Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Subsection 1

Rod Cutting

The Rod-Cutting Problem

- Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods.

  Long steel rods are bought and cut into shorter rods, which are then sold. Each cut is free. The management wants to know the most profitable way to cut up the rods.
- We work under the following hypotheses:
  - We know, for $i = 1, 2, \ldots$, the price p_i in dollars that is charged for a rod of length i inches;
  - Rod lengths are always an integral number of inches.
- The **rod-cutting problem** is the following:

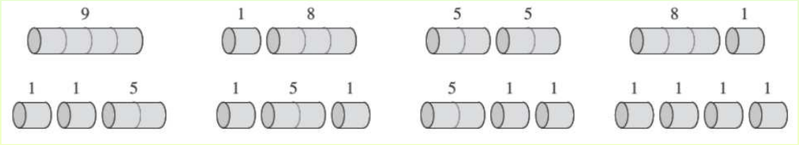
  Given a rod of length n inches and a table of prices p_i, $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
- Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Illustration for $n = 4$

- Consider the case when $n = 4$. Assume pricing, given by

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

The ways to cut up a rod of 4 inches in length, including the way with no cuts at all is shown here:



We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

- We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end.

Considering Smaller Problems

- We denote a decomposition in sum notation.
  E.g., $7 = 2 + 2 + 3$ means that a rod of length 7 is cut into three pieces with length 2, 2 and 3.
- If an optimal solution cuts a rod of length n into k pieces, then an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$ provides maximum corresponding revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$.
- We can frame the values r_n, for $n \geq 1$, in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1).$$

- Since we do nor know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue.
  We also have the option of picking no i at all if we can obtain more revenue by selling the rod uncut.

Optimal Substructure

- Thus, to solve the original problem of size n, we solve smaller problems of the same type, but of smaller sizes.

  Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem.
- The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.
- We say that the rod-cutting problem exhibits **optimal substructure**:

  Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

An Alternative Formulation

- In a slightly simpler way to arrange a recursive structure for the rod cutting problem, we view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n-i$.

  Only the remainder, and not the first piece, may be further divided.
- The solution with no cuts at all has first piece of size $i=n$ and revenue p_n and the remainder size 0 with corresponding revenue $r_0=0$.
- Overall, we obtain the simpler version

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}).$$

- In this formulation, an optimal solution embodies the solution to only one related subproblem rather than two.

Recursive Top-Down Implementation

- The following procedure implements the computation in a straightforward, top-down, recursive manner.

$\textsc{CutRod}(p, n)$

```
1. if $n == 0$
2.     return 0
3. $q = -\infty$
4. for $i = 1$ to $n$
5.     $q = \max(q, p[i] + \textsc{CutRod}(p, n - i))$
6. return $q$
```

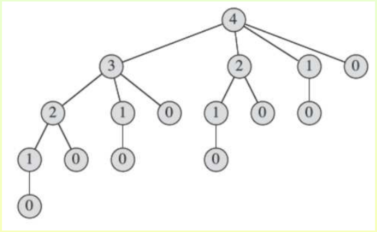
- Procedure $\textsc{CutRod}$ takes as input an array $p[1 \dots n]$ of prices and an integer n, and returns the max revenue possible for a rod of length n.
  - If $n = 0$, no revenue is possible, and so $\textsc{CutRod}$ returns 0 in Line 2.
  - Line 3 initializes the maximum revenue q to $-\infty$, so that the for loop in Lines 4-5 correctly computes $q = \max_{1 \leq i \leq n}(p_i + \textsc{CutRod}(p, n - i))$.

Running Time Explosion

- Once the input size becomes moderately large, CutRod would take a long time to run: For $n = 40$, the program takes at least several minutes, and most likely more than an hour. Each time n is increased by 1, the program's running time would approximately double.

  CutRod is inefficient because it calls itself recursively over and over again with the same parameter values.

  In fact, it solves the same subproblems repeatedly. E.g. for $n = 4$: CutRod(p, n) calls CutRod$(p, n - i)$ for $i = 1, 2, \ldots, n$. Equivalently, CutRod(p, n) calls CutRod(p, j), for each $j = 0, 1, \ldots, n - 1$.



  When this process unfolds recursively, the amount of work done, as a function of n, grows explosively.

Running Time of CutRod

- Let $T(n)$ denote the total number of calls made to CutRod when called with its second parameter equal to n.

  This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree.
- Since the count includes the initial call at its root, $T(0) = 1$ and $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$.

  The solution is $T(n) = 2^n$, whence the running time of CutRod is exponential in n.
- This should not be surprising since CutRod considers all 2^{n-1} possible ways of cutting up a rod of length n.
- The dynamic programming method is used to convert CutRod into an efficient algorithm.

Dynamic Programming: Time vs. Space Tradeoff

- Instead of computing the same subproblems repeatedly, each subproblem is solved only once and its solution is saved.
- If the solution is needed again later, it is looked up rather than being recomputed.
- Dynamic programming thus uses additional memory to save computation time, i.e., is an example of a **time-memory trade-off**.
- An exponential-time solution may be transformed in this way into a polynomial-time solution.
- A dynamic-programming approach runs in polynomial time when:
  - the number of distinct subproblems involved is polynomial in the input size; and
  - we can solve each such subproblem in polynomial time.

Top-Down With Memoization

- There are two equivalent ways to implement a dynamic programming approach.
    - The first approach is **top-down with memoization**.
      In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
      The procedure now first checks to see whether it has previously solved this subproblem.
        - If so, it returns the saved value, saving further computation at this level.
        - If not, the procedure computes the value in the usual manner.

      We say that the recursive procedure has been **memoized**, i.e., it "remembers" what results it has computed previously.

Bottom-Up Method

- There are two equivalent ways to implement a dynamic programming approach.
  - The second approach is the **bottom-up method**.
    This approach typically depends on some natural notion of the “size” of a subproblem, such that solving any particular subproblem depends only on solving “smaller” subproblems.
    We sort the subproblems by size and solve them in that order.
    When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.

The Top-Down Memoization Procedure

$\textsc{MemoizedCutRod}(p, n)$

```
1. let $r[0 \dots n]$ be a new array
2. for $i = 0$ to $n$
3.     $r[i] = -\infty$
4. return $\textsc{MemoizedCutRodAux}(p, n, r)$
```

$\textsc{MemoizedCutRodAux}(p, n, r)$

```
1. if $r[n] \geq 0$
2.     return $r[n]$
3. if $n == 0$
4.     $q = 0$
5. else $q = -\infty$
6.     for $i = 1$ to $n$
7.         $q = \max(q, p[i] + \textsc{MemoizedCutRodAux}(p, n - i, r))$
8. $r[n] = q$
9. return $q$
```

How MemoizedCutRod Works

- The main procedure MemoizedCutRod initializes a new auxiliary array $r[0 \ldots n]$ with the value $-\infty$, a convenient choice with which to denote "unknown".
- It then calls its helper routine, MemoizedCutRodAux.

