangle$ is associated to each chain of matrixes, having the property that $d_{i-1} = rows(A_i)$ and $d_i = columns(A_i)$ for each matrix A_i in the chain. Then the matrix-chain multiplication can be formulated as follows:

MATRIX-CHAIN-MULTIPLICATION

INPUT: A chain of matrixes $\mathcal{C} = \langle A_1, A_2, ..., A_n \rangle$.

OUPTUT: The product $A_1 \times A_2 \times \cdots A_n$, using a minimum number of multiplications.

This is a typical problem of optimum which admits a nice straightforward dynamic programming algorithm.

Step 1) If we let $\textsc{Mult}(D, i, j)$ denote the minimum number of multiplications needed to calculate the product $A_i \times A_{i+1} \times \cdots \times A_j$ (this minimum does not depend on the elements of the matrixes, but only on the dimensions of the matrixes), then we get the following optimum condition:

$$\textsc{Mult}(D, i, j) = \begin{cases} 0, \text{when } i = j, \text{ and} \\ \min_{i \leq k < j}\{\textsc{Mult}(D, i, k) + \textsc{Mult}(D, k+1, j) + D[i-1] \cdot D[k] \cdot D[j]\}, \text{otherwise.} \end{cases}$$

Therefore, in order to calculate the product $A_i \times A_{i+1} \times \cdots \times A_j$ we first choose an index k between i and j where we split the problem in two subproblems of smaller size, then calculate the subproducts $A_i \times A_{i+1} \times \cdots \times A_k$ and $A_k \times A_{k+1} \times \cdots \times A_j$ obtaining two matrixes of dimensions $(D[i-1], D[k])$ and $(D[k], D[j])$, and multiply those two matrixes using $D[i-1] \cdot D[k] \cdot D[j]$ multiplications. The optimality principle suggests that we choose that k which minimizes the total number of multiplications.

A natural recursive algorithm implementing the optimum condition above can be easily written now. Notice that, in order to calculate the product of chain we need to store the local decisions, that is the indexes k; thus we need a table, say $split?[1, ..., n][1, ..., n]$, and each time we find a k as in the optimum condition above we store it in $split?[i, j]$. We now go directly to the next step.

Step 2) Let $m[1, ..., n][1, ..., n]$ be a table which initially contains only the special element ∞. Then $\textsc{Mult}(D, i, j)$ can be simply implemented as follows:

$\textsc{Mult}(D, i, j)$

if $m[i, j] \neq \infty$ **then return** $m[i, j]$

if $i = j$ **then return** $m[i, j] \leftarrow 0$

for $k \leftarrow i$ **to** $j - 1$ **do**

  if $m[i, j] > (temp \leftarrow \textsc{Mult}(D, i, k) + \textsc{Mult}(D, k+1, j) + D[i-1] \cdot D[k] \cdot D[j])$

  then $\{m[i, j] \leftarrow temp;\ split?[i, j] \leftarrow k\}$

return $m[i, j]$

[1] In fact, it can be minimized but this is a complex research subject which we do not approach in this course.

The optimal product of all the matrixes in a chain can be now easily calculated:

MATRIX-CHAIN-MULTIPLICATION($\mathcal{C}$)

Let D be the vector of dimensions of $\mathcal{C}$; it can be part of input

MULT($D, 1, length(\mathcal{C})$)

return CALCULATE-PRODUCT($\mathcal{C}, 1, length(\mathcal{C}), split?$)

where CALCULATE-PRODUCT can be recursively defined as follows:

CALCULATE-PRODUCT($\mathcal{C}, i, j, split?$)

if $i = j$ **then return** $\mathcal{C}[i]$

$k \leftarrow split?[i, j]$

$A \leftarrow$ CALCULATE-PRODUCT($\mathcal{C}, i, k, split?$)

$B \leftarrow$ CALCULATE-PRODUCT($\mathcal{C}, k + 1, j, split?$)

return MATRIX-PRODUCT(A, B)

Exercise 4.4 Argue that the running time of MULT($D, 1, n$) is $O(n^3)$.

