Universitatea din Craiova
Facultatea de Automatică,Calculatoare
şi Electronică

01 June 2018

Project : Algorithm Design
Title : Change Making Problem
Teachers : Bădică Costin & Murareţu Ionuţ & Becheru Alex
Student : Moraru Teodor Valentin
Section : Calculatoare Română
Year I
Group : 1.2 B

Contents

1 Problem Statement

1.1 Title

A project designed to solve the Change Making Problem in two different methods : a greedy algorithm and a dynamic algorithm.

1.2 Description

My project target is to solve the change making problem in two different implementations .The owed sum , number of coins and the actual coins values are read form the file ”input.txt”.The values in the file will be added by the uses as following : the first number in the file represents the owed sum , the second one is the number of coins and the others will be the coin values .

Dynamic allocation is the automatic allocation of memory in C/C++, Unlike declarations, which load data onto the programs data segment, dynamic allocation creates new usable space on the programs STACK(an area of RAM specifically allocated to that program).For this reason the matrix was dynamically allocated in the dynamic implementation.

The greedy method was kept as simple as possible in order to hit it's main target that of being an easy comprehensive code .But it is not as effective nor as

2 Pseudocode

2.1 get_value

```
get_value($INT$ *owed , $INT$ *coins_no , $INT$ coins_list[100])
$START$
$INT$ i , hold_owed ,hold_no
$FILE * input\_file$
$input\_file = fopen("input.txt", "r")$
$fscanf(input\_file, "\%d", \&hold\_owed)$
$*owed = hold_owed;$
$fscanf(input_file, "\%d", \&hold\_no)$
$*coins\_no = hold\_no$
for $i = 0$ to coins_no do
    $fscanf(input_file, "\%d", \&coins\_list[i])$
end for
$fclose(input\_file)$
$END$
```

2.2 greedy_method

```
greedy_method($INT$ owed ,$INT$ coins_no ,$INT$ coins_list[100],$INT$ good_picks[100])
$START$
$INT$ i , good_picks[100]
for $i = 0$ to coins_no do
    good_picks[i] = owed div coins_list[i]
    owed = owed mod coins_list[i]
end for
if ($owed == 0$) then
    print("Coins used:")
    for $i = 0$ to coins_no do
        print("%d x %d " , good_picks[i],coins_list[i])
    end for
else
    printf("Coins used:")
    for $i = 0$ to coins_no do
        print("%d x %d " , good_picks[i],coins_list[i])
    end for
    print("Remainder value that could not be changed is : %d",owed)
end if
$END$
```

2.3 dynamic_method

```
dynamic_method($CONST\ INT$ *coins_list, $INT$ coins_no, $INT$ owed, $INT$
    **solution)
$START$
$INT$ i , j , result
table = malloc((coins_no + 1) * sizeof(change_entry *))
**for** $i = 0$ **to** coins_no **do**
    table[i] = malloc((owed + 1) * sizeof(change_entry))
**end for**
**for** $i = 0$ **to** coins_no **do**
    **for** $j = 0$ **to** owed **do**
        **if** $(i == 0)$ **then**
            table[i][j].count = j
            table[i][j].coin = -1
            table[i][j].prev = NULL
        **else if** $(j == 0)$ **then**
            table[i][j].count = 0
            table[i][j].coin = -1;
            table[i][j].prev = NULL
        **else if** $(coins\_list[i-1] == j)$ **then**
            table[i][j].count = 1
            table[i][j].coin = i - 1
            table[i][j].prev = NULL
        **else if** $(coins\_list[i-1] > j)$ **then**
            table[i][j].count = table[i - 1][j].count
            table[i][j].coin = -1
            table[i][j].prev = &table[i - 1][j]
        **else**
            **if** $(table[i-1][j].count < table[i][j-coins_{l}ist[i-1]].count+1)$ **then**
                table[i][j].count = table[i - 1][j].count
                table[i][j].coin = -1
                table[i][j].prev = &table[i - 1][j]
            **else**
                table[i][j].count = table[i][j - coins_list[i - 1]].count + 1
                table[i][j].coin = i - 1
                table[i][j].prev = &table[i][j - coins_list[i - 1]]
            **end if**
        **end if**
    **end for**
**end for**
result = table[coins_no][owed].count
*solution = calloc(coins_no, sizeof(int))
```

```
**if** $(*solution)$ **then**
    change_entry *head
    **for** $head = \&table[coins\_no][owed]; head! = NULL; head = head- > prev$ **do**
        **if** $(head- > coin! = -1)$ **then**
            (*solution)[head-¿coin]++
        **end if**
    **end for**
**else**
    result = 0
**end if**
**for** $i = 0$ **to** coins_no **do**
    free(table[i])
**end for**
free(table)
return result
$END$
```

