I.T.U.
Faculty of Computer and Informatics
Computer Engineering

Lesson name: Analysis of Algorithms

Lesson Code: BLG 372E

Name Surname: Abdullah AYDEĞER

Number: 040090533

Instructor's Name: Zehra ÇATALTEPE

Assistant's Name: A.Aycan ATAK

Due Date: 04.05.2012

Table of Contents

Introduction

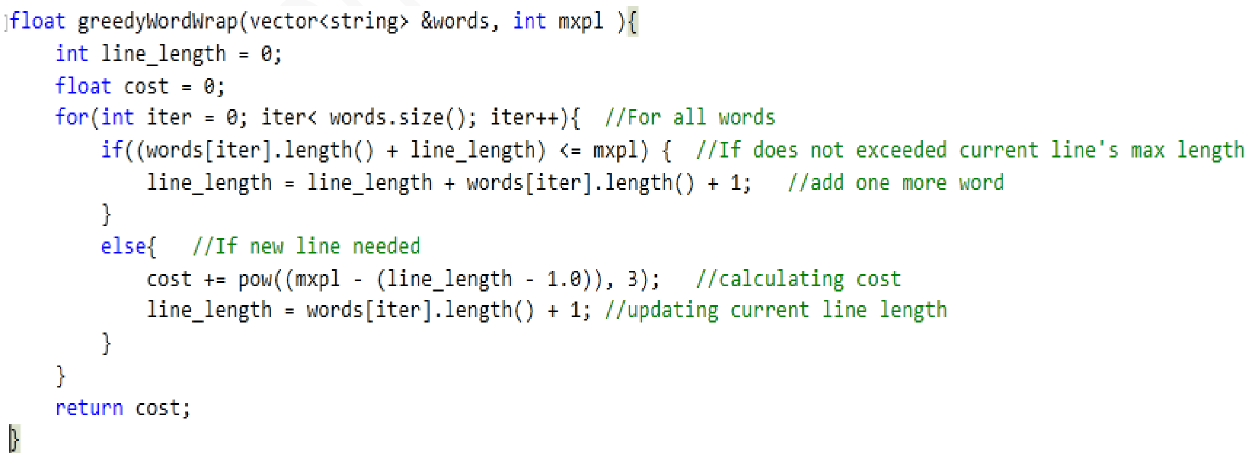
Word wrapping means setting the words to the lines by using some rules. Today, a lot of editors are using some complex word wrapping algorithms. In this project, I've implemented 2 word wrapping algorithms in C++. First algorithm is greedy and does not calculate optimal solution but faster. Second one calculates optimal solution but slower. *Optimal solution: All lines trying to get minumum empty space. I've calculated costs by formula: **costOfCurrentLine = (MaxCharPerLine – (# of words on line) – SumOfWordsOnLine)**^{3}. According to this formula, I've searched for minumum total cost.

My Algorithms and Their Time Complexities

Greedy Solution

My algorithm for greedy solution:

- For all words in the given text, calculating cost as seen in figure 1.



```
float greedyWordWrap(vector<string> &words, int mxpl ){
    int line_length = 0;
    float cost = 0;
    for(int iter = 0; iter< words.size(); iter++){  //For all words
        if((words[iter].length() + line_length) <= mxpl) {  //If does not exceeded current line's max length
            line_length = line_length + words[iter].length() + 1;   //add one more word
        }
        else{   //If new line needed
            cost += pow((mxpl - (line_length - 1.0)), 3);   //calculating cost
            line_length = words[iter].length() + 1; //updating current line length
        }
    }
    return cost;
}
```