  The procedure MemoizedCutRodAux is just the memoized version of our previous procedure, CutRod.
  - It first checks in Line 1 to see whether the desired value is already known.
    - If it is, then Line 2 returns it.
    - Otherwise, Lines 3-7 compute the desired value q in the usual manner.
  - Line 8 saves it in $r[n]$;
  - Line 9 returns it.

The Bottom Up Procedure

- The bottom-up version is even simpler.

BOTTOMUPCUTROD(p, n)

```
let $r[0 \dots n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j-i])$
    $r[j] = q$
return $r[n]$
```

- For the bottom-up approach, BOTTOMUPCUTROD uses the natural ordering of the subproblems: A problem of size i is "smaller" than a subproblem of size j if $i < j$.

  The procedure solves subproblems of sizes $j = 0, 1, \dots, n$, in order.

How the Bottom-Up Procedure Works

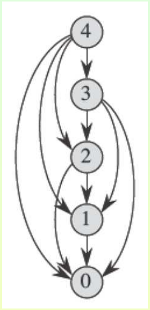
- Line 1 of procedure BottomUpCutRod creates a new array $r[0 \ldots n]$ in which to save the results of the subproblems.
- Line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue.
- Lines 3-6 solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size.

  The approach used to solve a problem of a particular size j is the same as that used by CutRod, except that Line 6 now references array entry $r[j - i]$ instead of making a recursive call.
- Line 7 saves the solution in $r[j]$.
- Line 8 returns $r[n]$, the optimal value for r_n.
- The bottom-up and top-down procedures have same asymptotic running time $\Theta(n^2)$.

Subproblem Graphs

- The **subproblem graph** of a problem shows the set of subproblems and how they depend on one another.

  The subproblem graph has a directed edge from the vertex for subproblem x to the vertex for subproblem y if determining an optimal solution for subproblem x involves directly considering an optimal solution for subproblem y.



- We can think of the subproblem graph as a "reduced" or "collapsed" version of the recursion tree for the top-down recursive method.

  In this version we coalesce all nodes for the same subproblem into a single vertex and direct all edges from parent to child.

Reconstructing a Solution

- To return an actual solution, we can extend the dynamic programming approach to record a choice that led to the optimal value:

ExtendedBottomUpCutRod(p, n)

```
1.  let $r[0 \dots n]$ and $s[0 \dots n]$ be new arrays
2.  $r[0] = 0$
3.  for $j = 1$ to $n$
4.      $q = -\infty$
5.      for $i = 1$ to $j$
6.          if $q < p[i] + r[j-i]$
7.              $q = p[i] + r[j-i]$
8.              $s[j] = i$
9.      $r[j] = q$
10. return $r$ and $s$
```

- This imitates BottomUpCutRod, except that it creates the array s in Line 1, and it updates $s[j]$ in Line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

Printing the List of Sizes in an Optimal Cut

- The following procedure takes a price table p and a rod size n.
  - It calls EXTENDEDBOTTOMUPCUTROD to compute the array $s[1 \ldots n]$ of optimal first-piece sizes;
  - It then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n.

PRINTCUTRODSOLUTION(p, n)

```
1. $(r, s) = $ EXTENDEDBOTTOMUPCUTROD$(p, n)$
2. while $n > 0$
3.     print $s[n]$
4.     $n = n - s[n]$
```

Illustration of Printing

- In our rod-cutting example, EXTENDEDBOTTOMUPCUTROD$(p, 10)$ would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- A call to PRINTCUTRODSOLUTION$(p, 10)$ would print just 10.
- A call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7.

Subsection 2

Matrix-Chain Multiplication

Matrix-Chain Multiplication

- We are given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product $A_1 A_2 \cdots A_n$.
- We can evaluate the expression as follows:
    - We parenthesize it to resolve all ambiguities concerning the order of multiplication;
    - We then use the standard algorithm for multiplying pairs of matrices as a subroutine.
- Matrix multiplication is associative, and so all parenthesizations yield the same product.

Full Parenthesization

- A product of matrices is **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

  Example: If the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

$$\begin{array}{c}(A_1(A_2(A_3A_4))),\\ (A_1((A_2A_3)A_4)),\\ ((A_1A_2)(A_3A_4)),\\ ((A_1(A_2A_3))A_4),\\ (((A_1A_2)A_3)A_4).\end{array}$$

Matrix Multiplication

- How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.
- Consider first the cost of multiplying two matrices.

  The standard algorithm is given by the following pseudocode:

MatrixMultiply(A, B)

```
1. if $A$.columns $\neq$ $B$.rows
2.    error "incompatible dimensions"
3. else let $C$ be a new $A$.rows $\times$ $B$.columns matrix
4.    for $i = 1$ to $A$.rows
5.       for $j = 1$ to $B$.columns
6.          $c_{ij} = 0$
7.          for $k = 1$ to $A$.columns
8.             $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
9. return $C$
```

Parenthesization and Efficiency

- If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in Line 8, which is pqr.

  Example: To compare costs due to different parenthesizations, consider $\langle A_1, A_2, A_3 \rangle$, with dimensions 10×100, 100×5 and 5×50.
  - If we multiply according to the parenthesization $((A_1 A_2) A_3)$, the total cost in scalar multiplications is:
  $$(10 \cdot 100 \cdot 5) + (10 \cdot 5 \cdot 50) = 7,500;$$
  - If we multiply according to the parenthesization $(A_1 (A_2 A_3))$, the total cost in scalar multiplications is:
  $$(100 \cdot 5 \cdot 50) + (10 \cdot 100 \cdot 50) = 75,000.$$

  Thus, computing the product according to the first parenthesization is 10 times faster.

The Matrix-Chain Multiplication Problem

- **The Matrix-Chain Multiplication Problem**:

  Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.
- Note that in the matrix-chain multiplication problem:
  - We are not actually multiplying matrices.
  - The goal is only to determine an order for multiplying matrices that has the lowest cost.
- Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications.