3 Application Design

3.1 Main

This module contains only the main function .The user is required to introduce an input in the file "input.txt" as following : the first digit from the file represents the owed sum , the second digit represents the number of coins and the following digits are the actual coin values (those should always be introduced in a descending order for the greedy algorithm to be usable). The program will run from the executable file generated from the respective makefile found in the main directory and the output and also the execution time will be shown on the run screen.

The main function contains only functions calls.

Functions called :

get_values()

greedy_method()

dynamic_method()

The get_values function will be used to read the "input.txt" file as it was already specified .

Variables used:

int *owed : pointer to the owed sum.

int *coins_no : pointer to the number of coins.

int coins_list[] : an array of coins values (in a descending order because of the greedy implementation requirements as specified in the **Section 3.2**).

int good_picks : an array of the selected coins for the solution.

int i : an iterator.

int hold_owed : a variable used to hold and pass the owed sum to a pointer.

int hold_no : a variable used to hold and pass the number of coins to a pointer.

The greedy_method uses a relatively easy algorithm to be understood by any user , it requires repetitive div and mod operations to the owed sum , the div operation is used every time to chose whether a coin is usable or not and the mod operation is used to transition to the next coin that will be tested during the iteration.

Variables used:

int owed : the owed sum.

int coins_no : the number of coins.

int coins_list[] : an array of coins values (in a descending order because of the greedy implementation requirements as specified in the **Section 3.2**).

int good_picks[] : an array of the selected coins for the solution.

int i : an iterator.

The dynamic algorithm uses a dynamic allocated table structure to store and eventually read back the solution , this method consists of multiple checks used to initially build owed sums and chains then the rows and columns are initialized , forward the algorithm has to determine whether a coin is usable or not and also whether that coin is the best solution for the respective input.

Variables used:

struct change_entry : a structure used to ease the readability of the solution.

int result : a variable used to keep the number of used coins.

int owed : the owed sum.

int coins_no : the number of coins.

int coins_list[] : an array of coins values (in a descending order because of the greedy implementation requirements as specified in the **Section 3.2**).

int good_picks[] : an array of the selected coins for the solution.

int i : an iterator.

int j : an iterator.

3.2 Input Data

The user is required to introduce an input in the file "input.txt" as following : the first digit from the file represents the owed sum , the second digit represents the number of coins and the following digits are the actual coin values (those should always be introduced in a descending order for the greedy algorithm to be usable).

3.3 Output Data

If the input data is selected as I specified in the **Section 3.1** and also **Section 3.2** the output will be successfully shown on the run screen by both of the algorithms used , also the runtime for both algorithms will be present in order to prove the efficiency of the dynamic implementation over the greedy one but also its deficit in runtime for a large input size.

3.4 Functions

The functions used in the program are presented in **Section 2** in their respective pseudocode forms and their functionality is explained in the above **Section 3.1** alongside the vast majority of all variables used and their respective roles.

