Edit Distance

Prepared by Michael Gabilondo UCF, COP 3503 Recitation November 20, 2013

Introduction

- We want to know how "different" two strings are
 - What is the difference between "kitten" and "kiten"? Answer: One "t".
 - "kiten" is closer to "kitten" than to "sitting".
- Edit distance is a distance measure between two strings – A number that represents how dissimilar (far apart) two strings are
- If this number is small, then the two strings are similar

Application: Approximate Substring Matching

- Find approximate matches of a small string in a long text that can be segmented into substrings which we compare with the small string
- Given a text document, a spell-checker could segment the text into individual words, and compare each individual word to the words in a dictionary
 - The spelling suggestions for a word in the text are words in the dictionary with the smallest distance to that word
- Molecular Biology Find approximate occurrences of a smaller protein/DNA string in a longer protein/DNA string – Find proteins/genes with shared properties
- We can compare two long strings, but the complexity of the algorithm to compute the distance between s1 and s2 is O(|s1|*| s2|) in both time and space

Problem

• Given two strings, a **source** and a **target**, find the minimum number of single-character edits to change source into target

• The answer to the above problem is a number that represents the distance between the target and the source

- What are the possible single-character edits?
 - Substitution Change a character of source
 - (change it to a character of target) ("hello" \rightarrow "yello")
 - Insertion Insert a character into source
 - (insert a character that is in target) ("yello" \rightarrow "yellow")
 - **Deletion** Delete a character from source
 - (delete a character that is not in target) ("nice" \rightarrow "ice")

- Source = hello Target = help
- Look at the last character of source
 - It doesn't match the last character of target, so we have to use one of the 3 edit operations
- Option 1: Substitution/Match. Change the "o" in source to a "p".
 - Source = hellp Target = help
 - Now they match at the cost of 1 edit operation
 - The problem has been reduced to finding the edit distance between all but the last characters of source and target, i.e., hell and hel
 - If this operation was the right thing to do, then the edit distance is just 1 + the edit distance between hell and hel
 - What if the last characters already happened to match? Then we still reduce the problem in the same way, but it costs 0, since no edit. This is called a *match*. Above, we did a *substitution*.

- Source = hello Target = help
- Option 2: Deletion. Delete the "o" from source.
 - Source = hell Target = help
 - (This makes sense if source has too many characters; it eventually needs to have the same number of characters as target.)
 - We have also reduced the problem at the cost of 1 edit operation: source has one fewer characters.
 We didn't reduce the size of target this time.
 - If this operation was the right thing to do, then the edit distance is just 1 + the edit distance between hell and help

- Source = hello Target = help
- Option 3: Insertion. Insert "p" to the end of source.
 - Source = hellop Target = help
 - (This makes sense if source has too few characters. But now we made it too long, so we're going to need a deletion in the future)
 - We made the last character of source match the last character of target
 - That means we only need to compare **hello** and **hel**
 - Note that we reduced the size of target, but not source at the cost of 1 edit operation
 - If this operation was the right thing to do, then the edit distance is just 1 + the edit distance between hello and hello

- We have 3 ways to recursively reduce the problem into a simpler problem
- Which one is the right way?
 - It is the way that has the minimum value
- Recursively try all three ways and return the minimum

Base Cases

- When do we stop the recursion? Consider examples.
- Source = (empty) Target = (empty)
 - Edit distance = 0
- Source = (empty) Target = "arbitrary"
 - Edit distance = ["arbitrary"] ; we need 9 insertions
- Source = "arbitrary" Target = (empty)
 - Edit distance = |"arbitrary"|; we need 9 deletions

Base case

- If min(|source|, |target|) == 0,
 - then return max(|source|, |target|)

