Solutions for the coin change problem

Algorithms for returning a change

Francesco Gazzola, Amadou Asmaou

ENSSAT, 2021

ENSSAT

LANNION

Contents

Chapter 1

Introduction

This report presents three different algorithms for solving the coin change problem. Such problem concerns in determining the optimal list of coins to use for giving back a change. Indeed, vary combinations exist for a given change and for such reason we decided to focus on the optimal one, which consists of the minimum number of coins.

We have hence developed three algorithms by exploiting three different problem solving techniques: backtracking, dynamic programming and greedy approach. For each algorithm an analysis is given together with the pseudo-code. We report also the results obtained by carrying out some tests with different inputs.

Chapter 2

Solutions

In this section we are listing the algorithms we developed and the results we obtained.
For each approach we start from giving the pseudo code, then we move to analyse the performance of the algorithm and we finish with the tests we carried out.
We conclude the chapter with a comparison of the algorithms in terms of their performance.

The code implementation for each technique here described can be found in the appendix at section 4.4.

2.1 Backtracking approach

Algorithm 1: procedure min_change(change, return_coins)

```
for coin in the list of coins do
    change = change - coin;
    return_coins = return_coins + coin;
    if change > 0 and size(return_coins) < size(best_coins) then
        if change == 0 then
            best_coins = return_coins ;
        end
        min_change(change, return_coins);
    end
    change = change + coin;
    return_coins = return_coins - coin;
end
```

2.1.1 Fundamental constituents

Looking at the code above, we can distinguish the fundamental constituents which are the core of such technique. These are the following:

* **Satisfaisant**

  change > 0 and size(return_coins) < size(best_coins)

* **Enregistrer**

  change = change - coin

  return_coins = return_coins + coin

* **Soltrouvée**

change == 0

* **Défaire**

  change = change + coin

  return_coins = return_coins - coin

2.1.2 Representation of the solution

For such problem we choose a list for representing the solution. Each content in the list represents a coin.
The criterion that a solution must respect are:

* The total sum of coins that the solution contains must be equal to the input change.
* The length of the solution (hence of the list) must be the shortest as possible among the possible combinations we can return the inputted change.

2.1.3 Time complexity analysis

We are now going to analyse our algorithm to give an order of time complexity.

The first consideration to be done regards the **pruning condition**. In the 'Satisfaisant' constituent we set the condition:

$$size(return\_coins) < size(best\_coins)$$

which let us stop the generation of the recursive calls by excluding all those solutions which are not going to build an optimal solution since their length is larger than the best one we have already found. Indeed, as soon as we find a feasible solution to be the best, we are interesting in computing only smaller

solutions than that one. In particular, without such condition we are going to build all the possible solutions and so generating all the combinations of coins we can use to give back the change. This would result in a really expensive computational effort, which would be proportional to the the total number of combinations of coins we can use to form the change with n coins, that is (n^{change}). Hence, this leads to a polynomial complexity $\Theta(n^k)$ with n = *cardinality of the input coins set* and k = *change in a no-decimal format* (e.g. change = 1.90€ -> k = 190).

By using the pruning condition, we could obtain a backtrack algorithm running in an order of time $O(n^{k-1})$ with n = *cardinality of the input coins set* and k = *change in a no-decimal format*. This estimations is based on the number of recursive calls the algorithm is doing and it can be achieved by observing that our algorithm is never calculating all the possible combinations (figure in section 4.1 in the appendix). Moreover, at the leaves there is never a recursive call and hence we can consider $k - 1$ instead of k. We can also notice that if we use set of coins whose smallest denomination is 1, then k represents the maximum depth the recursive tree can achieve since one combination might be representing k with only 1s.

We report in the following table the run time and the number of combinations computed by using and not using the pruning condition.
We inputted the change = 31 and we used the Euro currency.

	without Pruning	with Pruning
#combinations computed	10050981	6
run time	1605.484 seconds = 26.75 minutes	0.004 seconds
solution	[1, 10, 20]	

Possible Improvements

From the above analysis we can argue that such algorithm can be improved by avoiding to compute all the combinations. This can achieve by exploiting the optimization technique, called memoization and so by using the dynamic programming approach presented in the section 2.2.
Moreover, we can try to figure out some property of the set of coins and exploit such property to improve the run time of the program. In the section 4.2 in the appendix, we present a backtracking solution running in a time $O(n^2)$ for the Euro currency, which exploits the descending order imposed on the set of coins.

2.1.4 Experiments

We report the tests we carried out by running such algorithm with different inputs. No errors were encountered in the outputs that the program returned.

