BLG372E Analysis of Algorithms

PROJECT 2

Ozan Arkan Can

040090573

Instructor: Zehra Çataltepe

Teaching Assistant: Ahmet Aycan ATAK

Submission Date: 04.05.2012

CRN: 22534

Introduction

In this project, it is aim that designing an algorithm for word wrapping problem. According to the problem, it is aim that aparting a long line into small pieces and minimizing cost value. Cost value is defined as:

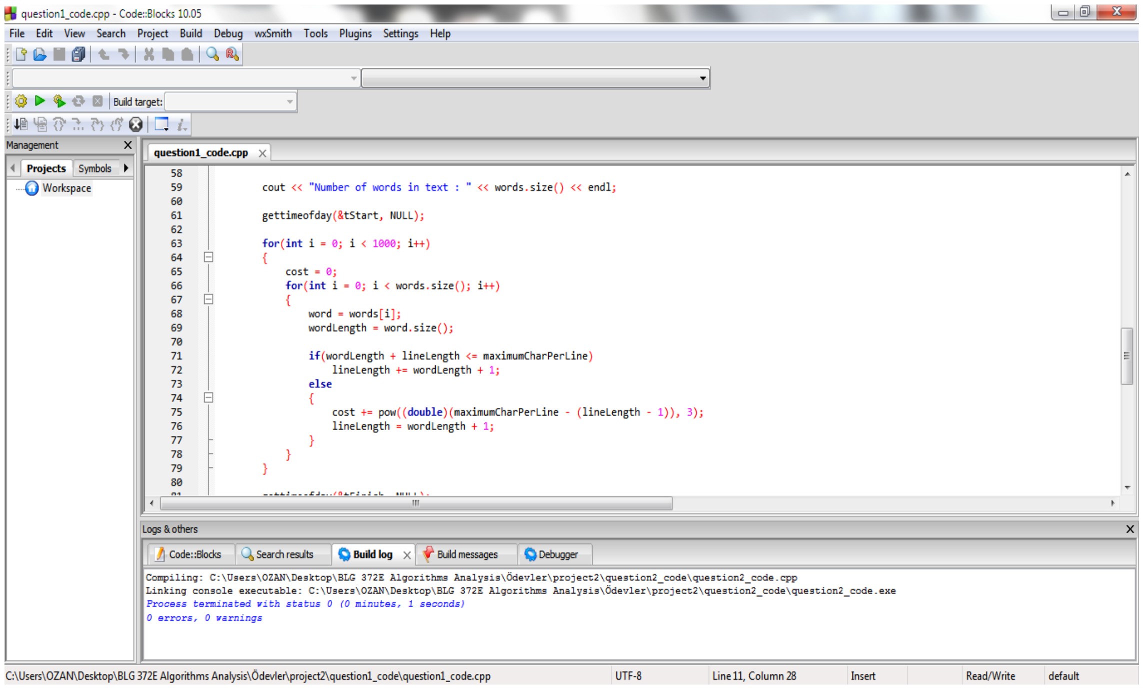
For each small part:
cost += (maximumCharPerLine-NumberOfCharactersInTheLine-(NumberOfWords-1))^3

The roblem can be solved by dynamic programming or greedy aproach. Pseudocode for greedy aproach was given in the project document. In first part of project, the greedy aproach is implemented by using c++. In second part, a dynamic programming algorithm is designed and implemented by using c++ for word wrapping problem. For both aproach, complexity of algorithms, running times of algorithms and cost results for test texts are analyzed. In last section, greedy aproach and dynamic programming aproach are compared.