Figure 1. Greedy solution

Time complexity of greedy algorithm: **O(n)** n: total number of words in text

Dynamic Programming

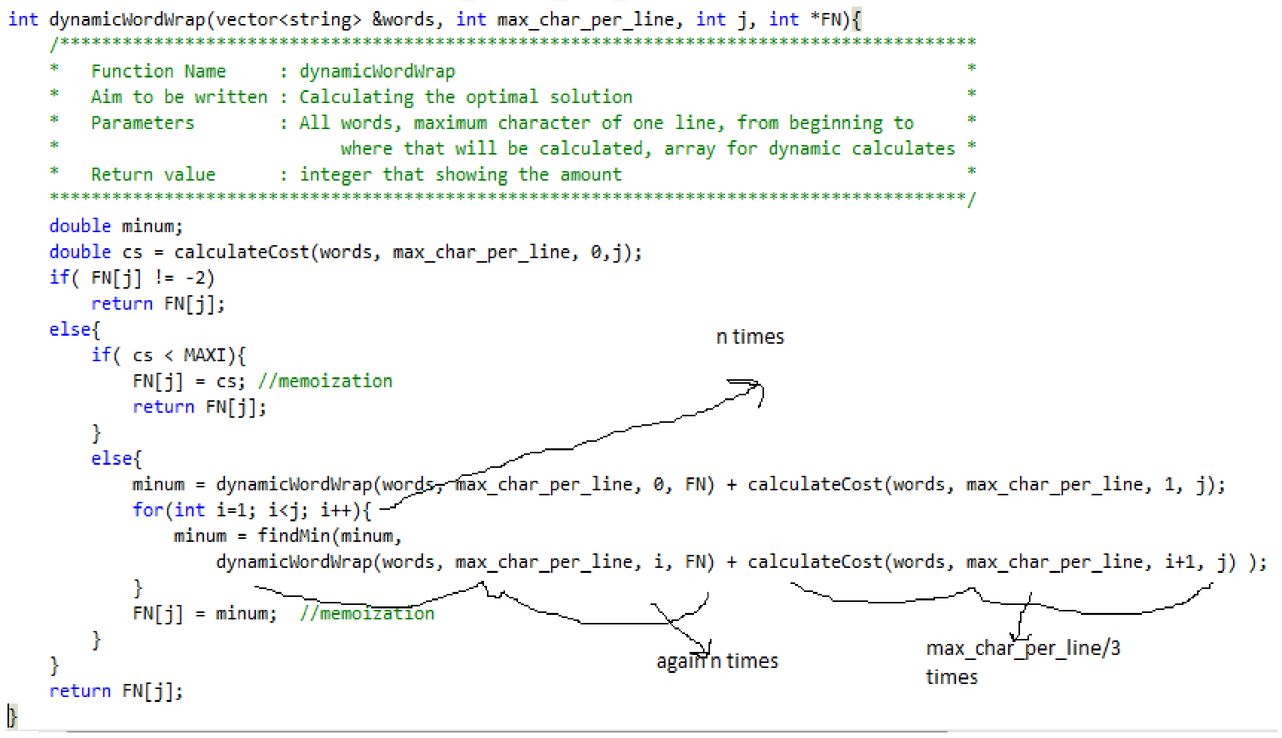
My algorithm for word wrapping problem by using dynamic programming:

- By giving ‘j’ value(for first calling, j must equal to the amount of all words), if FN[j] is calculated before, then return this value back. If not, if the given interval from first word to j.th word, does it fit in the one line? If yes, then FN[j] equals cost of this current line. If no, then calculating the minimum of FN[i] + cost(i+1, j) from i = 0 to j value. As seen in the below figure 2.

$$f(j) = \begin{cases} c(1,j) & \text{if } c(1,j) < \infty, \\ \min\limits_{1 \le k < j} \big(f(k) + c(k+1,j)\big) & \text{if } c(1,j) = \infty. \end{cases}$$

