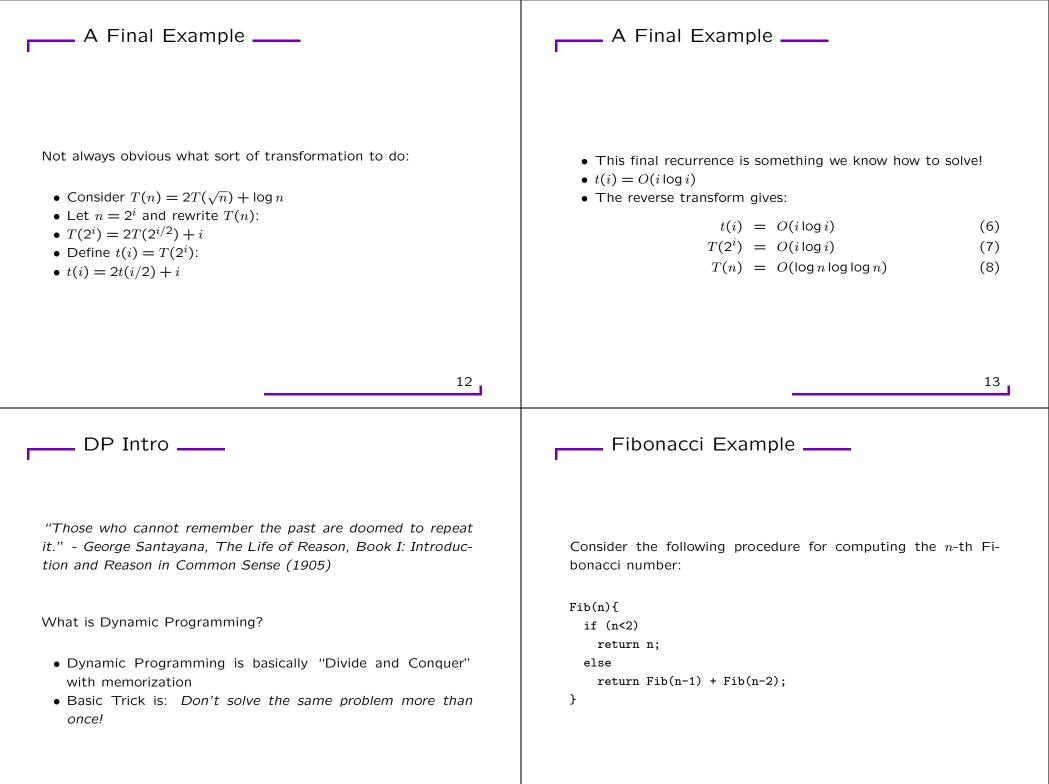


Another Example \_\_\_\_\_ Now Solve • Consider the recurrence  $T(n) = 9T(\frac{n}{3}) + kn$ , where T(1) = 1and k is some constant •  $t(0) = 1, t(i) = 9t(i-1) + k3^{i}$ • Let  $n = 3^i$  and rewrite T(n): • This is annihilated by (L - 9)(L - 3)•  $T(3^0) = 1$  and  $T(3^i) = 9T(3^{i-1}) + k3^i$ • So t(i) is of the form  $t(i) = c_1 9^i + c_2 3^i$ • Now define a sequence t as follows  $t(i) = T(3^i)$ • Then t(0) = 1,  $t(i) = 9t(i-1) + k3^i$ 8 9 Reverse Transformation \_\_\_\_\_ In Class Exercise \_\_\_\_\_ Consider the recurrence T(n) = 2T(n/4) + kn, where T(1) = 1, •  $t(i) = c_1 9^i + c_2 3^i$ and k is some constant • Recall:  $t(i) = T(3^i)$  and  $3^i = n$  $t(i) = c_1 9^i + c_2 3^i$ • Q1: What is the transformed recurrence t(i)? How do we  $T(3^i) = c_1 9^i + c_2 3^i$ rewrite n and T(n) to get this sequence? • Q2: What is the annihilator of t(i)? What is the solution for  $T(n) = c_1(3^i)^2 + c_2 3^i$ the recurrence t(i)?  $= c_1 n^2 + c_2 n$ • Q3: What is the solution for T(n)? (i.e. do the reverse  $= O(n^2)$ transformation)