Development and Operating Environments

MS Windows

The Code::Blocks IDE has been used to write source code, compile and run the code.


question1_code.cpp - Code::Blocks 10.05
File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help
Build target:
Management
Projects Symbols
Workspace
question1_code.cpp
Logs & others
Code::Blocks Search results Build log Build messages Debugger

```
        cout << "Number of words in text : " << words.size() << endl;

        gettimeofday(&tStart, NULL);

        for(int i = 0; i < 1000; i++)
        {
            cost = 0;
            for(int i = 0; i < words.size(); i++)
            {
                word = words[i];
                wordLength = word.size();

                if(wordLength + lineLength <= maximumCharPerLine)
                    lineLength += wordLength + 1;
                else
                {
                    cost += pow((double)(maximumCharPerLine - (lineLength - 1)), 3);
                    lineLength = wordLength + 1;
                }
            }
        }
```

```
Compiling: C:\Users\OZAN\Desktop\BLG 372E Algorithms Analysis\Ödevler\project2\question2_code\question2_code.cpp
Linking console executable: C:\Users\OZAN\Desktop\BLG 372E Algorithms Analysis\Ödevler\project2\question2_code\question2_code.exe
Process terminated with status 0 (0 minutes, 1 seconds)
0 errors, 0 warnings
```

C:\Users\OZAN\Desktop\BLG 372E Algorithms Analysis\Ödevler\project2\question1_code\question1_code.cpp | UTF-8 | Line 11, Column 28 | Insert | Read/Write | default

UNIX

The source code has been also copied to Unix, then compiled and tested with GNU C++ Compiler. The following commands have been used;

To Compile:

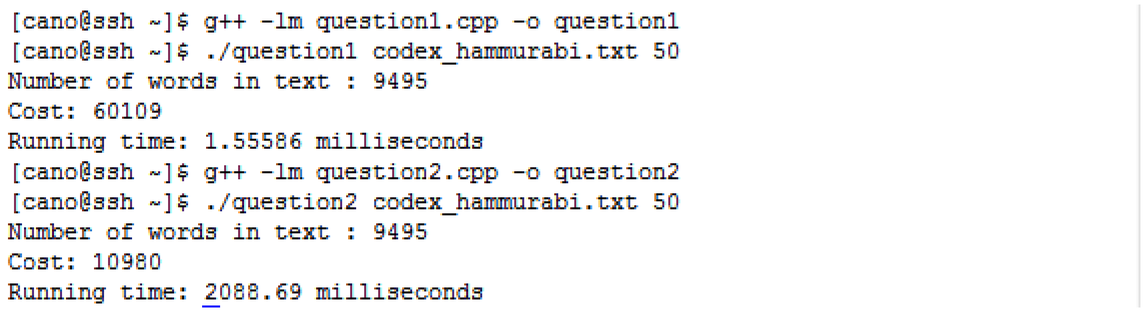
For greedy aproach
▷ g++ -lm question1.cpp -o question1

For dynamic programming aproach
▷ g++ -lm question2.cpp -o question2

To Run:

For greedy aproach
▷ .\question1 codex_hammurabi.txt 50

For dynamic programming aproach
▷ .\question2 codex_hammurabi.txt 50



```
[cano@ssh ~]$ g++ -lm question1.cpp -o question1
[cano@ssh ~]$ ./question1 codex_hammurabi.txt 50
Number of words in text : 9495
Cost: 60109
Running time: 1.55586 milliseconds
[cano@ssh ~]$ g++ -lm question2.cpp -o question2
[cano@ssh ~]$ ./question2 codex_hammurabi.txt 50
Number of words in text : 9495
Cost: 10980
Running time: 2088.69 milliseconds
```

Complexity of Algorithm

In this section greedy aproach for word wrapping problem is analyzed. It is obvious that there is one loop and it iterates n times that equals number of words in text. So the complextiy of algorithm is $O(n)$.

Here is the graph for running time against number of words in text. The test texts and number of words in texts are like following:

$1 : codex\_hammurabi.txt \rightarrow NumberOfWords : 9495$
$2 : patriarch\_and\_generations\_after\_flood.txt \rightarrow NumberOfWords : 1067$
$3 : pulp\_fiction\_screenplay.txt \rightarrow NumberOfWords : 27691$
$4 : war\_and\_peace\_first\_book.txt \rightarrow NumberOfWords : 47913$
$5 : why\_does\_it\_always\_rain\_on\_me.txt \rightarrow NumberOfWords : 259$

