Andrew DeMaria~MariaDeslis~Tri Nguyen

CSCI406-FALL2012

TSP PROJECT

Professor Mehta

Theory: Dynamic Algorithm

Recursive Algorithm

The first step is to determine the recursive relationship for finding the next largest palindromic subsequence from a given string of characters as well as the base case for a palindromic subsequence.

There are two main base cases. The first is when the entire string (palindrome) is one character. If the string is one character, that character is returned. The second is when the string has two characters, but both characters are the same (i.e. aa). If this is the case, then the entire string is also returned. Otherwise, the recursive relationship can be defined as follows: If the outer two characters are equivalent then the length of the longest palindromic subsequence is equal to two plus the longest palindromic subsequence of the inner string. If the outer two characters are not equivalent, then the palindromic subsequence will be the largest of either the string with the leftmost character removed or the rightmost character removed. This ensures that all possibilities are covered.

The issue with the above recursive function is that repeated calls to find the largest palindromic subsequence will occur. To mitigate this, a lookup table is created from the base case up. The lookup table has cells consisting of a simple struct that holds the current length of the maximum palindrome within the start index and end index (row, column). It also holds a reference to the source of the maximum palindrome subsequence. These references can either be: LEFT, DOWN, DIAGONAL and BASECASE. Once all cells are calculated, the largest subsequence of the entire string is then found by starting at the upper right cell in the table and tracing back the decisions made until a base case is found.

One key note is that the second base case ( two characters that match ) is slightly altered to simplify the design. This base case is changed be of case DIAGONAL and will now reference an invalid cell on the opposite end of the table. These cells are now considered to be a base case of zero character length and in the algorithm this will have the same overall result of returning the two characters.

Dynamic Algorithm Pseudo Code

INIT TABLE:

Table dimensions = length of string x length of string (rows x columns)

Initialize the table with references to empty strings with a tag of BASECASE.

Initialize the main diagonal with a length of one, a single character, and keep the tag BASECASE.

row - start character

column - end character

FOR each diagonal starting at the main diagonal and propagating to the upper right:

FOR each cell In the diagonal:

Look at whether the two characters represented by the indices are equivalent

IF not equivalent

compare the cell to left and to the cell underneath

IF the left is larger in length

Store a reference to LEFT and its size

ELSE

Store a reference to DOWN and its size

IF they are equivalent

Store a reference to DIAGONAL (row + 1, col - 1) and its size

Traceback Algorithm Pseudo Code

String traceback( cell ):

Start at the top right of the table.

IF the cell reference is BASECASE and on the *main* diagonal:

RETURN the character at that index

IF the cell reference is BASECASE and not on the *main* diagonal:

RETURN null

IF the cell is of reference type diagonal:

RETURN the character of the string at index row plus traceback( cell at diagonal) plus the character of the string at index col.

IF the cell is of reference type left:

RETURN traceback( cell at the left)

IF the cell is of reference type down:

RETURN traceback( cell at the right)

Theory: Complexity

The bulk of the work done is inside the nested for-loops. Since the comparisons for each cell is done in constant time, we can formulate the nested for-loops as the following:



Therefore, the time complexity of the algorithm is .

Implementation

See attached.

Example: "ATTCA"

The code would will initialize a table as follows (notice the main diagonal):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | END | | | | |
| A | T | T | C | A  **KEY**  **BC**: Base Case  **L**: Left  **DG**: Diagonal  **D**: Down  **(referenced cell, size)** |
| START | A | BC, 1 | BC, 0 | BC, 0 | BC, 0 | BC, 0 |
| T | BC, 0 | BC, 1 | BC, 0 | BC, 0 | BC, 0 |
| T | BC, 0 | BC, 0 | BC, 1 | BC, 0 | BC, 0 |
| C | BC, 0 | BC, 0 | BC, 0 | BC, 1 | BC, 0 |
| A | BC, 0 | BC, 0 | BC, 0 | BC, 0 | BC, 1 |