Step 3) Examining the dependences among indexes i, j, k in the optimum condition and also the order in which the table $m[1, ..., n][1, ..., n]$ is filled by MULT($D, 1, n$), we deduce that in order to do it iteratively, we need to vary the index i from n downto 1, the index j from $i + 1$ to n, and the index k from i to $j - 1$. Suppose that the table m is filled with 0s. Then we obtain the following iterative version of the algorithm presented in Step 2):

MATRIX-CHAIN-MULTIPLICATION($\mathcal{C}$)

1 Let D be the vector of dimensions of $\mathcal{C}$; it can be part of input

2 **for** $i \leftarrow n$ **downto** 1 **do**

3   **for** $j \leftarrow i + 1$ **to** n **do**

4   **for** $k \leftarrow i$ **to** $j - 1$ **do**

5   **if** $m[i, j] > (temp \leftarrow m[i, k] + m[k + 1, j] + D[i - 1] \cdot D[k] \cdot D[j])$

           then $\{m[i, j] \leftarrow temp;\ split?[i, j] \leftarrow k\}$

6 **return** CALCULATE-PRODUCT($\mathcal{C}, 1, length(\mathcal{C}), split?$)

where CALCULATE-PRODUCT is the function defined in Step 2).

Exercise 4.5 We assumed that the table m was initially filled with 0s in the algorithm above. Why? Is it necessary to be completely filled with 0s?

Exercise 4.6 Show that the running time of the steps 1-5 in the algorithm MATRIX-CHAIN-MULTIPLICATION above is $O(n^3)$.

4.1.5 The Partition Problem

See Subsection 3.1.2 in the textbook.

4.1.6 Maximum Sum in Any Contiguous Sublist

Given a list of n real numbers, consider the problem of finding the maximum sum in any contiguous sublist of the input. We'll illustrate a series of algorithms for this problem, including an efficient dynamic programming one.

Needs serious revision! Take the following solutions as hints only.

Solution 1 (divide-and-conquer) The first algorithm uses a divide-and-conquer approach and runs in time $O(n \log n)$. Informally, it works as follows: Divide the given list into two sublists (call them "left" and "right") of roughly 1/2 the size of the original list, and recursively "solve" each sublist by finding the maximum "contiguous sum" in both. How do we then solve our original list? Take the largest of:

(1) the maximum "contiguous sum" in the "left" sublist

(2) the maximum "contiguous sum" in the "right" sublist

(3) the maximum "contiguous sum" that "spreads" from the point at which we split the list into two sublists (this sum may or may not include values from both sublists). To compute this, we will need to compute:

(3a) the maximum "contiguous sum" whose corresponding values make up the last l positions of the "left" sublist (for some l)

(3b) the maximum "contiguous sum" whose corresponding values make up the first r positions of the "right" sublist (for some r)

We then want the maximum of (3a), (3b), and (3a)+(3b) as our answer to (3).
Observe that we could just compare (3a)+(3b) with (1) and (2), because if (3a) is the maximum "contiguous sum" in the big list, then (3a) must equal (1); the same goes for (3b) and (2).

Here is the algorithm:

Inputs: A, an array of real values; $Left$ and $Right$, the leftmost and rightmost position numbers of A in the subarray under consideration.
Outputs: the maximum "contiguous sum" in A.

procedure MAXSUM (A , $Left, Right$);
var
    $Center$: integer;
        # The position (in A) of the last element in the left subarray.
    $LeftSum, RightSum$: real;
        # The answers to (1) and (2)
    $LeftBorderSum, RightBorderSum$: real;
        # The answers to (3a) and (3b)
begin
  if $Left = Right$ **then**
    return $A[Left]$
      # The base case is $n = 1$ (i.e., when $Left = Right$).
      # We return the sole array entry
  $Center := \lfloor (Left + Right)/2 \rfloor$
  $LeftSum$:= MAXSUM($A, Left, Center$)
  $RightSum$:= MAXSUM($A, Center + 1, Right$)
  compute $LeftBorderSum$
  compute $RightBorderSum$
      # Code omitted. Both these computations are $O(n)$.
  return max($LeftSum, RightSum, LeftBorderSum + RightBorderSum$)
      # We want the maximum of (1), (2), and (3a)+(3b).
end;