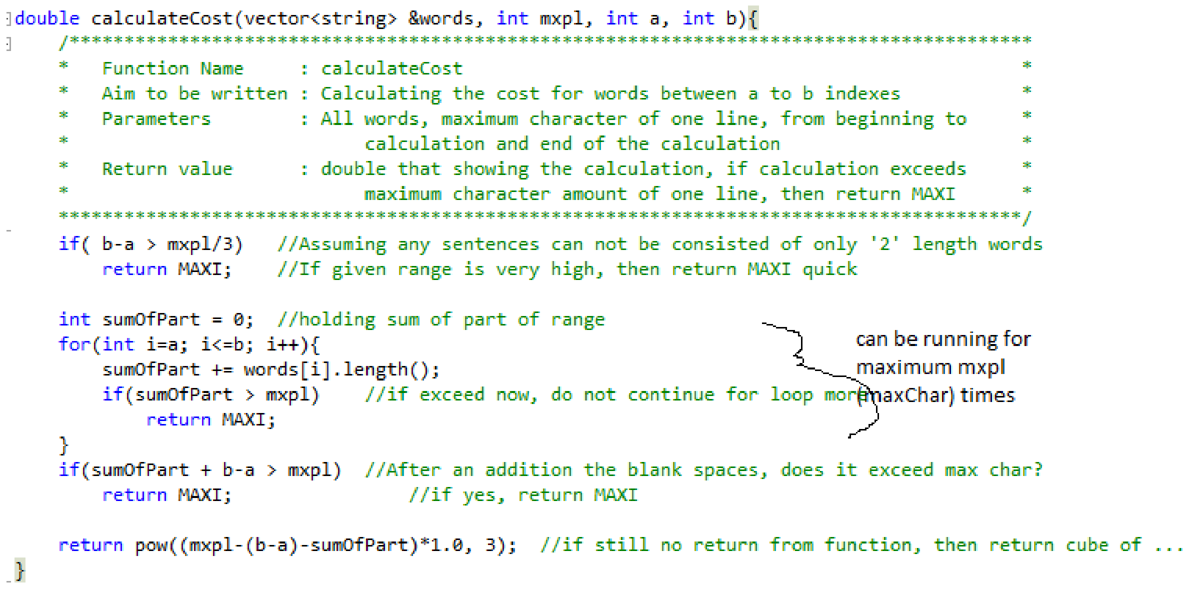
Figure 2. Dynamic solution algorithm. Photo taken from http://en.wikipedia.org/wiki/Word_wrap

- According to the algorithm, my codes are below.



```
int dynamicWordWrap(vector<string> &words, int max_char_per_line, int j, int *FN){
    /*****************************************************************************************
    *   Function Name     : dynamicWordWrap                                                 *
    *   Aim to be written : Calculating the optimal solution                                *
    *   Parameters        : All words, maximum character of one line, from beginning to     *
    *                            where that will be calculated, array for dynamic calculates *
    *   Return value      : integer that showing the amount                                 *
    *****************************************************************************************/
    double minum;
    double cs = calculateCost(words, max_char_per_line, 0,j);
    if( FN[j] != -2)
        return FN[j];
    else{
        if( cs < MAXI){
            FN[j] = cs; //memoization
            return FN[j];
        }
        else{
            minum = dynamicWordWrap(words, max_char_per_line, 0, FN) + calculateCost(words, max_char_per_line, 1, j);
            for(int i=1; i<j; i++){
                minum = findMin(minum,
                    dynamicWordWrap(words, max_char_per_line, i, FN) + calculateCost(words, max_char_per_line, i+1, j) );
            }
            FN[j] = minum;   //memoization
        }
    }
    return FN[j];
}
```

Figure 3. Dynamic solution in C++



```
double calculateCost(vector<string> &words, int mxpl, int a, int b){
    /*****************************************************************************************
    *   Function Name     : calculateCost                                                     *
    *   Aim to be written : Calculating the cost for words between a to b indexes             *
    *   Parameters        : All words, maximum character of one line, from beginning to       *
    *                           calculation and end of the calculation                        *
    *   Return value      : double that showing the calculation, if calculation exceeds       *
    *                           maximum character amount of one line, then return MAXI        *
    *****************************************************************************************/
    if( b-a > mxpl/3)   //Assuming any sentences can not be consisted of only '2' length words
        return MAXI;    //If given range is very high, then return MAXI quick

    int sumOfPart = 0;  //holding sum of part of range
    for(int i=a; i<=b; i++){
        sumOfPart += words[i].length();
        if(sumOfPart > mxpl)    //if exceed now, do not continue for loop more
            return MAXI;
    }
    if(sumOfPart + b-a > mxpl)  //After an addition the blank spaces, does it exceed max char?
        return MAXI;                //if yes, return MAXI

    return pow((mxpl-(b-a)-sumOfPart)*1.0, 3);  //if still no return from function, then return cube of ...
}
```