Counting the Number of Parenthesizations

- Exhaustively checking all possible parenthesizations is not efficient.
- Denote the number of alternative parenthesizations by $P(n)$.
  - When $n = 1$, $P(1) = 1$;
  - When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts.
    The split between the two subproducts may occur between the kth and $(k+1)$st matrices, for any $k = 1, 2, \ldots, n-1$.

  Thus, we obtain

$$P(n) = \begin{cases} 1, & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{if } n \geq 2 \end{cases}.$$

- The solution to this recurrence is $\Omega(2^n)$, whence the number of possible solutions is exponential in n.
- Exhaustive search is therefore a poor strategy.

Applying Dynamic Programming to Parenthesize

- To use the dynamic-programming method to determine how to optimally parenthesize a matrix chain, we shall follow a four-step sequence:
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.
- We go through these steps in order, demonstrating clearly how we apply each step to the problem.

Step 1: Structure of an Optimal Parenthesization

- We find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.
- We adopt the notation $A_{i\dots j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$.
- Observe that, if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1}, for some integer k in the range $i \leq k < j$. That is, for some k, we first compute the matrices $A_{i\dots k}$ and $A_{k+1\dots j}$ and then multiply them together to produce the final product $A_{i\dots j}$.
- The cost of parenthesizing this way consists of:
  - The cost of computing the matrix $A_{i\dots k}$;
  - The cost of computing $A_{k+1\dots j}$;
  - The cost of multiplying them together.

Step 1 (Cont'd)

- The optimal substructure of this problem is as follows.

  Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}.

  - Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$.

    If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ with cost lower than the optimum, a contradiction.

  - A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$. It must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Summary of Substructure Analysis

- Optimal substructure shows that we can construct an optimal solution to the problem from optimal solutions to subproblems.
- We have seen that:
  - Any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product;
  - Any optimal solution contains within it optimal solutions to subproblem instances.
- Thus, we can build an optimal solution to an instance by:
  - Splitting the problem into two subproblems (optimally parenthesizing $A_iA_{i+1}\cdots A_k$ and $A_{k+1}A_{k+2}\cdots A_j$);
  - Finding optimal solutions to subproblem instances;
  - Combining these optimal subproblem solutions.
- When searching for the correct place to split the product, we must consider all possible places so as to pick the optimal one.

Step 2: A Recursive Solution

- We pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$, for $1 \leq i \leq j \leq n$.
- Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i\ldots j}$. For the full problem, the lowest cost way to compute $A_{1\ldots n}$ would thus be $m[1,n]$.
- We can define $m[i,j]$ recursively as follows.
    - If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i\ldots i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i,i] = 0$, for $i = 1, 2. \ldots, n$.
    - To compute $m[i,j]$ when $i < j$, we assume that to optimally parenthesize, we split $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \leq k < j$. Then, $m[i,j]$ equals the minimum cost for computing the subproducts $A_{i\ldots k}$ and $A_{k+1\ldots j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we get $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$.

The Recursive Equation

- The equation $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$ assumes that we know the value of k, which we do not.
- The possible values for k, are $k = i, i+1, \ldots, j-1$.
- Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best.
- Thus, our recursive definition for the minimum cost of parenthesizing the product $A_iA_{i+1}\cdots A_j$ becomes

$$m[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, & \text{if } i < j \end{cases}$$

- We define $s[i,j]$ to be a value of k at which we split $A_iA_{i+1}\cdots A_j$ in an optimal parenthesization. That is, $s[i,j] = k$, such that

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

Step 3: Computing the Optimal Costs

- We implement a bottom-up method in MATRIXCHAINORDER.

  It assumes that matrix A_i has dimensions $p_{i-1} \times p_i$, $i = 1, 2, \ldots, n$.
- Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.\text{length} = n + 1$.
- It uses an auxiliary table $m[1 \ldots n, 1 \ldots n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1 \ldots n-1, 2 \ldots n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$.
- In bottom-up, the cost $m[i, j]$ of computing a matrix-chain product of $j - i + 1$ matrices depends only on the costs of computing products of fewer than $j - i + 1$ matrices. For $k = i, i + 1, \ldots, j - 1$,
  - the matrix $A_{i \ldots k}$ is a product of $k - i + 1 < j - i + 1$ matrices;
  - the matrix $A_{k+1 \ldots j}$ is a product of $j - k < j - i + 1$ matrices.
- The algorithm fills in the table m in a manner corresponding to solving the parenthesization problem on matrix chains of increasing length.

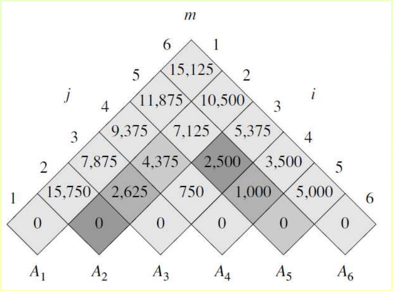
The Procedure MatrixChainOrder

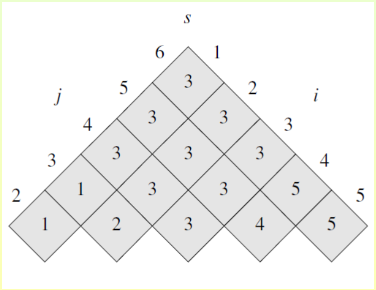
MatrixChainOrder(p)

```
$n = p$.length $- 1$
let $m[1 \ldots n, 1 \ldots n]$ and $s[1 \ldots n-1, 2 \ldots n]$ be new tables
for $i = 1$ to $n$
    $m[i,i] = 0$
for $\ell = 2$ to $n$ $//\ell$ is the chain length
    for $i = 1$ to $n - \ell + 1$
        $j = i + \ell - 1$
        $m[i,j] = 1$
        for $k = i$ to $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i+1}p_kp_j$
            if $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
return $m$ and $s$
```

How MatrixChainOrder Works

- The algorithm computes $m[i,i] = 0$ for $i = 1, 2 \ldots, n$ in Lines 3-4.
- It then uses recurrence to compute $m[i, i+1]$, for $i = 1, 2, \ldots, n-1$, during the first execution of the loop in Lines 5-13.
- The second time through the loop, it computes $m[i, i+2]$, for $i = 1, \ldots, n-2$, and so on.
- At each step the $m[i,j]$ cost computed in Lines 10-13 depends only on the entries $m[i,k]$ and $m[k+1,j]$ that have already been computed.