The recurrence is $T(n) = 2T(n/2) + O(n)$. We split arrays into two smaller arrays, each of size $\approx$ 1/2 the size of the big array. The nonrecursive portion of the code is $O(n)$; in fact, it is $\Theta(n)$. Its solution is, by Case 2 of the Master Theorem (with $a = 2$ and $b = 2$ and thus $n^{\log_b a} = n^{\log_2 2} = n^1 = n$), $T(n) \,\epsilon\, \Theta(n \lg n)$.

Note: You could also look at the corresponding recursion tree, which will have depth $\approx \log_2 n$, or $\lg n$. At each level in the recursion tree, $O(n)$ work is being done. So, the algorithm is $O(n \lg n)$.

Solution 2 (divide-and-conquer) There is a way to reduce the $O(n)$ "overhead" in the previous solution to $O(1)$. Instead of having the algorithm compute the "border sums" in $\Theta(n)$ time in each call, let's have the algorithm "paste together" information provided by results from the recursive calls. Consider the following algorithm:

```
procedure MAXSUM2 (A , Left, Right, var LeftEndSum, var RightEndSum,
var AllSum)
var
        Center : integer;
        LeftSum, RightSum : real;
        LeftEndSum, RightEndSum : real;
                # The maximum "contiguous sums" starting from the left end
                # (respectively, ending at the right end) of the array
        LeftBorderSum, RightBorderSum : real;
                # The answers to (3a) and (3b)
        AllLeftSum, AllRightSum : real;
                # The total sum of all elements in the left (respectively right) subarrays.
        AllSum : real;
                # This is the sum of all the elements in the array.
begin
    if Left = Right then
        begin
            LeftEndSum := A[Left]
            RightEndSum := A[Left]
            AllSum := A[Left]
            return A[Left]
        end
    Center := ⌊(Left + Right)/2⌋
    LeftSum := MAXSUM2(A, Left, Center, LeftEndSum, LeftBorderSum, AllLeftSum)
    RightSum := MAXSUM2(A, Center + 1, Right, RightBorderSum, RightEndSum, AllRightSum)
    AllSum := AllLeftSum + AllRightSum
    LeftEndSum := max(LeftEndSum, AllLeftSum + RightBorderSum)
    RightEndSum := max(RightEndSum, LeftBorderSum + AllRightSum)
    return max(LeftSum, RightSum, LeftBorderSum + RightBorderSum)
end;
```

The recurrence is $T(n) = 2T(n/2) + O(1)$, since we split arrays into two smaller arrays, each roughly half the size of the larger array. The nonrecursive portion of the code is $O(1)$ (i.e., the complexity is bounded by a constant). Its solution is, by Case 1 of the Master Theorem (with $a = 2$, $b = 2$, $\log_b a = \log_2 2 = 1$ and $\epsilon = \log_2 2 = 1$, say), $T(n) \,\epsilon\, \Theta(n)$.

Note: You could also look at the corresponding recursion tree, which will have depth $\approx \log_2 n$. For simplicity, let's say that the depth of all leaves in the recursion tree is $\log_2 n$. Then, when we add up the number of subproblems at each recursive level, we get:

$$2^0 + 2^1 + ... + 2^{\log_2 n} = \frac{1 - 2^{(\log_2 n)+1}}{1-2} = 2^{(\log_2 n)+1} - 1 = 2n - 1$$

(Think: $\frac{\text{“first in” - “first out”}}{1 - \text{common base}}$.)

Remember that $\Theta(1)$ work is being done at each node in the recursion tree.