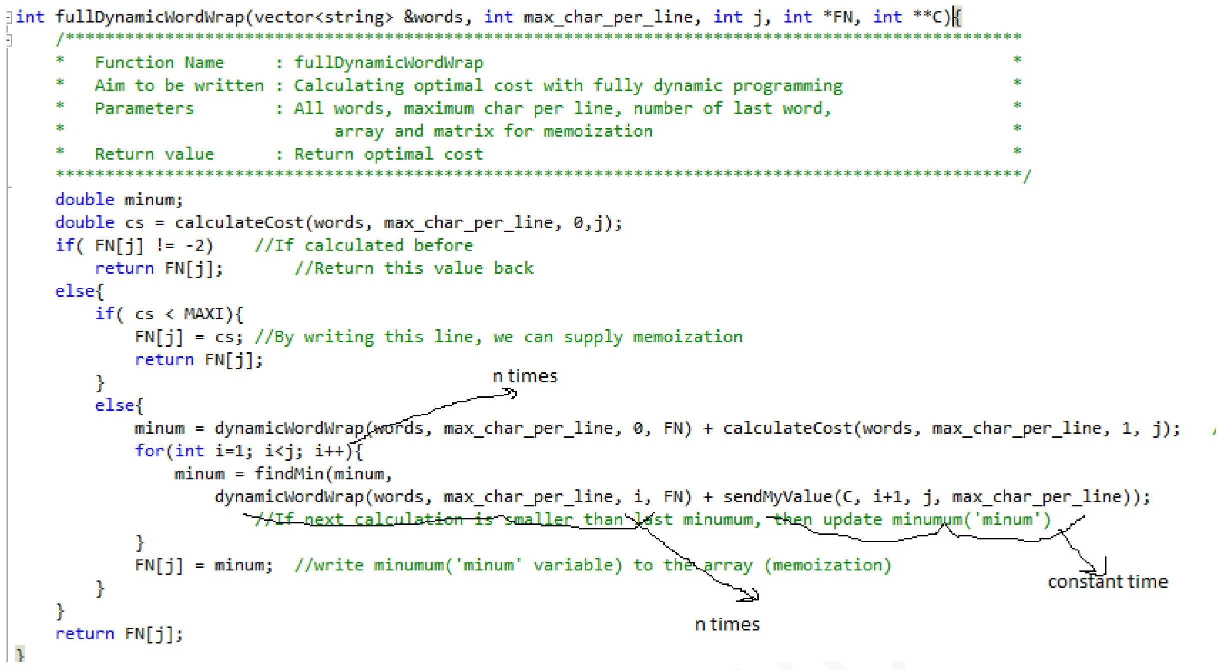
Figure 4. CalculateCost Function and running time

As seen the Figure 3&Figure 4, my dynamic algorithm's time complexity: **O(n * (n + maxChar))**

Fully Dynamic Programming

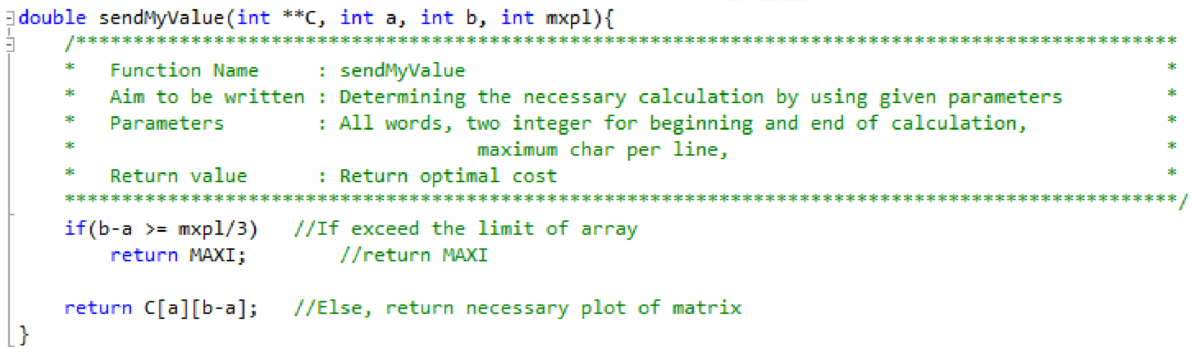
My algorithm for problem with fully dynamic programming solution:

