CSE 5311: Design and Analysis of Algorithms

Dynamic Programming

Alex Dillhoff

University of Texas at Arlington

Dynamic Programming

Dynamic programming is a technique for solving problems by breaking them down into simpler subproblems, very much like divide and conquer algorithms.

Subproblems are designed in such a way that they do not need to be recomputed.

Dynamic Programming

A dynamic programming solution can be applied if the problem has the following features:

- **Optimal substructure:** An optimal solution can be constructed by optimal solutions to the subproblems.
- **Overlapping subproblems:** The problem can be broken down into subproblems which can be reused.

Dynamic Programming

Fibonacci sequence

- **Optimal substructure:** the value of the sequence at any index is the sum of the values at the two previous indices.
- **Overlapping subproblems:** the value of the sequence at any index is used in the calculation of the values at the two subsequent indices.

Dynamic Programming

A recursive solution to the Fibonacci sequence will have exponential time complexity.

A dynamic programming solution will have **linear time** complexity.

Dynamic Programming

Two main approaches to dynamic programming...

1. **Memoization (top-down):** involves writing a recursive solution that stores each sub-solution in a table so that it can be reused.
2. **Tabulation (bottom-up):** involves solving the problem by filling in a table of subproblems from the bottom up.

Dynamic Programming

In either case, the four steps of dynamic programming are the same:

1. **Identify subproblems** so that the problem can be broken down.

Dynamic Programming

In either case, the four steps of dynamic programming are the same:

1. **Identify subproblems** so that the problem can be broken down.
2. **Solve the subproblems** following an optimal solution.

Dynamic Programming

In either case, the four steps of dynamic programming are the same:

1. **Identify subproblems** so that the problem can be broken down.
2. **Solve the subproblems** following an optimal solution.
3. **Store the solutions** to avoid redundant computation.

Dynamic Programming

In either case, the four steps of dynamic programming are the same:

1. **Identify subproblems** so that the problem can be broken down.
2. **Solve the subproblems** following an optimal solution.
3. **Store the solutions** to avoid redundant computation.
4. **Combine solutions** from the subproblems to solve the original problem.

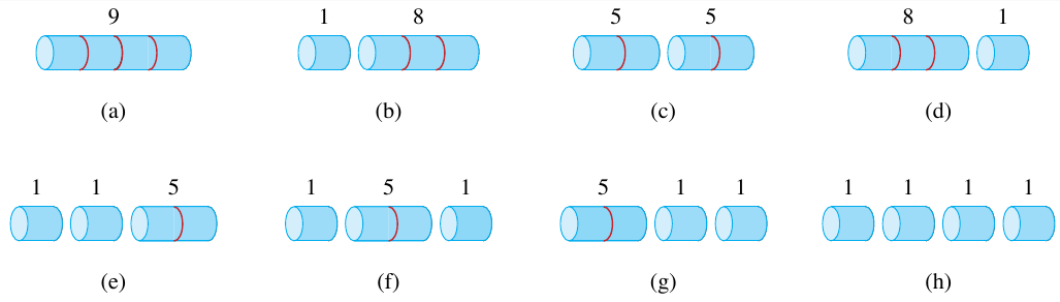
Rod Cutting

Rod Cutting

Given a rod of length n and table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n that can be obtained by cutting up the rod and selling the pieces.

Length	1	2	3	4	5	6	7	8	9
Price	1	5	8	9	10	17	17	20	24

Rod Cutting



8 different ways to cut a rod of length 4 (Cormen et al.).

Rod Cutting

The maximum revenue for a rod of length n can be determined by the following optimization problem:

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

where r_i is the maximum revenue for a rod of length i.

Rod Cutting

The maximum revenue for a rod of length n can be determined by solving the subproblems for rods of length i for $i = 1, 2, \ldots, n - 1$.

Rod Cutting

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

- Each of the terms r_i in the equation implies a recursive solution to the problem.
- Solving this recursively would lead to many redundant computations.
- For example, r_1 is computed at least twice in the equation above.