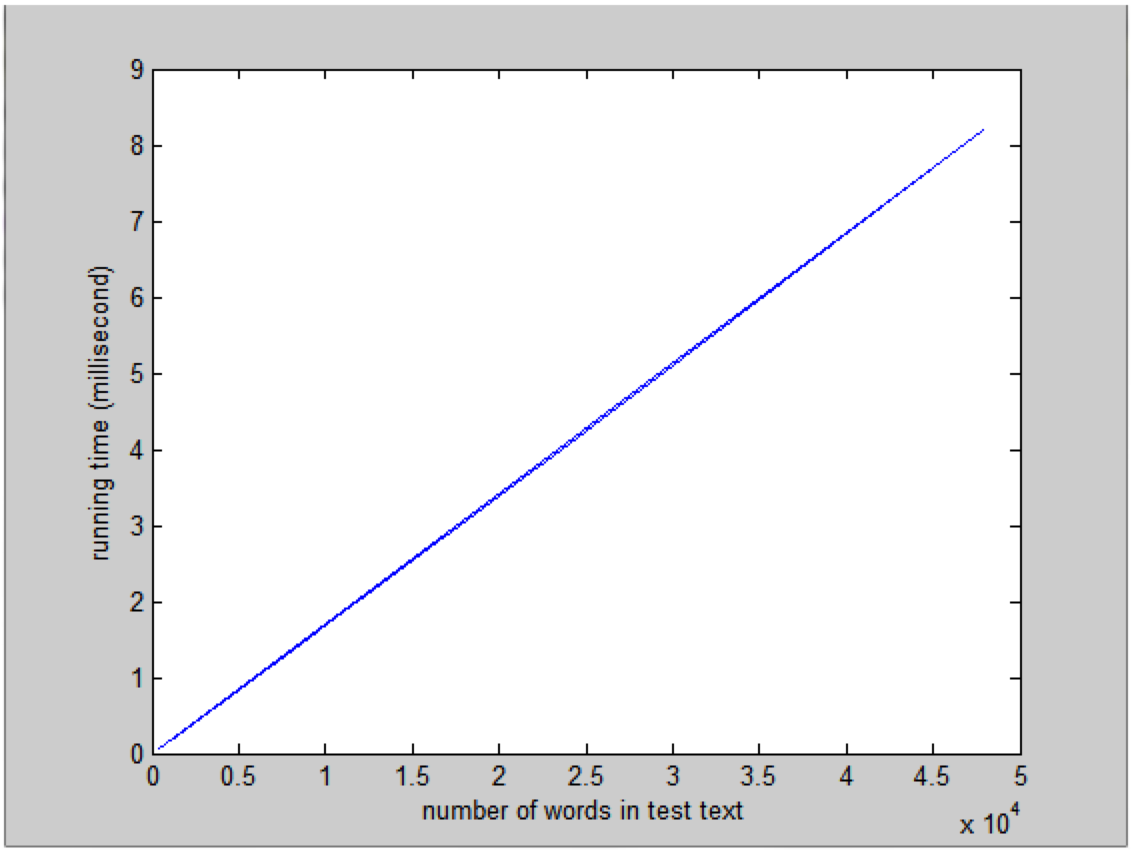


Figure1: Running time of greedy aproach

It can be seen that the running time function is a linear function. When number of words in text increases the running time increases linearly. It matches with complexity analysis.

Also, the cost value was analyzed for test text and graph shows the cost values for each test text.