Notation

- Let D[i, j] be the edit distance between the first i characters of source and the first j characters of target
- We use 1-based indexing and D[0,j] and D[i,0] correspond to source and target being empty strings, respectively
 - Base cases: D[i, 0] = i and D[0, j] = j for all i,j
- If we do a Substitution/Match...
 - If Source[i] != Target[i], then D[i,j] = 1 + D[i-1, j-1]
 - If Source[i] == Target[i], then D[i,j] = 0 + D[i-1, j-1]
- If we do a deletion...
 - then D[i,j] = 1 + D[i-1, j]
- If we do an insertion...
 - then D[i,j] = 1 + D[i, j-1]

Recursive Solution – Pseudocode

- Function D[i, j]
 - If min(i, j) == 0, then return max(i, j)
 - If source[i] != target[i]
 - SubMatchCost \leftarrow 1 + D[i-1, j-1]
 - Else
 - SubMatchCost ← 0 + D[i-1, j-1]
 - DeletionCost \leftarrow 1 + D[i-1, j]
 - InsertionCost \leftarrow 1 + D[i, j-1]
 - Return min(SubMatchCost, DeletionCost, InsertionCost)

Time complexity

- The recurrence relation for the number of operations
 is
 - T(n, m) = T(n-1, m-1) + T(n-1, m) + T(n, m-1) + 1
 - T(0, m) = T(n, 0) = 1
- This is exponential in n and m...
- The reason it is so slow is that the same D[i,j] (and the same T(i,j)) are being computed repeatedly

Idea to improve time complexity

- Consider: How many **unique** calls D[i,j] are there? We have
 - D[0, 0], D[1, 0], D[2, 0], ..., D[n, 0]
 - D[0, 1], D[1, 1], D[2, 1], ..., D[n, 1]

. . . .

- D[0, m], D[1, m], D[2, m], ..., T(n, m]

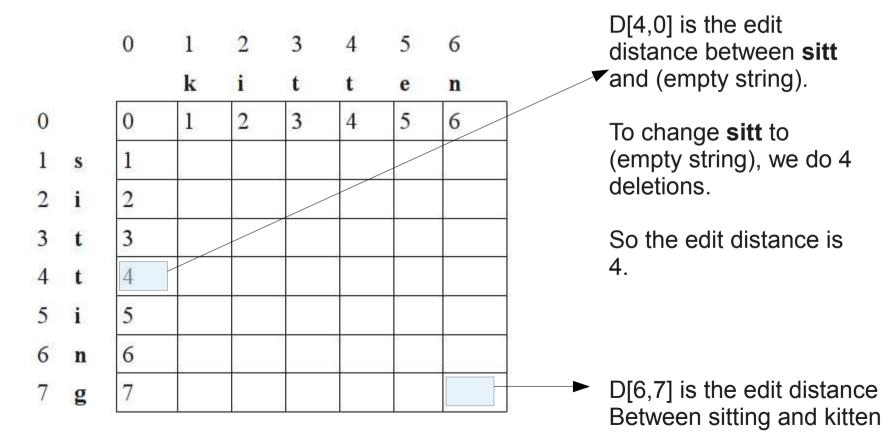
- There are (n+1)*(m+1) unique calls T(i,j)
- Idea: Build a (n+1)*(m+1) integer array, M, where M[i,j] stores the result of D[i,j]
 - Modify our recursive function D[i,j] so that the first thing it does is check if the result is already there in M[i,j] – If so, just return M[i,j]
 - If it is not there, compute it recursively. But before returning, store the result in M[i,j]