- My fully dynamic algorithm is very similar to the dynamic algorithm except not calculating cost for each time.
- According to the algorithm, my codes are below.



```
int fullDynamicWordWrap(vector<string> &words, int max_char_per_line, int j, int *FN, int **C){
    /*****************************************************************************************
    *   Function Name     : fullDynamicWordWrap                                               *
    *   Aim to be written : Calculating optimal cost with fully dynamic programming           *
    *   Parameters        : All words, maximum char per line, number of last word,            *
    *                           array and matrix for memoization                              *
    *   Return value      : Return optimal cost                                               *
    *****************************************************************************************/
    double minum;
    double cs = calculateCost(words, max_char_per_line, 0,j);
    if( FN[j] != -2)    //If calculated before
        return FN[j];       //Return this value back
    else{
        if( cs < MAXI){
            FN[j] = cs; //By writing this line, we can supply memoization
            return FN[j];
        }
        else{
            minum = dynamicWordWrap(words, max_char_per_line, 0, FN) + calculateCost(words, max_char_per_line, 1, j);
            for(int i=1; i<j; i++){
                minum = findMin(minum,
                    dynamicWordWrap(words, max_char_per_line, i, FN) + sendMyValue(C, i+1, j, max_char_per_line));
                    //If next calculation is smaller than last minumum, then update minumum('minum')
            }
            FN[j] = minum;  //write minumum('minum' variable) to the array (memoization)
        }
    }
    return FN[j];
}
```

Figure 5. Fully dynamic solution in C++



```
double sendMyValue(int **C, int a, int b, int mxpl){
    /****************************************************************************************
    *   Function Name     : sendMyValue                                                      *
    *   Aim to be written : Determining the necessary calculation by using given parameters  *
    *   Parameters        : All words, two integer for beginning and end of calculation,     *
    *                           maximum char per line,                                       *
    *   Return value      : Return optimal cost                                              *
    ****************************************************************************************/
    if(b-a >= mxpl/3)   //If exceed the limit of array
        return MAXI;        //return MAXI

    return C[a][b-a];   //Else, return necessary plot of matrix
}
```