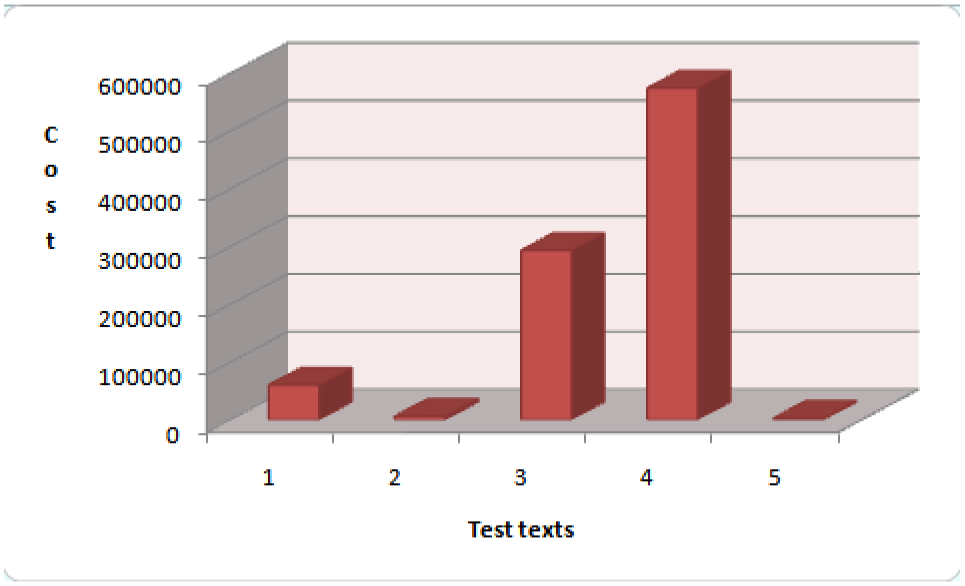


Figure2: Cost values of test text for greedy aproach

According to the figure2, it can be said that the cost value increases by number of words in text.

Question2

Dynamic programming aproach has two basic properties, recursion and memoization. The ideas behind them are solving problem with parting problem into subproblem and calculating subproblem only one time and store for using whent it necessery. An algorithm is developed by using dynammic programming for word wrapping problem.

Pseudocode

Algorithm solves problem with OptCost function. OptCost function does that if value of function was calculated and it is smaller than infinity(in code INT_MAX) ruturns it else finds minimum cost by calling function smaller index plus cost of rest of line.

Pseudocode for main program:

```
1: INPUT: test text, maximumCharPerLine
2: OUTPUT: number of words, running time cost
3: cost = 0
4: create a lookup table with INT_MIN initial values for OptCost function
5: cost ← OptCost(number of words in text)
```

Pseudocode for OptCost

```
1: INPUT: wordIndex, text, maxCharPerLine
2: OUTPUT: cost
3: cost ← getCost(firstIndex, lastIndex, text, maxCharPerLine)
4: min ← INT_MAX
5: IF cost == INT_MAX or cost == INT_MIN
6:     distance ← wordIndex - (maxCharPerLine + 1) / 2
7:     IF distance < 0
8:         distance ← 0
9:     FOR i = distance to wordIndex
10:        tempCost ← getCost(i + 1, wordIndex, text, maxCharPerLine)
11:        IF tempCost == INT_MAX
12:            currentValue ← INT_MAX
13:        ELSE
14:            IF functionValue[i] == INT_MIN
15:                functionValue[i] ← OptCost(i, words, maxCharPerLine)
16:            IF functionValue[i] == INT_MAX
17:                currentValue ← INT_MAX;
18:            ELSE
19:                currentValue ← tempCost + functionValue[i]
20:        IF currentValue < min
21:            min ← currentValue
22:     return ← min
23: ELSE
24:     return ← cost
```

getCost function returns cost value for piece of line between firstIndex and lastIndex. And the cost defination was given in the project document.

Complexity of Algorithm

The dynamic programming aproach has recursive part and in this algorithm OptCost function is a recursive function. OptCost function calls itself in loop at line 9. There is a optimization at line 6. distance variable holds how many words can be in a line part that total number of characters and spaces do not exceed the maximum character per line value. It can be said that if every word has one character the distance value can be calculated with this equation: $distance = wordIndex - (maxCharPerLine + 1)/2$. OptCost function is called n times and is called m times for each recursive function call.Here, n is number of words in text and m is wordIndex minus distance. So, complextiy of algorithm is $O(n * m)$. Algorithm has also memoization. funcitonValue array holds value that OptCost returned. If funcitonValue[i] equals INT_MIN that means OptCost(i) has not been calculated yet. If it has pozitive integer value OptCost function does not be called.

Here is the graph for running time against number of words in text.