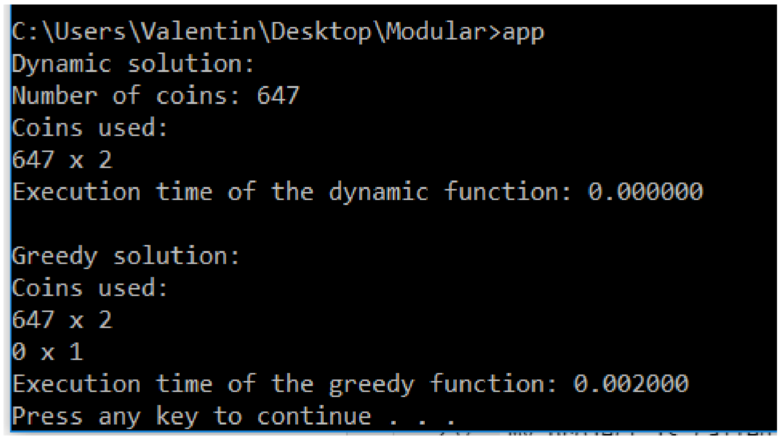
4 Source Code

My project is called "The Change Making Problem". The source code is created in the programming language standard C99 and it was successfully compiled in two different compilers.

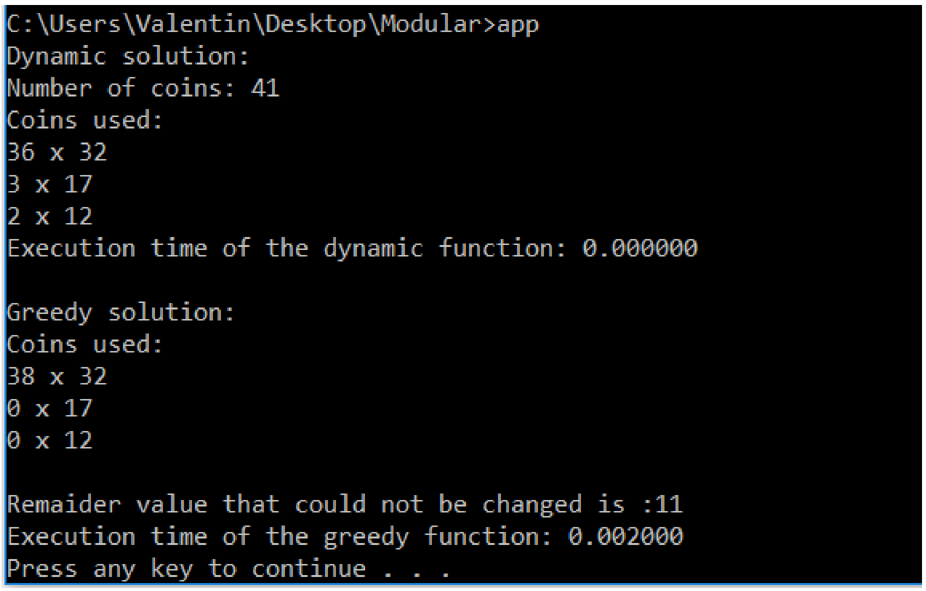
5 Experiments and results

5.1 GNU GCC Compiler

For the GCC compiler, I will present multiple generated outputs, from those the execution time will show the actual speed of each algorithm for small and big input sizes for both the owed sum and also the number of denominations used. Those conclusions will be presented in the next subsection.

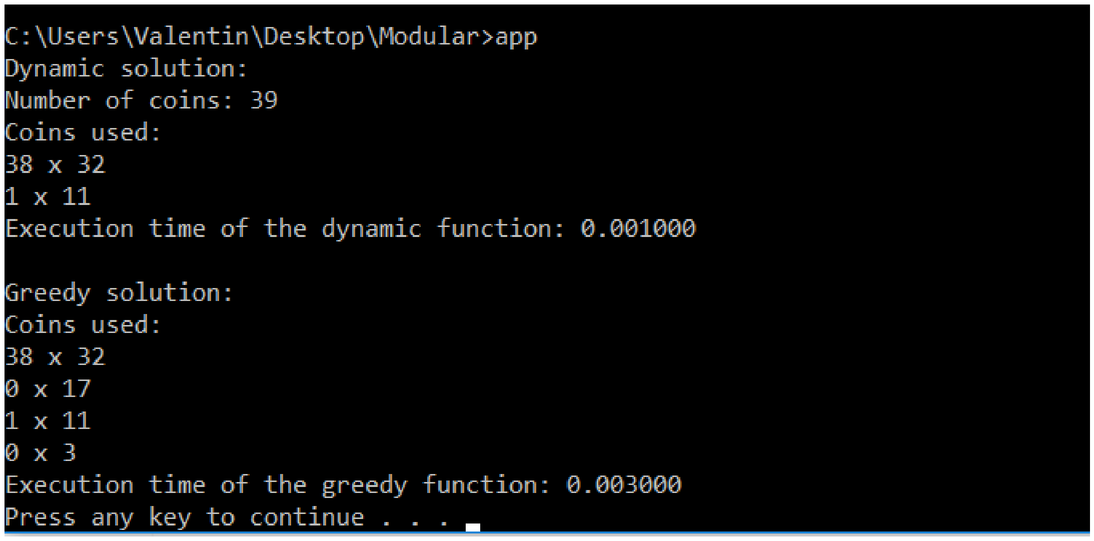


```
C:\Users\Valentin\Desktop\Modular>app
Dynamic solution:
Number of coins: 647
Coins used:
647 x 2
Execution time of the dynamic function: 0.000000

Greedy solution:
Coins used:
647 x 2
0 x 1
Execution time of the greedy function: 0.002000
Press any key to continue . . .
```