Change	List of coins			
	[200, 100, 50, 20, 10, 5, 2, 1]	[50, 30, 10, 5, 3, 1]	[6, 4, 1]	[9, 6, 5, 1]
1	1	1	1	1
2	2	[1, 1]	[1, 1]	[1, 1]
3	[2, 1]	[3]	[1, 1, 1]	[1, 1, 1]
4	[2, 2]	[3, 1]	[4]	[1, 1, 1, 1]
5	[5]	[5]	[4, 1]	[5]
6	[5, 1]	[5, 1]	[6]	[6]
7	[5, 2]	[5, 1, 1]	[6, 1]	[6, 1]
8	[5, 2, 1]	[5, 3]	[4, 4]	[6, 1, 1]
9	[5, 2, 2]	[5, 3, 1]	[4, 4, 1]	[9]
10	[10]	[10]	[6, 4]	[9, 1]

2.2 Dynamic programming approach

In this section we are going to present the recursive dynamic programming solution.
In order to develop a dynamic programming algorithm, we first need to find a recurrence relation.

Let N be the change and minNumber(N) the minimum number of coins for returning the N. We pose the following question: If we are able to return N with minNumber(N) coins, which change are we able to return with 1 + minNumber(N) coins?
If we have the following list of coins available : $c_1, c_2, c_3, ..., c_n$ and we are able to give back the change N, we are also able to give back: $N - c_1, N - c_2, N - c_3, ..., N - c_n$ (provided that c_i, $_{1 \leq i \leq n}$ is less than or equal to the remaining change to be return).
For instance, if I am able to give back 92 cents and I own the following coins: 2, 5, 10, 50 and 100, I can also give back :

92 - 2 = 90 cent

92 - 5 = 87 cent

92 - 10 = 82 cent

92 - 50 = 42 cent

However, I cannot use a 100 cent coin (1 euro).

Then if $minNumber(N-c_i)_{1<i<n}$ is the minimum number of coins to be returned for the amount $N - c_i$, then $minNumber(N) = 1 + minNumber(N - c_i)$ is the minimum number of coins to be returned for the amount N.

We can therefore set the following recurrence formula:

$$\begin{cases} 0 & \text{if change} = 0 \\ minNumber = 1 + min(minNumber(change - c_i)), \quad 1 \leq i \leq n & \text{if change} > 0 \end{cases} \tag{2.1}$$

The "min" in the recurrence formula expresses the fact that the number of pieces to be returned for amount $N - c_i$ should be the minimum.

2.2.1 Data structure

Our implementation is making use of an array to store the denomination of the coin computed and to determine the number of ways we can return the change. By dynamic programming, we need to figure out how to exploit the previous solutions so that we do not need to recompute them. We have to iterate through the entire array of coins and we also need to check whether the coin is larger than the change amount. So we initialize an array with 0s whose size is "value of change" + 1.

We will iterate through each coin to check how many times this coin can be used to form the value of change.

During the iteration, we take coins one by one, then we test if its value is less than or equal to the index, then compute storeWays[index - current_coins] + storeWays[index], which is the minimum number of ways to make the coin *current_coins* at the index *index*. By repeating such process through all the coins, we will find the desired result at the last cell in the array.

Algorithm 2: function min_coins(coins, change)

```
storeWays : new array;
for i from 0 to change + 1 do
    storeWays[i] = 0;
end
return RLMMO_dyn(coins, change, storeWays)
```

Algorithm 3: function RLMMO_dyn(coins, change, ways)

```
minimum : int;
i : int;
minNumber : int;
if change == 0 then
    return 0 ;
if ways[change] >0 then
    return ways[change] ;
else
    minimum = 9000;
    for ( i = 0, i < size(coins), i++) do
        if coins[i] <= change then
            minNumber = 1 + RLMMO_dyn(coins, change-coins[i], ways);
            if minNumber < minimum then
                minimum = minNumber;
            end
        end
    end
end
return minimum
```

2.2.2 Time complexity analysis

Let's consider that we have n coins to render for the change N. The array needs a size of N to store all the results of the sub-problems.
Analysis of the algorithm :
n = size(coins).
N = change.
If the coin is greater than 1, the inner loop will execute (N - coin) iterations instead of N, here we will consider the worst case.
T(n) = 2 + n*N + 2 $\in O(n * N)$.

We report in the following table the run time and the solution by using as input the change = 31 and the Euro currency.

	[200, 100, 50, 20, 10, 5, 2, 1]
run time (in seconds)	5.159999999992948e-05
solution	3

2.2.3 Experiments

We report the tests we carried out by running the algorithm with the same inputs with which we tested the backtracking program. No errors were encountered in the outputs that the program returned.

Change	List of coins			
	[200, 100, 50, 20, 10, 5, 2, 1]	[50, 30, 10, 5, 3, 1]	[6, 4, 1]	[9, 6, 5, 1]
1	1	1	1	1
2	1	2	2	2
3	2	1	3	3
4	2	2	1	4
5	1	1	2	1
6	2	1	1	1
7	2	3	2	2
8	3	2	2	3
9	3	3	3	1
10	1	1	1	2

2.3 Greedy approach

In this section we present the greedy solution.

The following algorithm is running through the assumption that the set of coins among which it has to pick the coins to return as solution is ordered in a descending way.