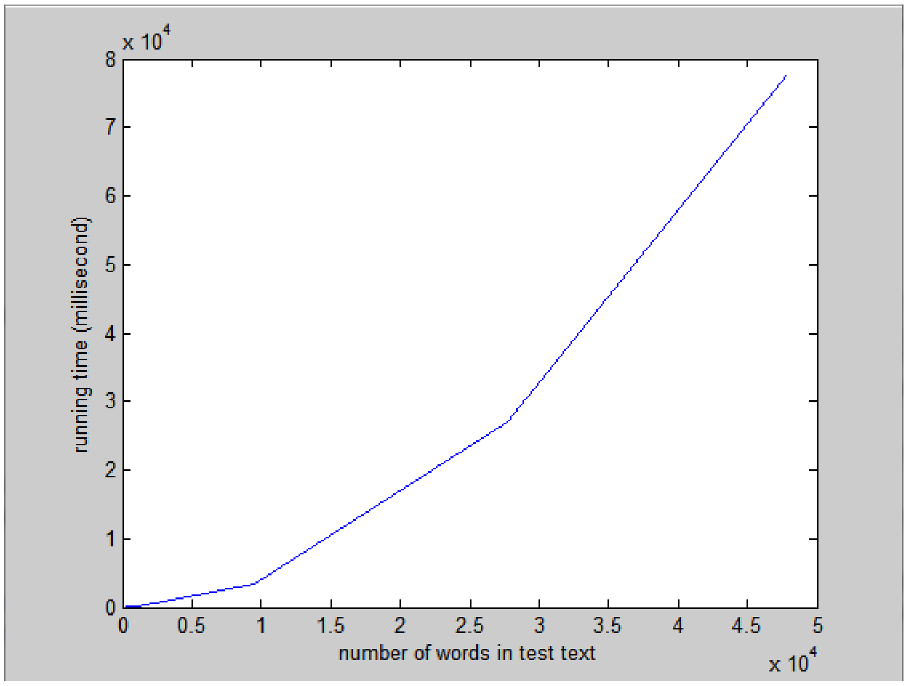


Figure3: Running time of dynamic programming aproach

It can be seen that the running time function is a curve that increases proportional with second pow of number of words in text.It matches with complexity analysis.

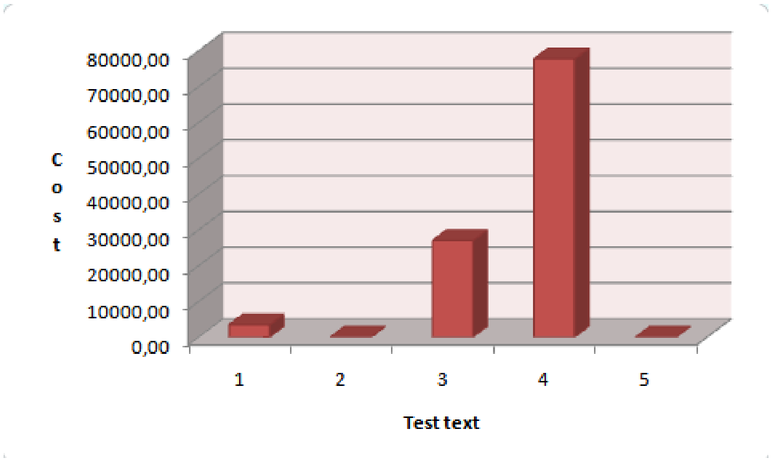


Figure4: Cost values of test text for dynamic programming aproach

According to the figure4, it can be said that the cost value increases by number of words in text.

Question3

It is obvious that greedy algorithm faster than dynamic programming aproach, but in general case number of words is distinct for word wrapping problem. Also, it can be seen from figure2 and figure4 that dynamic programming aproach reduces the cost value by one over ten against greedy algorithm. Dynamic programming algorithm needs $O(n)$ space for memoization. There is no space demand in greedy aproach. According to the theese comparisons, if optimum cost is aimed, dynamic programming algorithm must be used, if optimum running time is aimed, greedy aproach must be used.

Conclusion

In this project,

- Dynamic programming theory and greedy algorithms are investigated and learned.
- To experience analyzing algorithm.
- To experience devoloping efficent algorithm for a problem.
- To experience there are several aproaches for a problem with advantages and disadvantages.
- Memoization issue is learned.