Time and Space Requirements of MatrixChainOrder

- The nested loop structure of MatrixChainOrder yields a running time of $O\left(n^3\right)$ for the algorithm:

  The loops are nested three deep, and each loop index (ℓ, i and k) takes on at most $n-1$ values.
- The running time of this algorithm is in fact also $\Omega(n^3)$.
- The algorithm requires $\Theta(n^2)$ space to store the m and s tables.
- MatrixChainOrder is much more efficient than the exponential time method of enumerating all possible parenthesizations and checking each one.

Step 4: Constructing an Optimal Solution

- MatrixChainOrder determines the optimal number of scalar multiplications needed to compute a matrix-chain product.
- To show how to multiply the matrices, we look at the table $s[1 \dots n-1; 2 \dots n]$.

  Each entry $s[i,j]$ records a value of k, such that an optimal parenthesization of $A_iA_{i+1}\cdots A_j$ splits the product between A_k and A_{k+1}.
- Thus, we know that the final matrix multiplication in computing $A_{1\dots n}$ optimally is $A_{1\dots s[1,n]}A_{s[1,n]+1\dots n}$.
- We can determine the earlier matrix multiplications recursively:
  - $s[1,s[1,n]]$ determines the last matrix multiplication when computing $A_{1\dots s[1,n]}$;
  - $s[s[1,n]+1,n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1\dots n}$.

Printing Optimal Parenthesization

- The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the s table computed by $\textsc{MatrixChainOrder}$ and the indices i and j.
- The initial call $\textsc{PrintOptimalParens}(s, 1, n)$ prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$.

$\textsc{PrintOptimalParens}(s, i, j)$

```
1. if $i == j$
2.     print "$A_i$"
3. else print "("
4.     $\textsc{PrintOptimalParens}(s, i, s[i,j])$
5.     $\textsc{PrintOptimalParens}(s, s[i,j]+1, j)$
6.     print ")"
```

Subsection 3

Elements of Dynamic Programming

Pattern in Optimal Substructure

- A common pattern in discovering optimal substructure:
    1. You show that a solution to the problem consists of making a choice, such as choosing an index at which to split the matrix chain.
       Making this choice leaves one or more subproblems to be solved.
    2. You suppose that for a given problem, you are given the choice that leads to an optimal solution.
       You assume the choice has been given to you.
    3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
    4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut and paste" technique.
       You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction.

The Space of Subproblems

- To characterize the space of subproblems, a good rule of thumb says to try to keep the space as simple as possible and then expand it as necessary.
  - The space of subproblems that we considered for the rod-cutting problem contained the problems of optimally cutting up a rod of length i for each size i.
    This subproblem space worked well, and we had no need to try a more general space of subproblems.
  - Conversely, suppose that we had tried to constrain our subproblem space for matrix-chain multiplication to matrix products of the form $A_1 A_2 \cdots A_j$. As before, an optimal parenthesization must split this product between A_k and A_{k+1}, for some $1 \leq k < j$.
    Unless we could guarantee that k always equals $j-1$, we would find that we had subproblems of the form $A_1 A_2 \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$, and that the latter subproblem is not of the form $A_1 A_2 \cdots A_j$. For this problem, we needed to allow our subproblems to vary at "both ends", i.e., to allow both i and j to vary in the subproblem $A_i A_{i+1} \cdots A_j$.

Optimal Substructure

- Optimal substructure varies across problem domains in two ways:
  1. How many subproblems an optimal solution uses;
  2. How many choices we have in determining which subproblem(s) to use in an optimal solution.
- In the rod-cutting problem, an optimal solution for cutting up a rod of size n uses just one subproblem (of size $n - i$), but we must consider n choices for i in order to determine which one yields an optimal solution.
- Matrix-chain multiplication for the subchain $A_iA_{i+1}\cdots A_j$ serves as an example with two subproblems and $j - i$ choices.

  For given A_k at which the split occurs, we have two subproblems:
  - Parenthesizing $A_iA_{i+1}\cdots A_k$;
  - Parenthesizing $A_{k+1}A_{k+2}\cdots A_j$.

  Once we determine the optimal solutions to subproblems, we choose from among $j - i$ candidates for the index k.

The Running Time

- Informally, the running time of a dynamic-programming algorithm depends on the product of two factors:
  - The number of subproblems overall;
  - How many choices we look at for each subproblem.
- In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $O\left(n^2\right)$ running time.
- Matrix-chain multiplication had $\Theta(n^2)$ subproblems overall, and in each we had at most $n-1$ choices, giving an $O\left(n^3\right)$ running time (actually, a $\Theta(n^3)$ running time).
- Usually, the subproblem graph gives an alternative way to perform the same analysis.
  - Each vertex corresponds to a subproblem;
  - The choices for a sub problem are the edges incident to that subproblem.

Two Graph Problems: Optimal Substructure

- Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.
- **Unweighted Shortest Path**:

  Find a path from u to v consisting of the fewest edges.

  Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.
- **Unweighted Longest Simple Path**:

  Find a simple path from u to v consisting of the most edges.

  We need to include the requirement of simplicity because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.

The Unweighted Shortest-Path Problem

- The unweighted shortest-path problem exhibits optimal substructure.

  Suppose that $u \neq v$, so that the problem is nontrivial. Then, any path p from u to v must contain an intermediate vertex, say w. w may be u or v. Thus, we can decompose the path $u \overset{p}{\rightsquigarrow} v$ into subpaths $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$. Clearly, the number of edges in p equals the number of edges in p_1 plus the number of edges in p_2.

  Claim: If p is an optimal (shortest) path from u to v, then p_1 must be a shortest path from u to w.

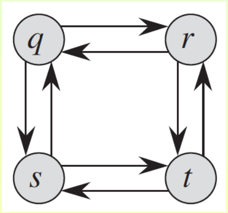
  We use a "cut-and-paste" argument: If there were another path, say p_1' from u to w with fewer edges than p_1, then we could cut out p_1 and paste in p_1' to produce a path $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ with fewer edges than p, thus contradicting p's optimality.

  Symmetrically, p_2 must be a shortest path from w to v.

The Unweighted Shortest-Path Problem: Method

- The fact that the unweighted shortest-path problem exhibits optimal substructure implies that we can find a shortest path from u to v by:
  - Considering all intermediate vertices w;
  - Finding:
    - a shortest path from u to w;
    - a shortest path from w to v;
  - Choosing an intermediate vertex w that yields the overall shortest path.