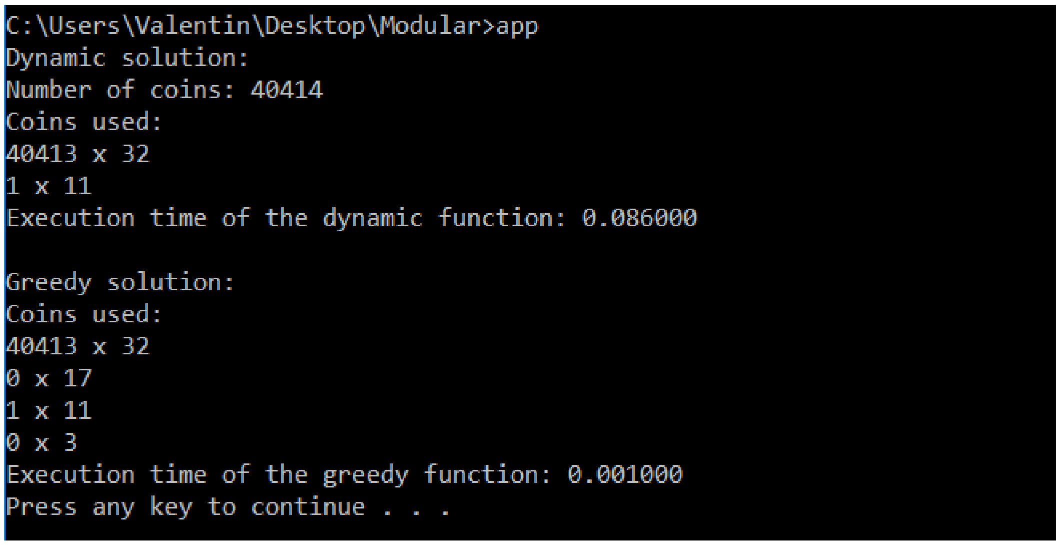


```
C:\Users\Valentin\Desktop\Modular>app
Dynamic solution:
Number of coins: 41
Coins used:
36 x 32
3 x 17
2 x 12
Execution time of the dynamic function: 0.000000

Greedy solution:
Coins used:
38 x 32
0 x 17
0 x 12

Remaider value that could not be changed is :11
Execution time of the greedy function: 0.002000
Press any key to continue . . .
```