Top-down vs. Bottom-up

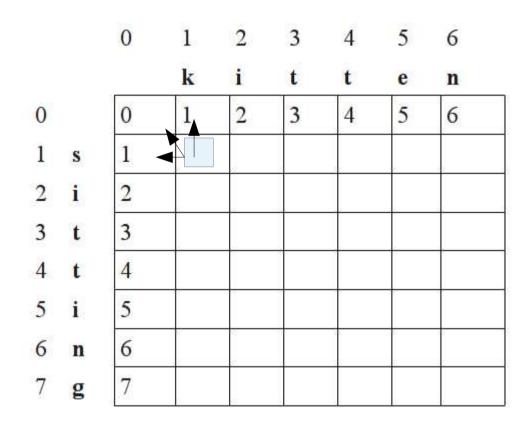
- The use of the table M to store the results of the recursive calls is called **memoization** and it is a form of dynamic programming
- This reduces the time complexity to O(n*m) and our space complexity is O(n*m) for the table M
- This is a "top-down" solution we start with the big problem and recursively break it down into smaller problems
- However, notice that the first problems to be solved are the smallest subproblems, i.e., the base cases, and then the next smallest ones, etc.
- Using that observation, we can construct a solution where we *iteratively* first solve the smallest sub-problems, then the next smallest ones, etc. – this is called a "bottom-up" solution
- The "bottom-up" solution will fill out the table M with the smallest values first, just as the "top-down" solution does. (We'll actually call it D, not M.)
- Note that the "bottom-up" solution is **not** called memoization

Bottom-up: D as a table instead of a function

- Source = sitting (7 chars) Target = kitten (6 chars)
- We need an (7+1) by (6+1) table, D.
- We can fill out the base cases right away. D[i, 0] = i and D[0, j] = j for all i,j



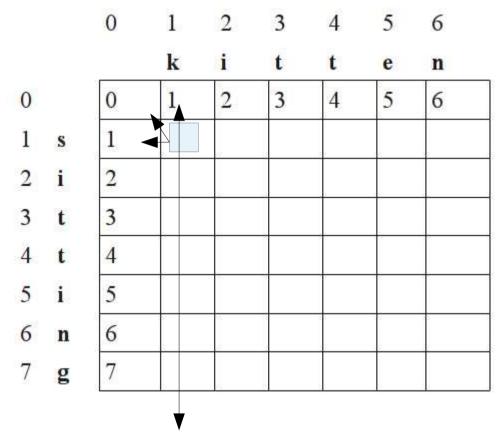
Use the recursive solution to fill out the table



• We will fill it out row-by-row, starting from the top row, and each row from left-to-right

- Fill out the table iteratively until we reach D[7,6], the solution.
- To compute D[i,j], we use the recursive solution – but instead of making recursive calls, we look up the (already computed) values in the table
- In what order should we visit the cells? Any order that allows guarantees we have already computed all the values we need to compute D[i,j]
- What values does D[i,j] need? It needs D[i-1, j-1], D[i-1, j] and D[i, j-1]
- With respect to D[i, j], these are the cells: left&up, up, left

Use the recursive solution to fill out the table

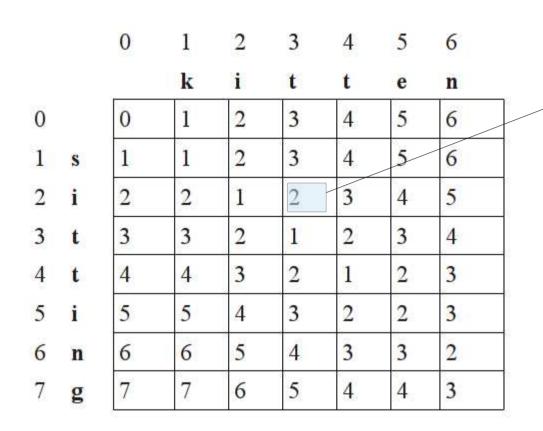


D[i,j] = min(1 + 0, 1 + 1, 1 + 1) = 1.

We picked the first one, match/sub. That means we changed the "s" to a "k"

- What is the recursive solution D[i,j]?
- Remember, it depends on if the last characters of the two prefixes of source and target are equal
- In the case of D[1,1] they are not, so we have
- D[i,j] = min(1 + D[i-1, j-1], Match/sub 1 + D[i-1, j], Deletion 1+ D[i, j-1]) Insertion
- If those last characters were equal, then we would be adding a 0 instead of a 1 in the first argument of min (in bold)