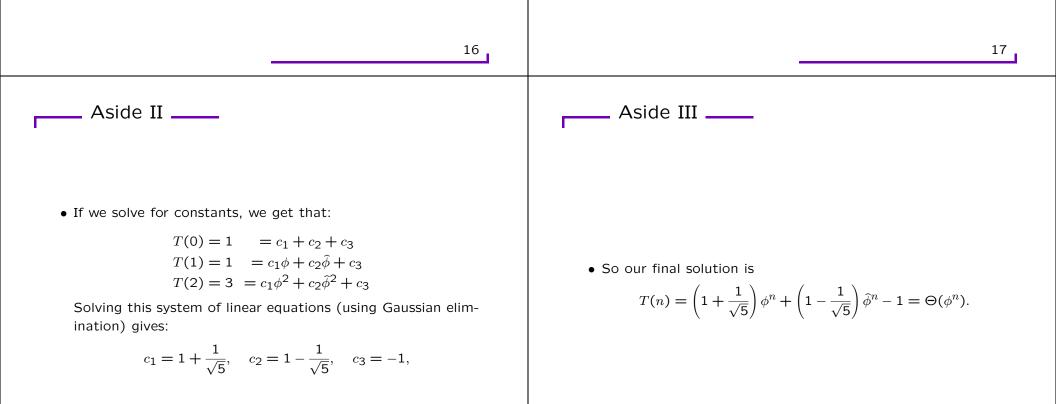
\_ Aside \_\_

• Q: What is the runtime of Fib?

Analysis \_\_\_\_\_

- A: Except for recursive calls, the entire algorithm takes a constant number of steps. If T(n) is the run time of the algorithm on input n, then we can say that:
  - T(0) = T(1) = 1, T(n) = T(n-2) + T(n-1) + 1
- It's easy to show by induction that  $T(n) = 2F_{n+1} 1$ . This is very bad!

- Q: How can we solve T(n) exactly?
- A: We solved this recurrence using annihilaotrs in the last lecture to get  $T(n) = c_1 \phi^n + c_2 \hat{\phi}^n + c_3 1^n$  where  $\phi = \frac{1+\sqrt{5}}{2}$  and  $\hat{\phi} = \frac{1-\sqrt{5}}{2}$ .



### The Problem \_\_\_\_\_

\_\_\_ DP-Fib \_\_\_\_

- The reason Fib is so slow is that it computes the same Fibonacci numbers over and over
- In general, there are  $F_{k-1}$  recursive calls to Fib(n-k)
- We can greatly speed up the algorithm by writing down the results of the recursive calls and looking them up if needed

DP-Fib(n){ if (n<2) return n; else{ if (F[n] is undefined){ F[n] = DP-Fib(n-1) + DP-Fib(n-2);} return F[n]; }}

20 21 Analysis \_\_\_\_\_ \_ Take Away \_\_\_\_\_ Dynamic Programming is different than Divide and Conquer in the following way: • "Divide and Conquer" divides problem into independent sub-• For every value of x between 1 and n, DP-Fib(x) is called problems, solves the subproblems recursively and then comexactly one time. bines solutions to solve original problem • Each call does constant work • Dynamic Programming is used when the subproblems are not • Thus runtime of DP-Fib(n) is  $\Theta(n)$  - a huge savings independent, i.e. the subproblems share subsubproblems • For these kinds of problems, divide and conquer does more work than necessary • Dynamic Programming solves each subproblem once only and saves the answer in a table for future reference

# The Pattern Edit Distance • Formulate the problem recursively.. Write down a formula for the whole problem as a simple combination of answers to smaller subproblems • The *edit distance* between two words is the minimum number • Build solutions to your recurrence from the bottom up. of letter insertions, letter deletions, and letter substitutions Write an algorithm that starts with the base cases of your required to transform one word into another. For example, recurrence and works its way up to the final solution by conthe edit distance between FOOD and MONEY is at most four: sidering the intermediate subproblems in the correct order. $FOOD \rightarrow MOOD \rightarrow MON_{A}D \rightarrow MONED \rightarrow MONEY$ Note: Dynamic Programs store the results of intermediate subproblems. This is frequently *but not always* done with some type of table. 24 25 \_ Example \_\_\_\_\_ String Alignment \_\_\_\_\_ Better way to display this process: • String Alignment for "FOOD" and "MONEY": Place the words one above the other in a table. • Put a gap in the first word for every insertion and a gap in FOO D MONEY the second word for every deletion • Columns with two different characters correspond to substi-• It's not too hard to see that we can't do better than four for tutions the edit distance between "Food" and "Money" • Then the number of editing steps is just the number of columns that don't contain the same character twice

