Example II

• Unfortunately, it can be more difficult to compute the edit distance exactly. Example:

```
  A L G O R I T H M
  A L T R U I S T I C
```

Key Observation

• If we remove the last column in an optimal alignment, the remaining alignment must also be optimal

• Easy to prove by contradiction: Assume there is some better subalignment of all but the last column. Then we can just paste the last column onto this better subalignment to get a better overall alignment.

• Note: The last column can be either: 1) a blank on top aligned with a character on bottom, 2) a character on top aligned with a blank on bottom or 3) a character on top aligned with a character on bottom
To develop a DP algorithm for this problem, we first need to find a recursive definition

Assume we have a \( m \) length string \( A \) and an \( n \) length string \( B \)

Let \( E(i, j) \) be the edit distance between the first \( i \) characters of \( A \) and the first \( j \) characters of \( B \)

Then what we want to find is \( E(n, m) \)

There are three possible cases:

- **Insertion**: \( E(i, j) = 1 + E(i - 1, j) \)
- **Deletion**: \( E(i, j) = 1 + E(i, j - 1) \)
- **Substitution**: If \( a_i = b_j \), \( E(i, j) = E(i - 1, j - 1) \), else \( E(i, j) = E(i - 1, j - 1) + 1 \)

Say we want to compute \( E(i, j) \) for some \( i \) and \( j \)

Further say that the “Recursion Fairy” can tell us the solution to \( E(i', j') \), for all \( i' \leq i \), \( j' \leq j \), except for \( i' = i \) and \( j' = j \)

Q: Can we compute \( E(i, j) \) efficiently with help from the our fairy friend?

Let \( I(A[i] \neq B[j]) = 1 \) if \( A[i] \) and \( B[j] \) are different, and 0 if they are the same. Then:

\[
E(i, j) = \min \left\{ \begin{array}{ll}
E(i - 1, j) + 1, & \\
E(i, j - 1) + 1, & \\
E(i - 1, j - 1) + I(A[i] \neq B[j]) & \end{array} \right\}
\]
Base Case(s)

It's not too hard to see that:

- \( E(0, j) = j \) for all \( j \), since the \( j \) characters of \( B \) must be aligned with blanks
- Similarly, \( E(i, 0) = i \) for all \( i \)

Recursive Alg

- We now have enough info to directly create a recursive algorithm
- The run time of this recursive algorithm would be given by the following recurrence:
  \[
  T(m, 0) = T(0, n) = O(1), \quad T(m, n) = T(m, n-1) + T(m-1, n) + T
  \]
- \( T(n, n) = \Theta(1 + \sqrt{2^n}) \), which is terribly, terribly slow.

Better Idea

- We can build up a \( m \times n \) table which contains all values of \( E(i, j) \)
- We start by filling in the base cases for this table: the entries in the 0-th row and 0-th column
- To fill in any other entry, we need to know the values directly above, to the left and above and to the left.
- Thus we can fill in the table in the standard way: left to right and top down to ensure that the entries we need to fill in each cell are always available
The code

```
EditDistance(A[1,...,m],B[1,...,n]){
    for (i=1;i<=m;i++){
        Edit[i,0] = i;
    }
    for (j=1;j<=n;j++){
        Edit[0,j] = j;
    }
    for (i=1;i<=m;i++){
        for (j=1;j<=n;j++){
            if (A[i]==B[j]){
                Edit[i,j] = min(Edit[i-1,j]+1,
                                Edit[i,j-1]+1,
                                Edit[i-1,j-1]);
            }else{
                Edit[i,j] = min(Edit[i-1,j]+1,
                                Edit[i,j-1]+1,
                                Edit[i-1,j-1]+1);
            }
        }
    }
    return Edit[m,n];
}
```

Reconstructing an optimal alignment

- In this code, we do not keep info around to reconstruct the optimal alignment.
- However, it is a simple matter to keep around another array which stores, for each cell, a pointer to the cell that was used to achieve the current cell’s minimum edit distance.
- To reconstruct a solution, we then need only follow these pointers from the bottom right corner up to the top left corner.

In Class Exercise

- Create a string alignment table for the two strings “abba” and “bab”. Put “abba” at the top of the table and “bab” on the left side.
- Q1: \(i = 1, 2, \ldots, 5\) What is the \(i\)-th row of your table?
- Q6: What is the minimum edit distance and how many alignments achieve it?

Take Away

- To solve the string alignment problem, we did the following: 1) formulated the problem recursively 2) built a solution to the recurrence from the bottom up.
- Next we’ll see how a similar technique can be used to solve the matrix multiplication problem.
Matrix Chain Multiplication

Problem:

- We are given a sequence of $n$ matrices, $A_1, A_2, \ldots, A_n$, where for $i = 1, 2, \ldots, n$, matrix $A_i$ has dimension $p_{i-1}$ by $p_i$.
- We want to compute the product, $A_1A_2, \ldots, A_n$ as quickly as possible.
- In particular, we want to fully paranthesize the expression above so there are no ambiguities about the how the matrices are multiplied.
- 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.