Finding a Longest Simple Path

- The problem of finding an unweighted longest simple path does not exhibit optimal substructure.
- If we decompose a longest simple path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, then it is not necessarily the case that p_1 must be a longest simple path from u to w, and p_2 a longest simple path from w to v.
- 

Consider the path $q \to r \to t$, which is a longest simple path from q to t. $q \to r$ is not a longest simple path from q to r: The path $q \to s \to t \to r$ is a simple path that is longer. $r \to t$ is not a longest simple path from r to t: The path $r \to q \to s \to t$ is a simple path that is longer.
- No efficient dynamic-programming algorithm for this problem has ever been found. In fact, this problem is NP-complete.

Dependence of Subproblems

- Although a solution to a problem for both longest and shortest paths uses two subproblems, the subproblems in finding the longest simple path are not independent, whereas for shortest paths they are.
- For the example above, we have the problem of finding a longest simple path from q to t with two subproblems:
    - Finding longest simple paths from q to r;
    - Finding longest simple paths from r to t.

  Choosing the path $q \to s \to t \to r$, we use the vertices s and t.
    - We can no longer use these vertices in the second subproblem, since the combination of the two solutions to subproblems would yield a path that is not simple.
    - If we cannot use vertex t in the second problem, then we cannot solve it at all, since t is required to be on the path that we find. Moreover, t is not the vertex at which we are "splicing" together the subproblem solutions (that vertex being r).

Independence of Subproblems

- In contrast, the subproblems for finding a shortest path do not share resources and are, thus, independent.

  Claim: If a vertex w is on a shortest path p from u to v, then we can splice together any shortest path $u \overset{p_1}{\leadsto} w$ and any shortest path $w \overset{p_2}{\leadsto} v$ to produce a shortest path from u to v. We are assured that, other than w, no vertex can appear in both paths p_1 and p_2.

  Suppose that some vertex $x \neq w$ appears in both p_1 and p_2, so that we can decompose p_1 as $u \overset{p_{ux}}{\leadsto} x \leadsto w$ and p_2 as $w \leadsto x \overset{p_{xv}}{\leadsto} v$. By the optimal substructure of this problem, path p has as many edges as p_1 and p_2 together, say that p has e edges. Now let us construct a path $p' = u \overset{p_{ux}}{\leadsto} x \overset{p_{xv}}{\leadsto} v$ from u to v. Because we have excised the paths from x to w and from w to x, each of which contains at least one edge, path p' contains at most $e-2$ edges, which contradicts the assumption that p is a shortest path. Thus, we are assured that the subproblems for the shortest-path problem are independent.

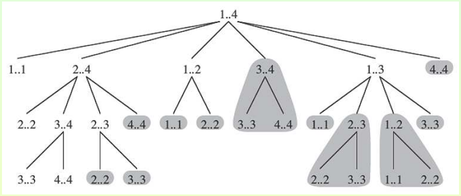
Overlapping Subproblems

- The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be "small".

  A recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.
- Typically, the total number of distinct subproblems is a polynomial in the input size.
- When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has **overlapping subproblems**.
- In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion.
- Dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed.

Illustration of the Overlapping-Subproblems Property

- We reexamine the matrix-chain multiplication problem.



Observe that MATRIXCHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows.

For example, it references entry $m[3,4]$ four times: during the computations of $m[2,4]$, $m[1,4]$, $m[3,5]$ and $m[3,6]$.

If $m[3,4]$ was recomputed each time, the running time would increase dramatically.

A Recursive Matrix Chain Procedure

- The following inefficient procedure computes $m[i,j]$, the minimum number of scalar multiplications needed to compute $A_{i..j} = A_i A_{i+1} \cdots A_J$:

RecursiveMatrixChain(p,i,j)

```
1. if $i == j$
2.     return 0
3. $m[i,j] = \infty$
4. for $k = i$ to $j - 1$
5.     $q = $ RecursiveMatrixChain$(p,i,k)$
           $+$RecursiveMatrixChain$(p,k+1,j) + p_{i-1}p_kp_j$
6.     if $q < m[i,j]$
7.         $m[i,j] = q$
8. return $m[i,j]$
```

Time Requirements of the Recursive Procedure

- Let $T(n)$ be the time taken by RECURSIVEMATRIXCHAIN to compute an optimal parenthesization of a chain of n matrices.
  The execution of Lines 1-2 and Lines 6-7 each take at least unit time.
  The multiplication in Line 5 also takes at least unit time.
  Thus,
  $$\begin{array}{rcl} T(1) & \geq & 1; \\ T(n) & \geq & 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1), \quad n>1. \end{array}$$
  Noting that for $i=1,2,\ldots,n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we get
  $$T(n) \geq 2\sum_{i=1}^{n-1} T(i)+n.$$
  We shall prove that $T(n)=\Omega(2^n)$ using the substitution method.

Time Requirements of the Recursive Procedure (Cont'd)

- We obtained
$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n.$$
We show that $T(n) \geq 2^{n-1}$, for all $n \geq 1$.
For the basis $T(1) \geq 1 = 2^0$.
Inductively, for $n \geq 2$, we have
$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n = 2^n - 2 + n \\ &\geq 2^{n-1}. \end{aligned}$$

Reconstructing an Optimal Solution

- As a practical matter, we often store which choice we made in each subproblem in a table so that we do not have to reconstruct this information from the costs that we stored.
- For matrix-chain multiplication, the table $s[i,j]$ saves us a significant amount of work when reconstructing an optimal solution.
- Suppose that we did not maintain the $s[i,j]$ table, having filled in only the table $m[i,j]$ containing optimal subproblem costs.

  We choose from among $j-i$ possibilities when we determine which subproblems to use in an optimal solution to parenthesizing $A_iA_{i+1}\cdots A_j$, and $j-i$ is not a constant.

  Therefore, it would take $\Theta(j-i)=\omega(1)$ time to reconstruct which subproblems we chose for a solution to a given problem.
- By storing in $s[i,j]$ the index of the matrix at which we split the product $A_iA_{i+1}\cdots A_j$, we can reconstruct each choice in $O(1)$ time.

Memoization

- An alternative approach to dynamic programming that offers the efficiency of the bottom-up dynamic programming approach while maintaining a top-down strategy is the **memoization** of the natural, but inefficient, recursive algorithm.
- As in the bottom-up approach, we maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.
- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.
    - Each table entry initially contains a special value to indicate that the entry has yet to be filled in.
    - When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table.
    - Each subsequent time that we encounter this subproblem, we simply look up the value stored in the table and return it.