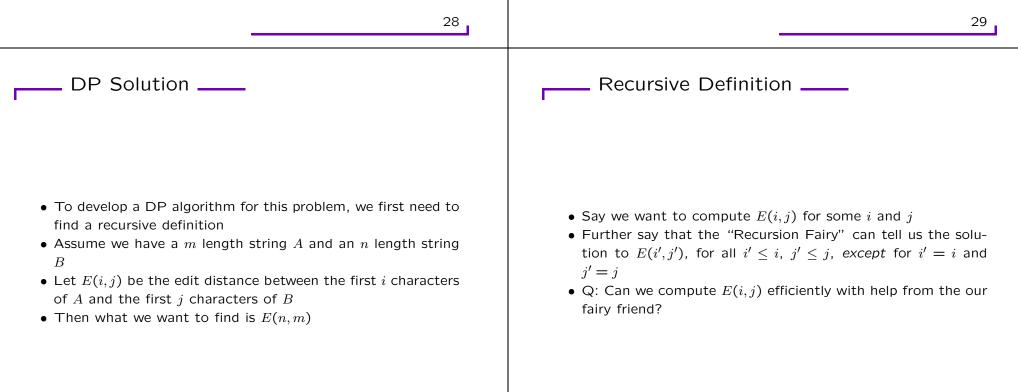
### Example II

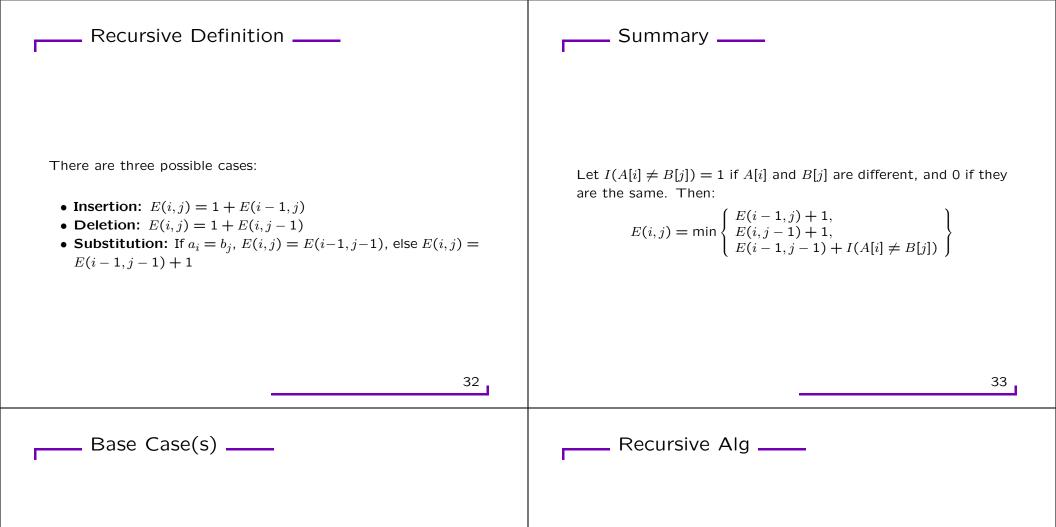
• Unfortunately, it can be more difficult to compute the edit distance exactly. Example:

ALGORITHM AL TRUISTIC

## 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





- 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:

$$T(m, 0) = T(0, n) = O(1)$$

T(m,n) = T(m,n-1) + T(m-1,n) + T(n-1,m-1) + O(1)

• Solution:  $T(n,n) = \Theta(1 + \sqrt{2}^n)$ , which is terribly, terribly slow.

It's not too hard to see that:

aligned with blanks

• Similarly, E(i,0) = i for all i

• E(0,j) = j for all j, since the j characters of B must be

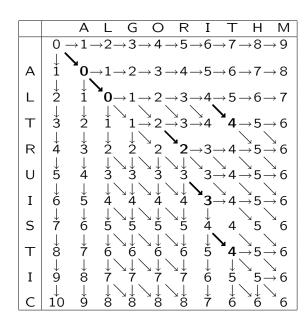
## Example Table \_\_\_\_\_

### Better Idea \_\_\_\_\_

- $\bullet$  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

- Bold numbers indicate places where characters in the strings are equal
- Arrows represent predecessors that define each entry: horizontal arrow is deletion, vertical is insertion and diagonal is substitution.
- Bold diagonal arrows are "free" substitutions of a letter for itself
- Any path of arrows from the top left to the bottom right corner gives an optimal alignment (there are three paths in this example table, so there are three optimal edit sequences).

36



Analysis \_\_\_\_\_

- Let n be the length of the first string and m the length of the second string
- Then there are ⊖(nm) entries in the table, and it takes ⊖(1) time to fill each entry
- This implies that the run time of the algorithm is  $\Theta(nm)$
- Q: Can you find a faster algorithm?

37