Rod Cutting

This recursion is more compactly written as

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

Rod Cutting

This problem has **optimal substructure**.

If we cut the rod into smaller subsections, we can recursively solve the subproblems and combine them.

Rod Cutting

```
def cut_rod(p, n):
    if n == 0:
        return 0
    q = -float('inf')
    for i in range(1, n+1):
        q = max(q, p[i] + cut_rod(p, n-i))
    return q
```

Rod Cutting Analysis

If $T(n)$ is a recurrence that represents the number of times `cur_rod` is called recursively, then we can write the following recurrence relation:

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j).$$

Rod Cutting Analysis

Using the substitution method, we can show that $T(n) = 2^n$.

$$\begin{aligned} T(n) &= 1 + \sum_{j=0}^{n-1} 2^j \\ &= 1 + (2^n - 1) \\ &= 2^n. \end{aligned}$$

Memoization

To solve this with dynamic programming, the goal is to make sure that each subproblem is computed *only once*.

Memoization

To solve this with dynamic programming, the goal is to make sure that each subproblem is computed *only once*.

This is accomplished by saving the result of each subproblem in a table so that it can be reused.

This does incur a space complexity of $O(n)$, but it reduces the time complexity to $O(n^2)$.

Memoization

Process

- The solution requires a small modification to the recursive algorithm.
- When the solution to a subproblem is required, the table is first checked for a stored solution.
- If the solution is not found, the subproblem is solved recursively and the solution is stored in the table.

Memoization

```
def memoized_cut_rod(p, n):
    r = [-float('inf') for _ in range(n+1)]
    return memoized_cut_rod_aux(p, n, r)

def memoized_cut_rod_aux(p, n, r):
    if r[n] >= 0:
        return r[n]
    if n == 0:
        q = 0
    else:
        q = -float('inf')
        for i in range(1, n+1):
            q = max(q, p[i] + memoized_cut_rod_aux(p, n-i, r))
    r[n] = q
    return q
```

Memoization Analysis

The algorithm starts off with a call to `memoized_cut_rod` which initializes the table `r` and then calls `memoized_cut_rod_aux`.

The table `r` is initialized with $-\infty$ so that we can check if a solution has been computed for a subproblem.

Memoization Analysis

Each subproblem is solved only once, leading to $O(1)$ lookups after that.

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

Tabulation

The other dynamic programming solution is to first sort the subproblems by their size, solve the smaller ones first, and build up to the larger ones.

This is called **tabulation** and the time complexity of this solution is also $O(n^2)$.

Tabulation

```
def bottom_up_cut_rod(p, n):
    r = [0 for _ in range(n+1)]
    for j in range(1, n+1):
        q = -float('inf')
        for i in range(1, j+1):
            q = max(q, p[i] + r[j-i])
        r[j] = q
    return r[n]
```

Tabulation

The first `for` loop effectively sorts the problem by size.

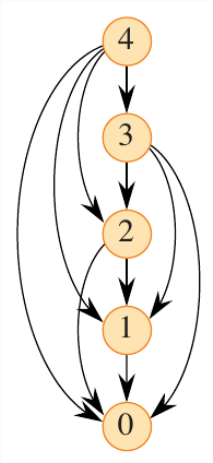
It starts with a cut of size 1 and builds up to a cut of size n.

Subproblem Graphs

Subproblem graphs offer a concise way to visualize the subproblems and their dependencies.

Consider the rod cutting problem with $n = 4$.

Subproblem Graphs



Subproblem graph for the rod cutting problem (Cormen et al.).

Subproblem Graphs

Subproblem $n = 4$ is dependent on subproblems $n = 3$, $n = 2$, and $n = 1$.

The bottom-up approach follows this dependency by ensuring that the subproblems are solved in the correct order.

Subproblem Graphs

Besides serving as a helpful visualization, depicting the problem using a DAG can also help to identify the time complexity of the problem.

This is the sum of the time needed to solve each subproblem.