The Memoized Matrix Chain Procedure

MemoizedMatrixChain(p)

```
1. $n = p.\text{length} - 1$
2. let $m[1 \ldots n, 1 \ldots n]$ be a new table
3. for $i = 1$ to $n$
4.     for $j = i$ to $n$
5.         $m[i,j] = \infty$
6. return LookupChain($m, p, 1, n$)
```

LookupChain(m, p, i, j)

```
1. if $m[i,j] < 1$
2.     return $m[i,j]$
3. if $i == j$
4.     $m[i,j] = 0$
5. else for $k = i$ to $j - 1$
6. $q =$ LookupChain($m, p, i, k$) $+$ LookupChain($m, p, k+1, j$) $+ p_{i+1} p_k p_j$
7.     if $q < m[i,j]$
8.         $m[i,j] = q$
9. return $m[i,j]$
```

How MemoizedMatrixChain Works

- The MemoizedMatrixChain, like MatrixChainOrder, maintains a table $m[1 \ldots n, 1 \ldots n]$ of computed values of $m[i,j]$, the minimum number of scalar multiplications needed to compute the matrix $A_{i \ldots j}$.
- Each table entry initially contains the value ∞ to indicate that the entry has yet to be filled in.
- Upon calling LookupChain(m,p,i,j):
  - If Line 1 finds that $m[i,j] < \infty$, then the procedure simply returns the previously computed cost $m[i,j]$ in Line 2;
  - Otherwise, the cost is computed as in RecursiveMatrixChain, stored in $m[i,j]$, and returned.
- Thus, LookupChain(m,p,i,j) always returns the value of $m[i,j]$, but it computes it only upon the first call of LookupChain with these specific values of i and j.

Time Requirements

- Like MatrixChainOrder, MemoizedMatrixChain runs in $O(n^3)$ time:

  Line 5 of MemoizedMatrixChain executes $\Theta(n^2)$ times.

  We can categorize the calls of LookupChain into two types:
  1. Calls in which $m[i,j] = \infty$, so that Lines 3-9 execute;
  2. Calls in which $m[i,j] < \infty$, so that LookupChain returns in Line 2.

  There are $\Theta(n^2)$ calls of the first type, one per table entry.

  All calls of the second type are made as recursive calls by calls of the first type. Whenever a given call of LookupChain makes recursive calls, it makes $O(n)$ of them. Therefore, there are $O(n^3)$ calls of the second type in all.

  Each call of the second type takes $O(1)$ time, and each call of the first type takes $O(n)$ time plus the time spent in its recursive calls.

  The total time, therefore, is $O(n^3)$. Memoization, thus, turns an $\Omega(2^n)$-time algorithm into an $O(n^3)$-time algorithm.

Subsection 4

Longest Common Subsequence

DNA and Similarity

- A strand of DNA consists of a string of molecules called **bases**, where the possible bases are adenine, guanine, cytosine and thymine.
- Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set $\{A, C, G, T\}$.
- One reason to compare two strands of DNA is to determine how "similar" the two strands are.
- This constitutes a measure of how closely related the two organisms are.

DNA Similarity

- We can define similarity of strands S_1 and S_2 in many different ways:
  - We can say that two DNA strands are similar if one is a substring of the other.
  - We could say that two strands are similar if the number of changes needed to turn one into the other is small.
  - We could find a third strand S_3 whose bases appear in each of S_1, S_2. These bases must appear in the same order, but not necessarily consecutively.
    The longer the strand S_3 we can find, the more similar S_1 and S_2 are.

Example of DNA Comparison

Example: The DNA of one organism may be

$$S_1 = \texttt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

and the DNA of another organism may be

$$S_2 = \texttt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}.$$

Suppose we measure the similarity of strands S_1 and S_2 by finding a third strand S_3, such that the bases in S_3 appear in each of S_1 and S_2 in the same order, but not necessarily consecutively.

Then, the longest strand is

$$S_3 = \texttt{GTCGTCGGAAGCCGGCCGAA}.$$

Longest Common Subsequences (LCSs)

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, a sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X, such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.

  Example: For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$, with index sequence $\langle 2, 3, 5, 7 \rangle$.
- Given two sequences X and Y, a sequence Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y.

  Example: If $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y.

  The sequence $\langle B, C, A \rangle$ is not a longest common subsequence (LCS) of X and Y, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both X and Y, has length 4.

  Both $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are LCSs of X and Y, since X and Y have no common subsequence of length 5 or greater.

The Longest-Common-Subsequence Problem

- The **longest-common-subsequence problem**:

  Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find a maximum length common subsequence of X and Y.
- A brute-force approach to solving the LCS problem:
  - Enumerate all subsequences of X;
  - Check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence found.
- Each subsequence of X corresponds to a subset of the indices $\langle 1, 2, \ldots, m \rangle$ of X.

  Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

The Dynamic Programming Approach

- In a dynamic programming approach, the subproblems correspond to pairs of "prefixes" of the two input sequences.
- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, the *i*-**th prefix** of X, for $i = 0, 1, \ldots, m$, is $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.

  Example: If $X = \langle A, B, C, B, D, A, B \rangle$, then:
    - $X_4 = \langle A, B, C, B \rangle$;
    - X_0 is the empty sequence.

Step 1: Characterizing a Longest Common Subsequence

Theorem (Optimal Substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

(1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k+1$, contradicting the supposition that Z is a longest common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k-1)$ common subsequence of X_{m-1} and Y_{n-1}.

Characterizing a Longest Common Subsequence

Claim: The prefix Z_{k-1} is a length-$(k-1)$ LCS of X_{m-1} and Y_{n-1}.

Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k-1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2).

Step 2: A Recursive Solution

- The theorem implies that we should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.
  - If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y.
  - If $x_m \neq y_n$, then we must solve two subproblems:
    - Finding an LCS of X_{m-1} and Y;
    - Finding an LCS of X and Y_{n-1}.

  Whichever of these two LCSs is longer is an LCS of X and Y.
- Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must appear within an LCS of X and Y.
- We can readily see the overlapping subproblems property in the LCS problem: To find an LCS of X and Y, we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. Each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}.

The Recursive Formula

- Define $c[i,j]$ to be the length of an LCS of the sequences X_i and Y_j.
  - If either $i=0$ or $j=0$, one of the sequences has length 0, and so the LCS has length 0.
  - The optimal substructure of the LCS problem gives the recursive formula

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

In this recursive formulation, a condition in the problem restricts which subproblems we may consider:

- When $x_i = y_j$, we can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}.
- Otherwise, we instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j.