Paranthesizing Matrices

- There are many ways to paranthesize the matrices.
- Each way gives the same output (because of associativity of matrix multiplications).
- However the way we paranthesize will effect the time to compute the output.
- Our Goal: Find a paranthesization which requires the minimal number of scalar multiplications.

Example

- In this example, it’s much better to multiply the last two matrices first (this gives us a short, narrow matrix on the right).
- Worse to multiply the first two matrices first (this gives us a short wide matrix on the left).
- In general, our goal is to find ways to always create narrow and short resulting matrices.

A Problem

Problem: There can be many ways to paranthesize. E.g.

- $(A_1(A_2(A_3A_4)))$
- $(A_1((A_2A_3)A_4))$
- $((A_1A_2)(A_3A_4))$
- $((A_1(A_2A_3))A_4)$
- $(((A_1A_2)A_3)A_4)$
A Problem

• Let $P(n)$ be the number of ways to paranthesize $n$ matrices.
  Then $P(1) = 1$
• For $n \geq 2$, we know that a fully paranthesized product is the
  product of two fully paranthesized products, and the split
  can occur anywhere from $k = 1$ to $k = n - 1$.
• Hence for $n \geq 2$:
  \[
P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)
  \]
• In the hw, you will show that the solution to this recurrence
  is $\Omega(2^n)$

The Pattern

Q: Can we develop a DP Solution to this problem?

• **Formulate the problem recursively.** Write down a formula
  for the whole problem as a simple combination of answers to
  smaller subproblems
• **Build solutions to your recurrence from the bottom up.**
  Write an algorithm that starts with the base cases of your
  recurrence and works its way up to the final solution by con-
  sidering the intermediate subproblems in the correct order.

Key Observation

• Let $A_{i..j}$ (for $i \leq j$) be the matrix that results from evaluating
  the product $A_iA_{i+1} \ldots A_j$
• Note that if $i < j$, then for some value of $k$, $i \leq k < j$, we
  must first compute $A_{i..k}$ and $A_{k+1..j}$, and then multiply them
  together to get $A_{i..j}$
• The cost of this particular parenthesization is then the cost
  of computing $A_{i..k}$ plus the cost of computing $A_{k+1..j}$ plus
  cost of multiplying $A_{i..k}$ by $A_{k+1..j}$

The Cost

• $A_{i..k}$ is a $p_i-1$ by $p_k$ matrix
• $A_{k+1..j}$ is a $p_k$ by $p_j$ matrix
• Thus multiplying $A_{i..k}$ and $A_{k+1..j}$ takes $p_{i-1}p_kp_j$ operations
Recursive Formulation

• Let \( m(i, j) \) be the minimum cost of computing \( A_{i,j} \)
• We've shown that \( m(i, j) \leq m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j \) for any \( k = i, i+1, \ldots, j-1 \)
• Further note that the optimal parenthesization must use some value of \( k = i, i+1, \ldots, j-1 \). So we need only pick the best

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

The Recursive Solution

• We now have enough information to write a recursive function to solve the problem
• The recursive solution will have runtime given by the following recurrence:
  \[
  T(1) = 1, \\
  T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n - k) + 1)
  \]
• Unfortunately, the solution to this recurrence is \( \Omega(2^n) \) (as shown on p. 346 of the text)

DP Solution

• Note that we must solve one subproblem for each choice of \( i \) and \( j \) satisfying \( 1 \leq i \leq j \leq n \)
• This is only \( \binom{n}{2} + n = \Theta(n^2) \) subproblems
• The recursive algorithm encounters each subproblem many times in the branches of the recursion tree.
• However, we can just compute these subproblems from the bottom up, storing the results in a table (this is the DP solution)
Consider the sequence of three matrices, $A_1, A_2, A_3$ whose dimensions are given by the sequence $3, 1, 2, 1, 2$.

Let’s construct the tables giving the optimal parenthesization.

The $(i, j)$ entry of the first table will give the optimal cost for computing $A_{i..j}$, the $(i, j)$ entry of the second table will give a $k$ value which achieves this optimal cost.

\[
\begin{array}{c|cccc}
1 & 0 & 6 & 5 & 10 \\
2 & - & 0 & 2 & 4 \\
3 & - & - & 0 & 4 \\
4 & - & - & - & - \\
\end{array}
\]

Thus an optimal parenthesization is $(A_1((A_2A_3)A_4))$.