Subproblem Graphs

Each problem of size n requires $n-1$ subproblems to be solved, and each subproblem of size $n-1$ requires $n-2$ subproblems to be solved.

This leads to a time complexity of $O(n^2)$.

Reconstructing a Solution

The two dynamic programming solutions above return the maximum revenue that can be obtained by cutting up the rod.

However, they do not return the actual cuts that should be made.

This can be done by modifying the algorithms to store the cuts that are made.

Reconstructing a Solution

```
def extended_bottom_up_cut_rod(p, n):
    r = [0 for _ in range(n+1)]
    s = [0 for _ in range(n+1)]
    for j in range(1, n+1):
        q = -float('inf')
        for i in range(1, j+1):
            if q < p[i] + r[j-i]:
                q = p[i] + r[j-i]
                s[j] = i
        r[j] = q
    return r, s
```

Reconstructing a Solution

```
def print_cut_rod_solution(p, n):
    r, s = extended_bottom_up_cut_rod(p, n)
    while n > 0:
        print(s[n])
        n -= s[n]
```

Reconstructiong a Solution

In the bottom-up approach, the table `s` is used to store the size of the first piece to cut off.

The function `print_cut_rod_solution` uses this table to print the cuts that should be made.

Matrix-chain Multiplication

Matrix-chain Multiplication

Given a sequence of matrices $A_1, A_2, \ldots, A_n$, where the dimensions of matrix A_i are $p_{i-1} \times p_i$, determine the most efficient way to multiply the matrices.

The problem is to determine the order in which the matrices should be multiplied so that the number of scalar multiplications is minimized.

Matrix-chain Multiplication

Consider three matrices $A \in \mathbb{R}^{10\times100}$, $B \in \mathbb{R}^{100\times5}$, and $C \in \mathbb{R}^{5\times50}$.

Scalar multiplications required for

- $(AB)C$ is $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ and
- $A(BC)$ is $10 \times 100 \times 50 + 100 \times 5 \times 50 = 75000$.

Matrix-chain Multiplication

- Matrix multiplication is associative, so the order in which the matrices are grouped does not matter.
- The key to solving this problem is to find the most efficient way to **group** the matrices.
- The first part of the solution is to determine the number of possible groupings, or parenthesizations.

Determining Parenthesizations

The number of possible parenthesizations of a chain of n matrices is given by $P(n)$. When $n \geq 2$, the number of possible parenthesizations is given by

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k).$$

Dynamic Programming Solution

Cormen et al. suggest a 4 step process to construct a dynamic programming solution to the matrix-chain multiplication problem:

1. **Structure of an optimal solution**
2. **Recursive solution**
3. **Compute the optimal costs**
4. **Computing the optimal solution**

Optimal Substructure

What is the optimal substructure of this problem?

Consider matrix-chain sequence $A_{i:j} = A_i A_{i+1} \cdots A_j$.

If we split the sequence at k, then the optimal solution to the problem is the optimal solution to the subproblems $A_{i:k}$ and $A_{k+1:j}$.

Optimal Substructure

The number of scalar multiplications required to compute $A_{i:j}$ is the sum of the number of scalar multiplications required to compute $A_{i:k}$ and $A_{k+1:j}$

combined with

Optimal Substructure

The number of scalar multiplications required to compute $A_{i:j}$ is the sum of the number of scalar multiplications required to compute $A_{i:k}$ and $A_{k+1:j}$

combined with

the number of scalar multiplications required to compute the product of the two subproblems.

Optimal Substructure

How can we ensure that there is not a more optimal grouping of $A_{h:l}$, where $i \leq h < k$ and $k < l \leq j$?

Optimal Substructure

How can we ensure that there is not a more optimal grouping of $A_{h:l}$, where $i \leq h < k$ and $k < l \leq j$?

The answer lies in evaluating **all** possible splits.

Recursive Solution

What is the cost of an optimal solution to the problem?

We must first compute the minimum cost of parenthesizing $A_{i:j}$ for $1 \leq i \leq j \leq n$.