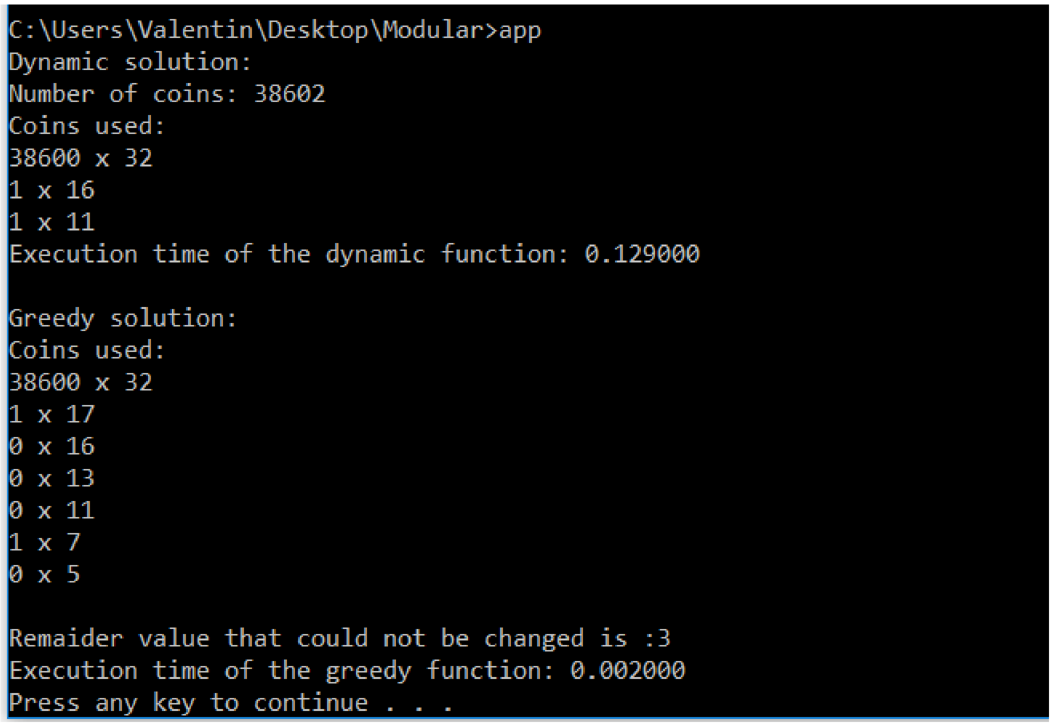
For a small input size the dynamic execution time is close to 0 seconds whereas the execution time for the greedy algorithm is countable.



```
C:\Users\Valentin\Desktop\Modular>app
Dynamic solution:
Number of coins: 39
Coins used:
38 x 32
1 x 11
Execution time of the dynamic function: 0.001000

Greedy solution:
Coins used:
38 x 32
0 x 17
1 x 11
0 x 3
Execution time of the greedy function: 0.003000
Press any key to continue . . .
```



```
C:\Users\Valentin\Desktop\Modular>app
Dynamic solution:
Number of coins: 40414
Coins used:
40413 x 32
1 x 11
Execution time of the dynamic function: 0.086000

Greedy solution:
Coins used:
40413 x 32
0 x 17
1 x 11
0 x 3
Execution time of the greedy function: 0.001000
Press any key to continue . . .
```



```
C:\Users\Valentin\Desktop\Modular>app
Dynamic solution:
Number of coins: 38602
Coins used:
38600 x 32
1 x 16
1 x 11
Execution time of the dynamic function: 0.129000

Greedy solution:
Coins used:
38600 x 32
1 x 17
0 x 16
0 x 13
0 x 11
1 x 7
0 x 5

Remaider value that could not be changed is :3
Execution time of the greedy function: 0.002000
Press any key to continue . . .
```

For a big input size the dynamic execution time is higher than the execution time of the greedy approach as shown in the above pictures taken from different tests.

6 Conclusion

This program depends on the size of the file , the dynamic approach has an execution time close to 0 for a small file but for a big file with a high owed sum and a high number of coins the execution time goes up by much .But using dynamic programming, the solution of the problem is always effective and optimal.

The greedy approach ,on the other hand,has a relatively constant execution time and it does not increase by much when the file gets larger . But its solution is not optimal every time.

The most challenging part of this program was the dynamic allocation of a matrix inside the dynamic implementation and also the method itself which is notably harder to comprehend with compared to the greedy method which is easy to use and understand but not as effective .

7 References

Books:
1. Totul despre C si C++
Year of publication :2005
Publisher :Teora
Author :Dr. Kris Jamsa Lars Klander

Web references:
1.$http://www.geeksforgeeks.org$
2.$https://www.sharelatex.com/learn/Main_Page$
3.$https://www.reddit.com/r/C_Programming$

Article:
1.$https://en.wikipedia.org/wiki/Change-making\_problem$