Solution 3 (dynamic programming) Let's say the array (or list) "A" is indexed from 1 to n. Set up an array L indexed from 0 to n. Let $L[i]$ be the "maximum contiguous sum" whose corresponding subarray ends with $A[i]$, the element of the A array in the ith position ($1 \leq i \leq n$). Let $L[0] = 0$, since 0 is a viable answer if all of the $A[i]$ values are nonpositive or if the A array has no elements (a weird case). Our "maximum contiguous sum" in A is thus $\max_{0 \leq i \leq n} L[i]$; this is because the true "maximum contiguous sum" is $L[0] = 0$, or it corresponds to a subarray that terminates at some position i in the array ($1 \leq i \leq n$). (There may, incidentally, be more than one subarray that corresponds to the sum.)

Here's the algorithm:

Inputs: n, array A indexed from 1 to n.
Outputs: $MaxSum$, the "maximum contiguous sum" in A.

```
procedure MAX.CONTIGUOUS.SUM (n, A);
var
    L : array [0..n];
begin
    L[0] := 0
    for i = 1 to n do
        if L[i − 1] < 0 then
            L[i] := A[i]
                # If the "maximum contiguous sum" ending at array position i − 1
                # is negative (let's call the corresponding subarray B), then we would
                # rather take A[i] as our "maximum contiguous sum" ending at position i
                # instead of attaching A[i] to B, since B just "drags A[i] down."
        else
            L[i] := L[i − 1] + A[i]
                # If the "maximum contiguous sum" ending at array position i − 1
                # is nonnegative, then we want to attach A[i] to the corresponding
                # subarray and take the sum.
        end
    end
    MaxSum:=max(L)
        # Here, max takes the maximum value in L. This operation is Θ(n).
    return MaxSum
end;
```

This iterative algorithm is $\Theta(n)$, since the **for** loop and the **max** function are both $\Theta(n)$.

Solution 4 (refinement of the above)

Set up a variable $MaxSum$ that will keep track of $\max_{0 \leq k \leq i} L[k]$ as i increases from 0 to n. Then, the second pass through the L array to find the maximum value becomes unnecessary.

In fact, we don't even need the L array! We can replace it with another variable, $CurrentSum$. We don't need to maintain the L array because (1) $MaxSum$ is already keeping track of the running maximum of the $L[i]$ quantities, and (2) $CurrentSum$ contains all the information from previous array elements that is needed to update $MaxSum$. In particular, $CurrentSum$ keeps track of max(0,$L[i]$) as i increases from 0 to n.

Here's the algorithm:

Inputs: n, array A indexed from 1 to n.
Outputs: $MaxSum$, the "maximum contiguous sum" in A.

```
procedure MAX.CONTIGUOUS.SUM2 (n, A);
var
```

$CurrentSum, MaxSum$: real;
begin
$CurrentSum := 0$
$MaxSum := 0$
for $i = 1$ **to** n **do**
$CurrentSum := CurrentSum + A[i]$
# This is the sum we would get if we were to attach $A[i]$ to the
# "maximum contiguous subarray" B ending at position $i - 1$
# *if the sum of B's elements is nonnegative*. The initialization
# of $CurrentSum$ and the upcoming **if** statement guarantee that,
# if the B sum is negative, it will not be added to $A[i]$, and we
# will take $A[i]$ as our $CurrentSum$. In short, $CurrentSum$ is
# assigned $\max(0, L[i-1]) + A[i]$, which was how we computed
# successive values of $L[i]$ in the first algorithm.
if $CurrentSum < 0$ **then**
$CurrentSum := 0$
This guarantees that [in the next iteration], array position $i + 1$ will start
off a new subarray under consideration. Since the "maximum contiguous
sum" ending at position i is negative, we don't want to add it in next
time.
else if $CurrentSum > MaxSum$ **then**
$MaxSum := CurrentSum$
Update $MaxSum$ as necessary.
end
end
return $MaxSum$
end;

This iterative algorithm is $\Theta(n)$, since the **for** loop is $\Theta(n)$.