Recursive Solution

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute $A_{i:j}$.

Starting with the base case, $m[i, i]$ is the cost to compute the multiplication of a single matrix, which is 0.

Assuming optimal subproblems are chosen,
$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$, where the last term is the cost of multiplying $A_{i:k} A_{k+1:j}$.

Recursive Solution

All possible splits must be evaluated: **how many are there?**

Omitting the first and last matrices, there are $j - i$ possible splits.

Recursive Solution

We can now define the optimal solution in terms of the following recursion:

$$m[i,j] = \min\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j : i \leq k < j\}.$$

Storing the Solutions

This is no better than the brute-force method until we figure out how to select the optimal subproblems and store their solutions.

How can we optimally select k?

Storing the Solutions

How can we optimally select k?

A bottom-up approach involves computing the cost of all possible combinations of the n matrices and building up from there.

This requires $O(n^2)$ memory to store both the costs $m[i, j]$ as well as the value of k that splits them $s[i, j]$.

Storing the Solutions

```
def matrix_chain_order(p):
    n = len(p) - 1
    m = [[0 for _ in range(n)] for _ in range(n)]
    s = [[0 for _ in range(n)] for _ in range(n)]
    for l in range(2, n+1):          # chain length
        for i in range(1, n-l+2):    # start index
            j = i + l - 1            # end index
            m[i][j] = float('inf')
            for k in range(i, j):
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                if q < m[i][j]:
                    m[i][j] = q      # save the cost
                    s[i][j] = k      # save the index
    return m, s
```

Storing the Solutions

- This computes the cost of all possible combinations of the n matrices and stores the value of k that splits them.
- The outer-most `for` loop controls the length of the chain being evaluated.
- It starts at 2 since the cost of a length 1 chain is 0.
- Intuition tells us that the triply-nested `for` loop has a time complexity of $O(n^3)$.

Storing the Solutions

This algorithm computes the cost in ascending order of chain length.

- When $l = 2$, the cost of all chains of length 2 is computed.
- When $l = 3$, the cost of all chains of length 3 is computed, and so on.
- The recursion in the inner-most nested loop will only ever access the entries in `m` which have been previously computed.

Reconstructing a Solution

We now have a solution which generates the optimal number of scalar multiplications needed for all possible combinations of the n matrices.

Reconstructing a Solution

We now have a solution which generates the optimal number of scalar multiplications needed for all possible combinations of the n matrices.

We do not yet have a solution which tells us the order in which the matrices should be multiplied: **this information is held in `s`, which records the value of k that splits the chain.**

Reconstructing a Solution

```
def print_optimal_parens(s, i, j):
    if i == j:
        print(f"A_{i}", end="")
    else:
        print("(", end="")
        print_optimal_parens(s, i, s[i][j])
        print_optimal_parens(s, s[i][j]+1, j)
        print(")", end="")
```

Reconstructing a Solution

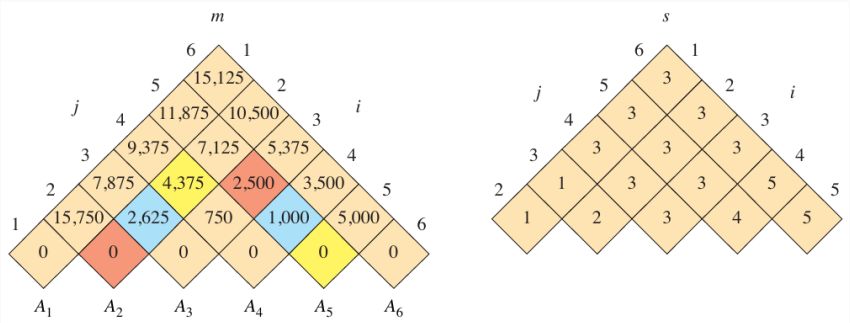


Figure 14.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Stored tables from calling Matrix-Chain-Order (Cormen et al.).