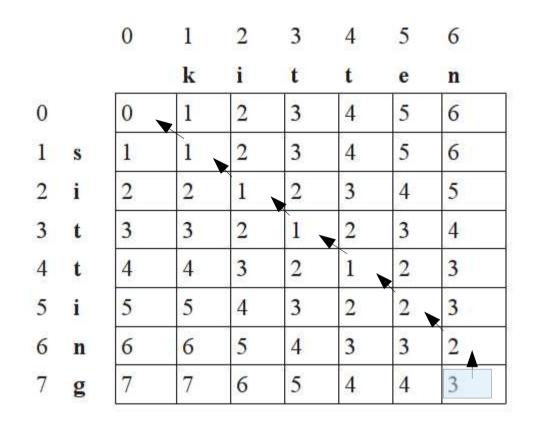
The complete table – Another example



That means to change **si** to **kit**, the best thing to do is insert **t** to the end of **si**

Because t != i

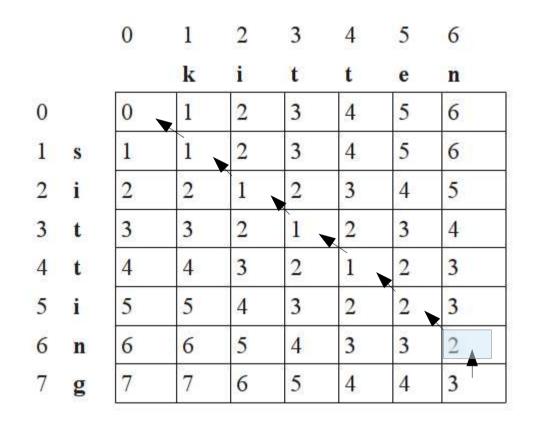
- D[2,3] = 2, the edit distance between si and kit
- How did we get this?
 - Look left, D[i, j-1] = 1.
 - 1 + D[i, j-1] = 2 is the total number of edits if we insert t to the end of si
 - Look up, D[i-1, j] = 3.
 - 1 + D[i-1, j] = 4 is the total number of edits if we delete i from the end of si
 - Look left&up, D[i-1,j-1] = 2.
 1 + D[i-1, j-1] = 3 is the total number of edits if we change the i at the end of si to a t



- Start from lower-right cell, D[7,6]
- What edit did we make to get there?
- We added 1 + 2 = 3, and so we came from one cell **above**.
- That means we deleted g from sitting
- So we have sittin and kitten
- It cost us 1 edit

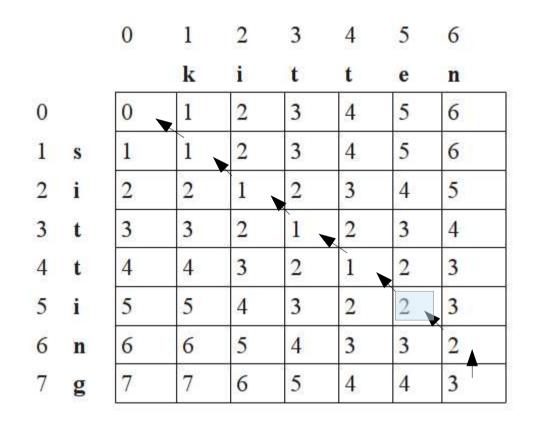
So far...

S I T T I N G D S I T T I N



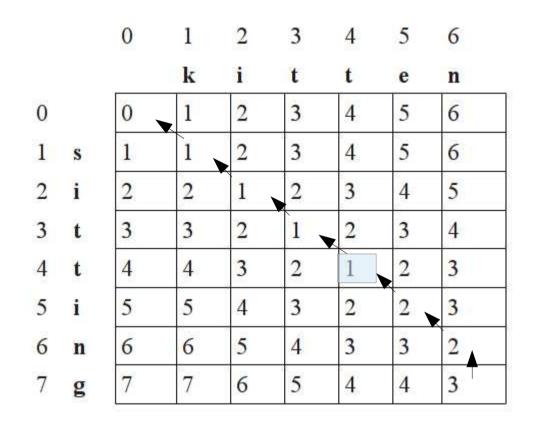
- How did we get to D[6,6]? The last characters match, so we added 0 + 2 = 2, and we came from up&left.
- We didn't make an edit, but now the problem is reduced to sitti and kitte in cell D[5,5]

So far...