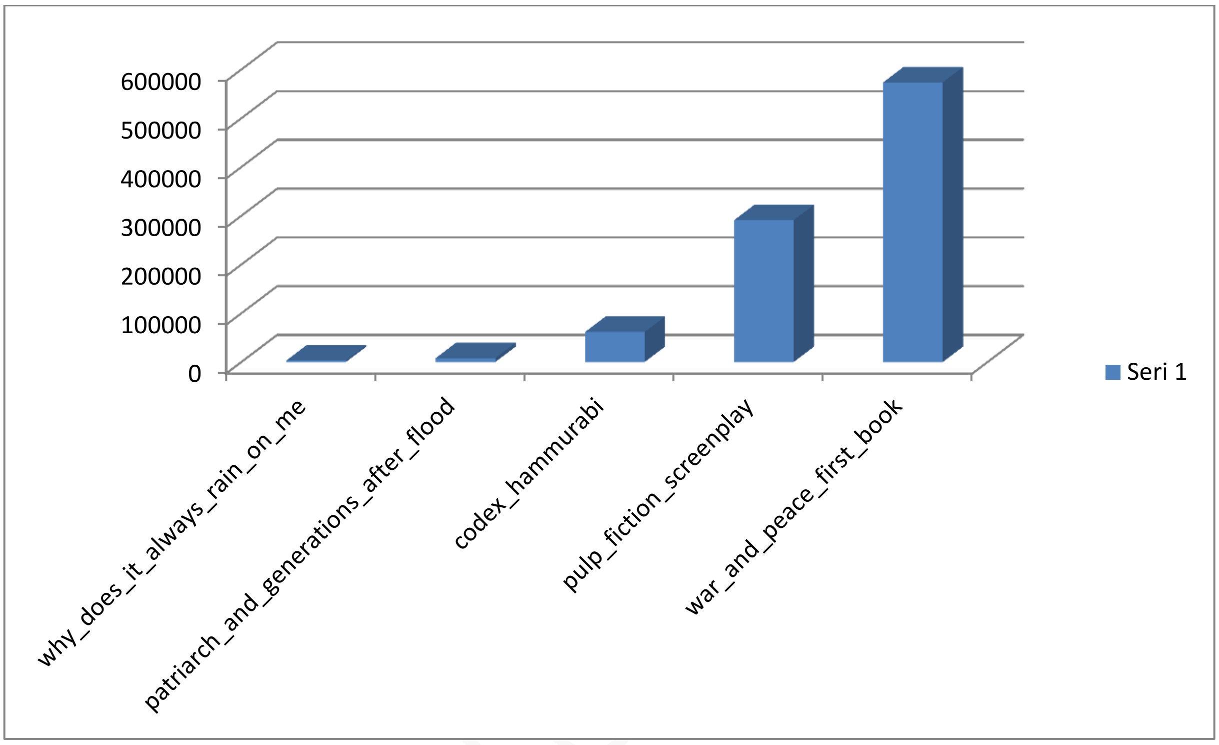
Figure 6. sendMyValue Function

As seen the Figure 5&Figure 6, my dynamic algorithm's time complexity: **O(n * (n + c)) = O(n^2)**