Step 3: Computing the Length of an LCS

- Since the LCS problem has only $\Theta(mn)$ distinct subproblems, we can use dynamic programming to compute the solutions bottom up.
- Procedure LCSLength takes two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as inputs.
- It stores the $c[i,j]$ values in a table $c[0 \ldots m, 0 \ldots n]$, and it computes the entries in row-major order, i.e., the procedure fills in the first row of c from left to right, then the second row, and so on.
- The procedure also maintains the table $b[1 \ldots m, 1 \ldots n]$ to help us construct an optimal solution. Intuitively, $b[i,j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i,j]$.
- The procedure returns the b and c tables;

  $c[m,n]$ contains the length of an LCS of X and Y.

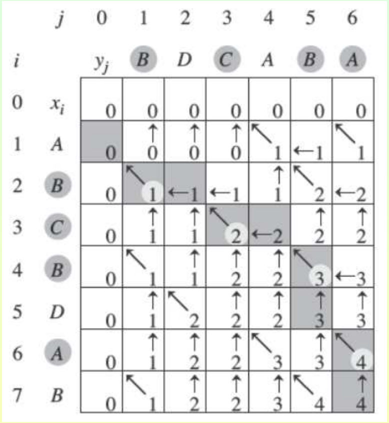
The Procedure LCS Length

LCSLength(X, Y)

```
1. $m = X$.length
2. $n = Y$.length
3. let $b[1 \ldots m, 1 \ldots n]$ and $c[0 \ldots m, 0 \ldots n]$ be new tables
4. for $i = 1$ to $m$
5.    $c[i, 0] = 0$
6. for $j = 0$ to $n$
7.    $c[0, j] = 0$
8. for $i = 1$ to $m$
9.    for $j = 1$ to $n$
10.       if $x_i == y_j$
11.          $c[i, j] = c[i-1, j-1] + 1$
12.          $b[i, j] = $ "$\nwarrow$"
13.       elseif $c[i-1, j] \geq c[i, j-1]$
14.          $c[i, j] = c[i-1, j]$
15.          $b[i, j] = $ "$\uparrow$"
16.       else $c[i, j] = c[i, j-1]$
17.          $b[i, j] = $ "$\leftarrow$"
18. return $c$ and $b$
```

An Example

- The tables produced by LCSLength on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$:



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

- The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Step 4: Constructing an LCS

- To construct an LCS, we begin at $b[m,n]$ and trace through b by following the arrows.
- The elements of the LCS are found in reverse order, but the following prints them out in the correct order.
- The initial call is $\textsc{PrintLCS}(b, X, X.\text{lenth}, Y.\text{length})$.

$\textsc{PrintLCS}(b, X, i, j)$

```
1. If $i == 0$ or $j == 0$
2.    return
3. if $b[i.j] ==$ "$\nwarrow$"
4.    $\textsc{PrintLCS}(b, X, i-1, j-1)$
5.    print $x_i$
6. elseif $b[i,j] ==$ "$\uparrow$"
7.    $\textsc{PrintLCS}(b, X, i-1, j)$
8. else $\textsc{PrintLCS}(b, X, i, j-1)$
```

- The time needed is $O(m+n)$.

Subsection 5

Optimal Binary Search Trees

Optimal Binary Search Trees

- We are given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order (so that $k_1 < k_2 < \ldots < k_n$).
- We wish to build a binary search tree from these keys.
- For each key k_i, we have a probability p_i that a search will be for k_i.
- Some searches may be for values not in K.

  These are represented by $n+1$ "dummy keys" $d_0, d_1, \ldots, d_n$.

  Key d_i represents all values between k_i and k_{i+1}.
- Suppose for each d_i, there is a probability q_i that a search will correspond to d_i.
- We have
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Expected Cost of a Search

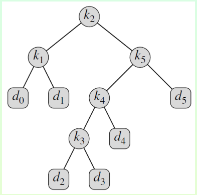
- Assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search plus 1.
- Then the expected cost of a search is

$$\begin{aligned} E[\text{Search Cost in } T] &= \sum_{i=1}^{n}(\text{depth}_T(k_i)+1)p_i + \sum_{i=0}^{n}(\text{depth}_T(d_i)+1)q_i \\ &= \sum_{i=1}^{n}p_i + \sum_{i=0}^{n}q_i + \sum_{i=1}^{n}\text{depth}_T(k_i)p_i + \sum_{i=0}^{n}\text{depth}_T(d_i)q_i. \\ &= 1 + \sum_{i=1}^{n}\text{depth}_T(k_i)p_i + \sum_{i=0}^{n}\text{depth}_T(d_i)q_i. \end{aligned}$$

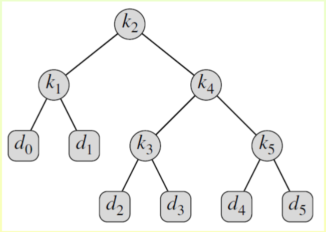
An Example

-

i	0	1	2	3	4	5
p_1		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



- A tree with smallest expected search cost is an **optimal binary search tree**.



Node	Depth	Probability	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.10
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Exhaustive Search

- An optimal binary search tree is not necessarily a tree whose overall height is smallest.
- Nor can we necessarily construct an optimal binary search tree by always putting the key with the greatest probability at the root.
- As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm.
  - We can label the nodes of any n-node binary tree with the keys $k_1, k_2, \ldots, k_n$ to construct a binary search tree, and then add in the dummy keys as leaves.
    The number of binary trees with n nodes is $\Omega(\frac{4^n}{n^{3/2}})$.
    So we would have to examine an exponential number of binary search trees in an exhaustive search.
- We shall solve this problem with dynamic programming.

Step 1: The Structure of an Optimal Binary Search Tree

- A subtree of a binary search tree must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \leq i \leq j \leq n$.
- In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.
- We state the optimal substructure:

  If an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.

  By cut-and-paste, if there were a subtree T'' whose expected cost is lower than that of T', then we could cut T' out of T and paste in T'', resulting in a binary search tree of lower expected cost than T, contradicting the optimality of T.

Step 1 (Cont'd)

- Given keys $k_i, \ldots, k_j$, one of these keys, say k_r ($i \leq r \leq j$), is the root of an optimal subtree containing these keys.
  - The left subtree of the root k_r contains the keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$);
  - The right subtree contains the keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$).
- As long as we examine all candidate roots k_r, where $i \leq r \leq j$, and we determine all optimal binary search trees containing $k_i, \ldots, k_{r-1}$ and those containing $k_{r+1}, \ldots, k_j$, we are guaranteed that we will find an optimal binary search tree.
- If in a subtree with keys $k_i, \ldots, k_j$, we select
  - k_i as the root, k_i's left subtree contains the keys $k_i, \ldots, k_{i-1}$, interpreted as a sequence with no keys, but only the single dummy key d_{i-1};
  - k_j as the root, then k_j's right subtree contains no actual keys, but only the dummy key d_j.