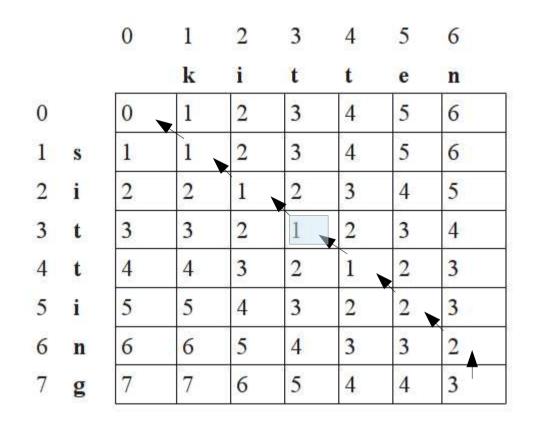
- We are at D[5,5] with sitti and kitte
- The last characters don't match, and we added 1 + 1 = 2, and we came from up&left
- That means we changed the i in sitti to an e
- It cost us 1 edit
- We have reduced the problem to sitt and kitt

So far...



- We are at D[4,4] with sitt and kitt
- The last characters match, and we added 0 + 1 = 1, and we came from up&left
- That means we matched the the last characters and we didn't make an edit
- We have reduced the problem to sit and kit

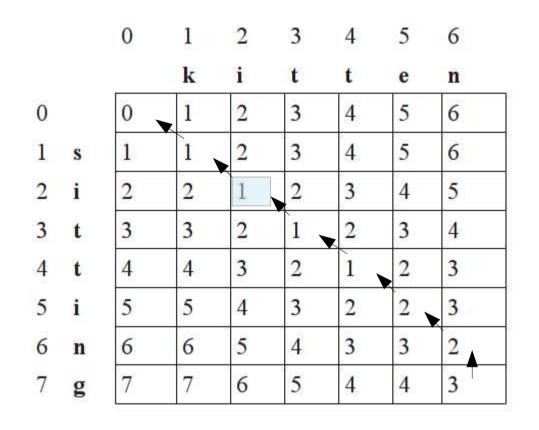
So far...



- We are at D[3,3] with sit and kit
- The last characters match, and we added 0 + 1 = 1, and we came from up&left
- That means we matched the the last characters and we didn't make an edit
- We have reduced the problem to si and ki

So far...

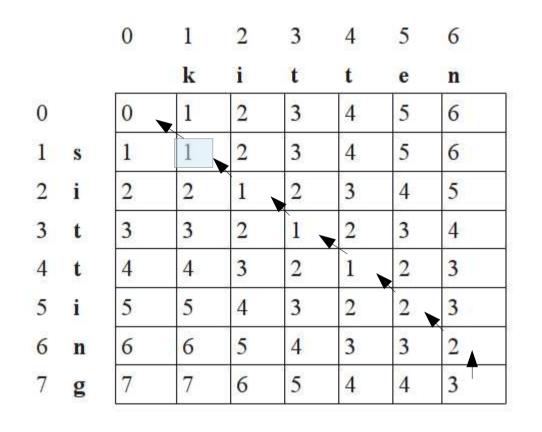
S I T T I N G M M S M D S I T T E N



- We are at D[2,2] with si and ki
- The last characters match, and we added 0 + 1 = 1, and we came from up&left
- That means we matched the the last characters and we didn't make an edit
- We have reduced the problem to s and k

So far...

S I T T I N G M M M S M D S I T T E N



• We are at D[1,1] with **s** and **k**

- The last characters don't match, and we added 1 + 0 = 1, and we came from up&left
- That means we changed the s to a k
- It cost us 1 edit operation
- We have reduced the problem to (empty string) and (empty string) at D[0,0] and we are done

So far...

S I T T I N G S M M M S M D K I T T E N