We first start at the first diagonal, which is colored blue and on the first cell which is darkened. The start character, "A" does not match the end character "T", so we then consider the cell to the left and the cell below. These have the same max palindrome size, 1 so we default to choosing the cell below. This cell now has a reference to the cell below it and the size of this cell is copied from the cell below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | END | | | | |
| A | T | T | C | A |
| START | A | BC, 1 | D, 1 | BC, 0 | BC, 0 | BC, 0 |
| T | BC, 0 | BC, 1 | BC, 0 | BC, 0 | BC, 0 |
| T | BC, 0 | BC, 0 | BC, 1 | BC, 0 | BC, 0 |
| C | BC, 0 | BC, 0 | BC, 0 | BC, 1 | BC, 0 |
| A | BC, 0 | BC, 0 | BC, 0 | BC, 0 | BC, 1 |

We now move onto the next cell. Here we are looking at the first T and the second T. Since they match, we tag this cell as DIAGONAL and store the length as 2 plus the length of the cell diagonal to it which is 0 (colored red). The remaining two cells are the same as the first so those steps are omitted here and we move on to the next diagonal.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | END | | | | |
| A | T | T | C | A |
| START | A | BC, 1 | D, 1 | BC, 0 | BC, 0 | BC, 0 |
| T | BC, 0 | BC, 1 | DG, 2 | BC, 0 | BC, 0 |
| T | BC, 0 | BC, 0 | BC, 1 | D, 1 | BC, 0 |
| C | BC, 0 | BC, 0 | BC, 0 | BC, 1 | D, 1 |
| A | BC, 0 | BC, 0 | BC, 0 | BC, 0 | BC, 1 |

In the cell selected, the start character, "A", does not match the end character, the second "T", so we choose the max of the cell to the left or the cell below. This is the cell below so we store DOWN and length 2. Again the next two cells are done in a similar fashion so we skip those.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | END | | | | |
| A | T | T | C | A |
| START | A | BC, 1 | D, 1 | D, 2 | BC, 0 | BC, 0 |
| T | BC, 0 | BC, 1 | DG, 2 | L, 2 | BC, 0 |
| T | BC, 0 | BC, 0 | BC, 1 | D, 1 | D, 1 |
| C | BC, 0 | BC, 0 | BC, 0 | BC, 1 | D, 1 |
| A | BC, 0 | BC, 0 | BC, 0 | BC, 0 | BC, 1 |

The next two cells are also a simple pick from either the left or below cells:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | END | | | | |
| A | T | T | C | A |
| START | A | BC, 1 | D, 1 | D, 2 | D, 2 | BC, 0 |
| T | BC, 0 | BC, 1 | DG, 2 | L, 2 | L, 2 |
| T | BC, 0 | BC, 0 | BC, 1 | D, 1 | D, 1 |
| C | BC, 0 | BC, 0 | BC, 0 | BC, 1 | D, 1 |
| A | BC, 0 | BC, 0 | BC, 0 | BC, 0 | BC, 1 |

For the final step to the table, we see that the characters match ( the first and last A's ) so this cells reference is DIAGONAL and its value is 2 plus the value in the cell diagonal which is 2 with a result of 4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | END | | | | |
| A | T | T | C | A |
| START | A | BC, 1 | D, 1 | D, 2 | D, 2 | DG, 4 |
| T | BC, 0 | BC, 1 | DG, 2 | L, 2 | L, 2 |
| T | BC, 0 | BC, 0 | BC, 1 | D, 1 | D, 1 |
| C | BC, 0 | BC, 0 | BC, 0 | BC, 1 | D, 1 |
| A | BC, 0 | BC, 0 | BC, 0 | BC, 0 | BC, 1 |

To then illustrate the traceback algorithm I have highlighted the route. The progression of events is as follows:

1. DG: return the string "A" + stringFromDiagonal + "A"
2. L: return the stringFromLeft
3. DG: return the string "T" + stringFromDiagonal2 + "T"
4. BC (not on main diagonal): return ""
5. Compiled result: "ATTA"

Group Effort

Group scores as decided and discussed by group as a whole:

* + Andrew DeMaria : 33.33%
  + Maria Deslis : 33.33%
  + Tri Nguyen : 33.33%