Reconstructing a Solution

First call is to `print_optimal_parens(s, 1, 6)`.

This recursively calls `print_optimal_parens(s, 1, 3)` and `print_optimal_parens(s, 4, 6)`.

Reconstructing a Solution

Second call: `print_optimal_parens(s, 1, 3)`

- This recursively calls `print_optimal_parens(s, 1, 1)` and `print_optimal_parens(s, 2, 3)`.
- This first call has $i == j$, so it prints A_1.
- The second call prints $(A_2 A_3)$.
- The initial call already set up the first set of parenthesis, so the intermediate result is $((A_1(A_2 A_3)) \cdots)$.

Reconstructing a Solution

Third call: `print_optimal_parens(s, 4, 6)`

- This recursively calls `print_optimal_parens(s, 4, 5)` and `print_optimal_parens(s, 6, 6)`.
- This will recursively call `print_optimal_parens(s, 4, 4)` and `print_optimal_parens(s, 5, 5)`.
- This produces (A_4A_5) from the first subcall and A_6 from the second subcall.
- The intermediate result is now $(\cdots((A_4A_5)A_6))$.

Reconstructing a Solution

Putting it all together

Combining these results yields

$$((A_1(A_2A_3))((A_4A_5)A_6)).$$

This is the optimal parenthesization of the matrix chain $A_1A_2A_3A_4A_5A_6$.

Applying Dynamic Programming

Applying Dynamic Programming

As shown in previous examples, determining the **optimal substructure** is the first step in formulating a dynamic programming solution.

In most cases, this comes from understanding the problem itself well: it is the result of a natural way of analysis and decomposition of the problem.

Determining Optimal Substructure

Determining the optimal substructure of a problem can be accomplished by following the steps below.

1. Show that a solution to a problem requires making a choice, like where to cut in the rod cutting problem.

Determining Optimal Substructure

Determining the optimal substructure of a problem can be accomplished by following the steps below.

1. Show that a solution to a problem requires making a choice, like where to cut in the rod cutting problem.
2. Assume that you are given an optimal choice.

Determining Optimal Substructure

Determining the optimal substructure of a problem can be accomplished by following the steps below.

1. Show that a solution to a problem requires making a choice, like where to cut in the rod cutting problem.
2. Assume that you are given an optimal choice.
3. Identify the subproblems that result from this choice.

Determining Optimal Substructure

Determining the optimal substructure of a problem can be accomplished by following the steps below.

1. Show that a solution to a problem requires making a choice, like where to cut in the rod cutting problem.
2. Assume that you are given an optimal choice.
3. Identify the subproblems that result from this choice.
4. Show that solutions to these subproblems are optimal.

Determining Optimal Substructure

In the last step, **we are typically looking for a contradiction.**

The assumption of step 2 means that if we end up finding a more optimal solution to a subproblem, then the original choice was not optimal.

The result is that we have a better overall solution.

Determining Optimal Substructure

- The efficiency of a solution depends on the number of subproblems times the number of choices we have for each subproblem.
- It is better to start with a simple case and expand outward as necessary.
- Using a subproblem graph is a great way to visualize the subproblems and their dependencies.

Counter-Example: The Longest Simple Path

Consider the following problems which first appear to have optimal substructure.

1. **Shortest path**: find a path $u \rightsquigarrow v$ with the fewest edges without cycles.
2. **Longest simple path**: find a path $u \rightsquigarrow v$ with the most edges without cycles.

Counter-Example: The Longest Simple Path

The first problem has optimal substructure.

- Suppose that the shortest path $u \rightsquigarrow v$ is given by p.

Counter-Example: The Longest Simple Path

The first problem has optimal substructure.

- Suppose that the shortest path $u \rightsquigarrow v$ is given by p.
- Given some intermediate vertex w, the optimal path from $u \rightsquigarrow w$ is given by p_1 and the optimal path from $w \rightsquigarrow v$ is given by p_2.

Counter-Example: The Longest Simple Path

The first problem has optimal substructure.