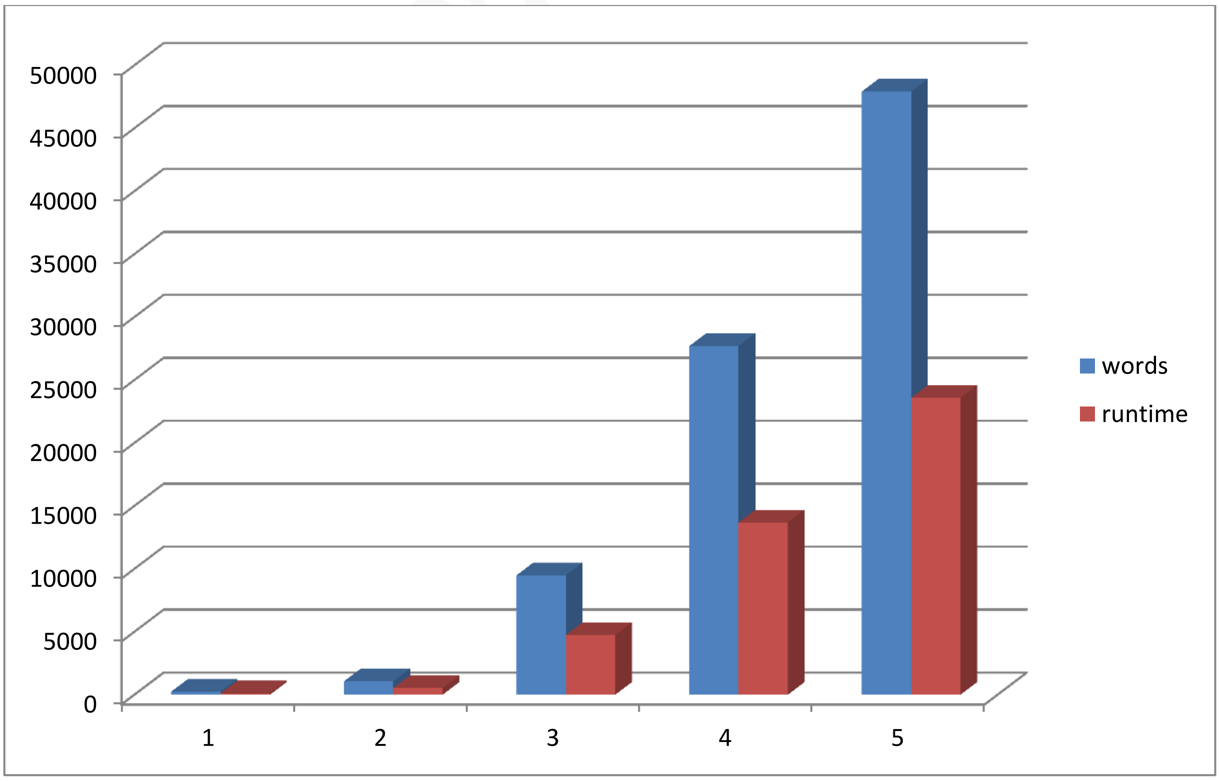
My Algorithms and Exact Running Times

<u>Greedy:</u>

<u>COST:</u>



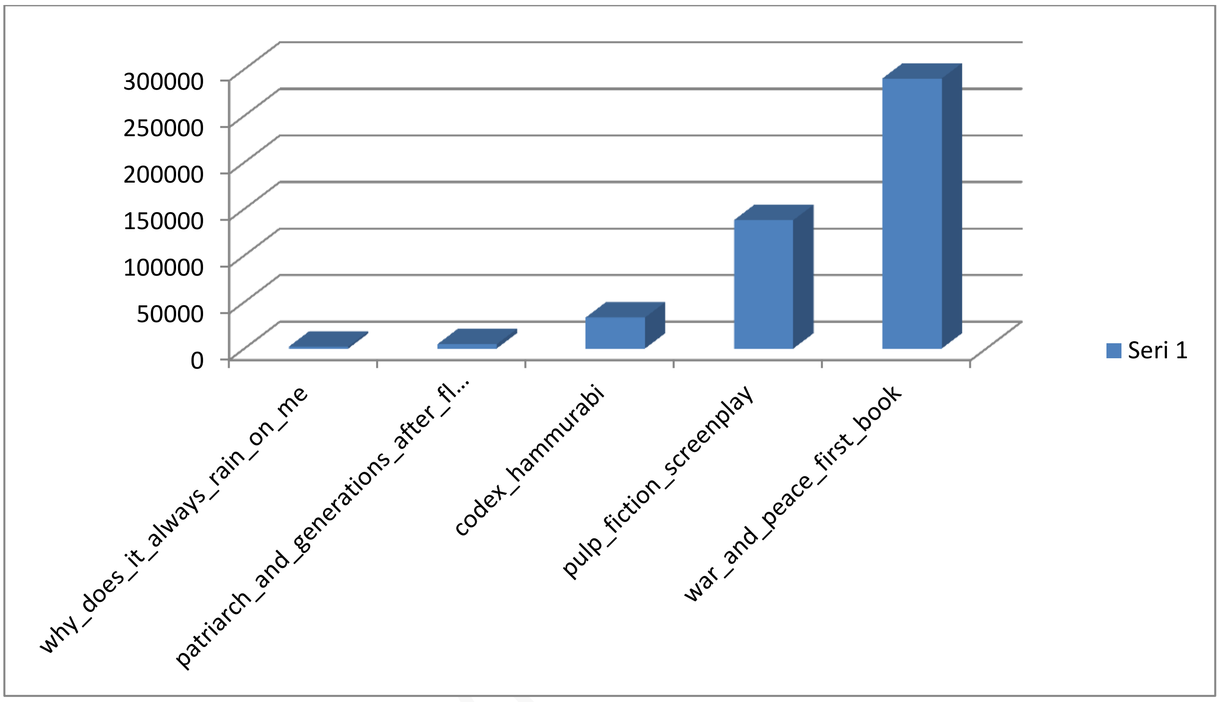
<u>Run Time - #of Words:</u>



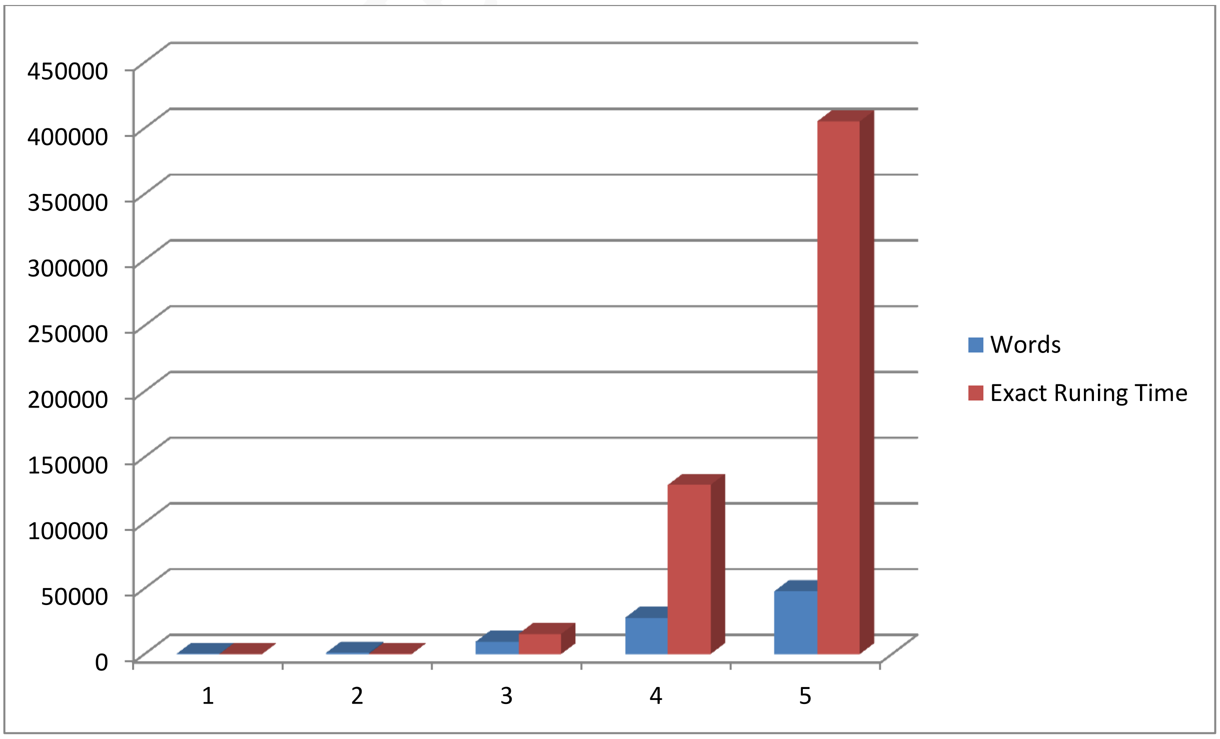
I've run the program 1000 times for all text files and runtime given above graphic is thought 1000 times more as normal time. In addition, run time graphic is expected since time complexity is 'n'.

Dynamic:

COST:



Run Time - #of Words:



Run time - # of words graphic is acceptable since the time complexity is O(n^2). I've calculated run time/(# of words)^2 for each text and all of these values are closer to others.

Comparision of 2 Algorithms

Firstly, I've implemented greedy algorithm. This algorithm runs fast as seen in above pages. But with greedy algorithm, costs are very high. Greedy algorithm is a solution of word wrapping but not good solution if we want to minimize spaces in the lines.

Secondly, I've implemented dynamic programming approach. By using this approach, I needed to wait many times while testing large texts. But end of the waiting, I've got optimal solution. In addition, with dynamic programming algorithm program will be need high amount of memory.

By comparing this two algorithms, if you need to get a solution but not have to be optimal, you must use greedy algorithm. In contrast, if you need to get an optimal solution independent from time, then you must use dynamic programming algorithm. In addition, if you do not have high amount of memory, then you must use the greedy approach because of the fact that dynamic programming approach requires more&more memory.

Compilation and Running

I have compiled and run, without any error, my code in Win7 Microsoft Visual Studio 2010.