Step 2: A Recursive Solution

- To define the value of an optimal solution recursively, we pick our subproblem domain as finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \geq 1, j \leq n$, and $j \geq i - 1$.
- Define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$.

  Ultimately, we wish to compute $e[1, n]$.
  - The easy case occurs when $j = i - 1$. Then we have just the dummy key d_{i-1}. The expected search cost is $e[i, i - 1] = q_{i-1}$.
  - When $j \geq i$, we need to select a root k_r from among $k_i, \ldots, k_j$ and then create:
    - An optimal binary search tree with keys $k_i, \ldots, k_{r-1}$ as its left subtree;
    - An optimal binary search tree with keys $k_{r+1}, \ldots, k_j$ as its right subtree.

Step 2: The Recursive Cost Equation

- When a subtree becomes a subtree of a node, the depth of each node in the subtree increases by 1. Then, the expected search cost of this subtree increases by the sum of all the probabilities in the subtree.
- For a subtree with keys $k_i, \ldots, k_j$, let us denote this sum of probabilities as $w(i,j) = \sum_{\ell=i}^{j} p_\ell + \sum_{\ell=i-1}^{j} q_\ell$.
- Thus, if k_r is the root of an optimal subtree containing keys $k_i, \ldots, k_j$, we have

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)).$$

- Since $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$ we rewrite $e[i,j]$ as

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j).$$

- This recursive equation assumes that we know which node k_r to use as the root.

Step 2: The Recursive Optimal Cost Equation

- We choose the root that gives the lowest expected search cost:
  $$e[i,j] = \begin{cases} q_{i-1}, & \text{if } j = i-1 \\ \min_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\}, & \text{if } i \le j \end{cases}$$
- The $e[i,j]$ values give the expected search costs in optimal binary search trees.
- To help us keep track of the structure of optimal binary search trees, we define $\text{root}[i,j]$, for $1 \le i \le j \le n$, to be the index r for which k_r is the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$.
- We show how to compute the values of $\text{root}[i,j]$.
- We omit the procedure for constructing an optimal binary search tree from these values.

Step 3: Computing the Optimal Expected Search Cost

- We store the $e[i,j]$ values in a table $e[1 \ldots n+1, 0 \ldots n]$.
  - The first index needs to run to $n+1$: In order to have a subtree containing only the dummy key d_n, we need to compute and store $e[n+1,n]$.
  - The second index needs to start from 0: To have a subtree containing only the dummy key d_0, we need to compute and store $e[1,0]$.

  We use only the entries $e[i,j]$ for which $j \leq i-1$.
- A table root$[i,j]$ records the root of the subtree containing keys $k_i, \ldots, k_j$. This table uses only the entries for which $1 \leq i \leq j \leq n$.
- Rather than compute the value of $w(i,j)$ from scratch every time we are computing $e[i,j]$, which would take $\Theta(j-i)$ additions, we store these values in a table $w[1 \ldots n+1, 0 \ldots n]$.
  - For the base case, we compute $w[i,i-1] = q_{i-1}$, for $1 \leq i \leq n+1$.
  - For $j \geq i$, we compute $w[i,j] = w[i,j-1] + p_j + q_j$.

  Thus, we can compute the $\Theta(n^2)$ values of $w[i,j]$ in $\Theta(1)$ time each.

The Optimal Binary Search Tree Procedure

- The pseudocode takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and returns the tables e and root.

OPTIMALBST(p, q, n)

```
1.  let $e[1 \ldots n+1, 0 \ldots n]$, $w[1 \ldots n+1, 0 \ldots n]$, and root$[1 \ldots n, 1 \ldots n]$ be
    new tables
2.  for $i = 1$ to $n+1$
3.      $e[i, i-1] = q_{i-1}$
4.      $w[i, i-1] = q_{i-1}$
5.  for $\ell = 1$ to $n$
6.      for $i = 1$ to $n - \ell + 1$
7.          $j = i + \ell - 1$
8.          $e[i, j] = 1$
9.          $w[i, j] = w[i, j-1] + p_j + q_j$
10.         for $r = i$ to $j$
11.             $t = e[i, r-1] + e[r+1, j] + w[i, j]$
12.             if $t < e[i, j]$
13.                 $e[i, j] = t$
14.                 root$[i, j] = r$
15. return $e$ and root
```

How OPTIMALBST Works

- The for loop of Lines 2-4 initializes the values of $e[i, i-1]$ and $w[i, i-1]$.
- The for loop of Lines 5-14 then uses the previously obtained recurrences to compute $e[i, j]$ and $w[i, j]$, for all $1 \leq i \leq j \leq n$.
    - In the first iteration, when $\ell = 1$, the loop computes $e[i, i]$ and $w[i, i]$, for $i = 1, 2, \ldots, n$.
    - The second iteration, with $\ell = 2$, computes $e[i, i+1]$ and $w[i, i+1]$, for $i = 1, 2, \ldots, n-1$, and so forth.
- The innermost for loop, in Lines 10-14, tries each candidate index r to determine which key k_r to use as the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$.

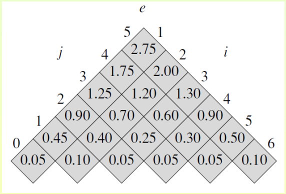
  This for loop saves the current value of the index r in root$[i, j]$ whenever it finds a better key to use as the root.

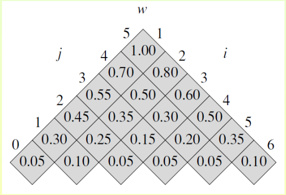
Example

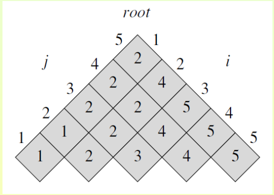
- For the key distribution

i	0	1	2	3	4	5
p_1		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

the following shows the tables $e[i,j], w[i,j]$ and $\text{root}[i,j]$ computed by the procedure OPTIMALBST:







OPTIMALBST computes the rows from bottom to top and left to right within each row.

Time Requirements

- OPTIMALBST takes time $\Theta(n^3)$.
- Its running time is $O\left(n^3\right)$:
    - Its for loops are nested three deep and each loop index takes at most n values.
- Like with MATRIXCHAINORDER, it also takes $\Omega(n^3)$ time as well, since it has an almost identical structure modulo a unit difference on the bounds of the loop indices.