- Suppose that the shortest path $u \rightsquigarrow v$ is given by p.
- Given some intermediate vertex w, the optimal path from $u \rightsquigarrow w$ is given by p_1 and the optimal path from $w \rightsquigarrow v$ is given by p_2.
- If there were a shorter path p_1' from $u \rightsquigarrow w$ then we could replace p_1 with it and get a total path with fewer edges.

Counter-Example: The Longest Simple Path

Why does that argument reinforce the idea of optimal substructure?

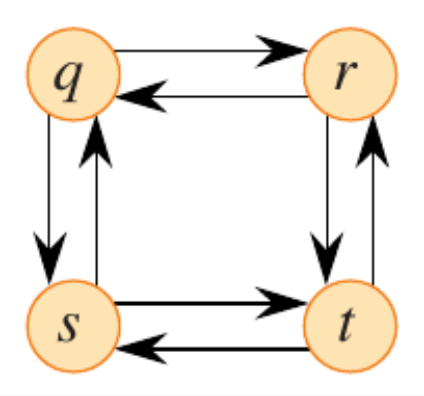
Counter-Example: The Longest Simple Path

Why does that argument reinforce the idea of optimal substructure?

By showing that the optimal solution to a subproblem is the optimal solution to the original problem.

This argument becomes clearer as we consider the longest simple path problem.

Counter-Example: The Longest Simple Path



Longest simple path problem (Cormen et al.).

Counter-Example: The Longest Simple Path

The path $q \to r \to t$ is the longest simple path from q to t.

Keep in mind that the problem is to find a simple path with the most edges. If the substructure is optimal, then the subpaths must also exhibit maximal edges.

Counter-Example: The Longest Simple Path

The subpath $q \rightsquigarrow r$ in this case is simply $q \rightarrow r$, but the longest simple path from q to r is $q \rightarrow s \rightarrow t \rightarrow r$.

Therefore, the subpath $q \rightsquigarrow r$ is not optimal.

This is a counter-example to the idea that the longest simple path problem has optimal substructure.

Counter-Example: The Longest Simple Path

The longest simple path problem does not have **independent** subproblems.

Consider a path from q to t.

This could be broken down into subproblem $q \rightsquigarrow r$ and $r \rightsquigarrow t$.

Counter-Example: The Longest Simple Path

For $q \rightsquigarrow r$, we have $q \to s \to t \to r$.

- This subproblem is dependent on s and t, so we cannot use them in the second subproblem $r \rightsquigarrow t$ without forming a path that is not simple.

Counter-Example: The Longest Simple Path

For $q \rightsquigarrow r$, we have $q \to s \to t \to r$.

- This subproblem is dependent on s and t, so we cannot use them in the second subproblem $r \rightsquigarrow t$ without forming a path that is not simple.
- Specifically, the first subproblem includes t, so the second subproblem cannot include t.

Counter-Example: The Longest Simple Path

For $q \rightsquigarrow r$, we have $q \to s \to t \to r$.

- This subproblem is dependent on s and t, so we cannot use them in the second subproblem $r \rightsquigarrow t$ without forming a path that is not simple.
- Specifically, the first subproblem includes t, so the second subproblem cannot include t.
- **However, the second subproblem MUST include t.**

Using Overlapping Subproblems

Do not confuse the idea of **overlapping subproblems** with the need for the subproblems to be **independent**.

- Subproblems are independent if they do not share resources.
- Overlapping subproblems means that a subproblem may require the result of another independent subproblem.
- This is the case in the rod cutting problem, for example.

Using Overlapping Subproblems

A desirable trait of any recursive problem is that it have a small number of unique subproblems.

The running time of such a solution is dependent on the number of subproblems, so having more of them will naturally lead to a less efficient solution.

Longest Common Subsequence

Longest Common Subsequence

A longest common subsequence (LCS) of two input sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ is a sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ such that Z is a subsequence of both X and Y and k is as large as possible.

Longest Common Subsequence

For example, given $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the LCS is $\langle B, C, A, B \rangle$.

Longest Common Subsequence

The subsequence is not necessarily consecutive!

A subsequence Z is common to a sequence X if it corresponds to a strictly increasing sequence of indices such that $x_{i_j} = z_j$.

Naive Solution

How would we solve this problem using a brute-force method?

We could generate all possible subsequences of X and Y and then compare them.

This would require $O(n2^m)$ time.

Dynamic Programming Solution

Following the four step process, we can formulate a dynamic programming solution to the LCS problem.

Step 1 is to determine the optimal substructure of the problem.

Optimal Substructure

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

- 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}.
- If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
- If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Optimal Substructure

Consider two sequences (words): **rocinante** and **canterbury**.

- The longest common subsequence is "cante".
- Since the last characters of the two original words do not match, we can remove the last character from either word and find the LCS of the two remaining words.
- This implies that we could have found the LCS of the two original words by finding the LCS of a smaller subproblem.

Optimal Substructure

What if the two words had the same last character?

Optimal Substructure

What if the two words had the same last character?

The LCS of the shorter strings is the same as the LCS of the original strings with the last character removed.

Recursive Solution

The next step is to write a recursive solution to the problem.

Given the substrucure just presented, a bottom-up approach seems intuitive.

Starting with indices $i = 0$ and $j = 0$ which indicate the length of the current strings X_i and Y_j, increase the length and compute the LCS as we go.

Recursive Solution

Define $c[i, j]$ as the LCS length of X_i and Y_j.

The goal is to compute $c[m, n]$, where m and n are the lengths of X and Y, respectively.

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

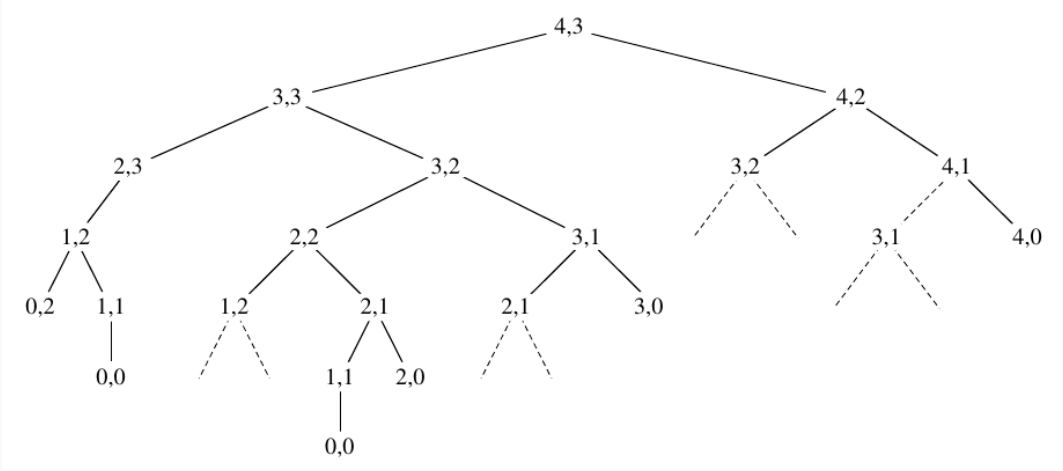
Recursive Solution

The LCS of "atom" and "ant" is "at".

The following tree shows the recursive calls to each subproblem.

A dashed line indicates that the subproblem has already been solved.

Recursive Solution



Recursive calls to the LCS problem (Cormen et al.).

Storing the Solutions

The LCS problem has $\Theta(mn)$ distinct subproblems, so storing the solutions to these subproblems will allow us to avoid redundant computation.

A dynamic programming solution goes as follows:

1. Store the lengths of the LCS of the prefixes of X and Y in a table c.
2. Additionally store the solution to the subproblems in a table b so that we can reconstruct the LCS.
3. The entries are filled in a row-major order.

Storing the Solutions

Example: lcs.py