Algorithm 4: procedure min_change(change, coin)

```
while change < coin do
    coin = next coin;
end
change = change - coin;
return_coins = return_coins + coin;
if change == 0 then
    return
end
min_coin(change, coin);
```

2.3.1 Time complexity analysis

The aim of this section is to find an order of magnitude for the computational complexity of the Greedy approach.

The analysis has been carried out starting from some considerations that we gained by taking into account the Euro currency. Such reflections are as follows:

* For each coin added to the result we do at most 2 comparisons and at least 1 comparison.
* Because of the descending order on the set of the input coins, not all the coins in such set can be repeated in the solution. Indeed:

- we can have $k, k \geq 0$ repetitions for 2€.
- we can have $c, 0 \leq c \leq 2$ repetitions for 0.20€ and 0.02€. For instance, 3 repetitions of 0.20€ cannot exist since 0.60€ is going to be composed by 0.50€ and 0.10€.
- we cannot have repetitions of all the other coins. For instance, a repetition of 0.10€ is included in one coin of value 0.20€.

From this fact, we could find out the time complexity expressed in terms of the number of comparisons. We took into account the worst scenario which is encountered when all the coins are used and some of them are repeating at their maximum rate (that is 2 coins by 0.02€, 2 coins by 0.2€ and k coins by 2€):

$$\begin{aligned} T(n) &= 2k + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 \\ &= 2k + 2n + 2 \\ &\leq 4n + 2 \in O(n) \end{aligned}$$

For the last inequality, we assumed $k \leq n$ considering that in a real scenario we are going to use also cash to return a change and not only coins.

2.3.2 Experiments

We here report and discuss some results from some experiments we carried out. We chose the sets of coins we used for the previous techniques and we observed the different results after carrying out 12 tests with 12 different changes as input. The results obtained are the following.

Change	List of coins			
	[200, 100, 50, 20, 10, 5, 2, 1]	[50, 30, 10, 5, 3, 1]	[6, 4, 1]	[9, 6, 5, 1]
1	1	1	1	1
2	2	[1, 1]	[1, 1]	[1, 1]
3	[2, 1]	[3]	[1, 1, 1]	[1, 1, 1]
4	[2, 2]	[3, 1]	[4]	[1, 1, 1, 1]
5	[5]	[5]	[4, 1]	[5]
6	[5, 1]	[5, 1]	[6]	[6]
7	[5, 2]	[5, 1, 1]	[6, 1]	[6, 1]
8	[5, 2, 1]	[5, 3]	[6, 1, 1]	[6, 1, 1]
9	[5, 2, 2]	[5, 3, 1]	[6, 1, 1, 1]	[9]
10	[10]	[10]	[6, 4]	[9, 1]
11	[10, 1]	[10, 1]	[6, 4, 1, 1]	[9, 1, 1]
12	[10, 2]	[10, 1, 1]	[6, 6]	[9, 1, 1, 1]

From the results above, we can deduce that the greedy algorithm exhibits optimal solutions only for some set of coins. Indeed, if we pay attention to the three yellow cells, for the 8th test the optimal result should be [4, 4] instead of [6, 1, 1]. The same is observed at the 9th test, where the expected optimal result should be [4, 4, 1], and at the 12th experiment, where we would expect [6,6] instead of [9, 1, 1, 1].
In the following paragraph, we try to give an explanation about this behaviour.

Particular set of coins

After carrying out different tests with several list of coins, we could observe that the wrong results might occur only for set of coins larger than or equal to

3 elements. In particular, we could find out some properties of the sets of coins that do not make the algorithm fail. These properties come from the followings considerations:

1. A set composed of at most 3 coins whose values go from 1 to 5 make the algorithm return right solutions. For instance, as observed from the experiments, the third set [6, 4 , 1] fails with change = 8 and change = 9.

2. The algorithm succeeds if the set has length larger than 3 and if it can be split into two or more subsets such that:

   * each subset has at most three elements

   * we can form a subset such as its values are the smallest and they go from 1 to 5.

   * each value in the position i in a subset can be expressed in terms of the value in the same position i in the subset containing the smallest elements by multiplying some constant multiple of 10.

   As counterexample to the (2) property let us consider the set [6, 5, 4, 1]. Such set does not obey to the second property and indeed it fails with the change 9: it would return [6, 1, 1, 1] instead of [5, 4].

Hence, the properties can be written formally as follows:
Let S be the set of coins such that $S = S_1 + ... + S_n$, with $|S| > 3$, $S_i \subset S$. S generates a right solutions if and only if:

* $|S_i| \leq 3 \ \forall \ 1 \leq i \leq n$

* $s_i \in S_i$, $1 \leq s_i \leq 5 \ \forall \ 1 \leq i \leq 3$

* $s_j i = c * s_1 i \ \forall \ 2 \leq i \leq n$ *with* $c \ mod \ 10 = 0$ *and* $s_j i$ *the element in the*

set S_j at index i .

where for *c mod 10 = 0* we mean that the rest from the division between c and 10 has to be 0, so c is multiple of 10.

For instance, if we consider the second set of coins $S = [50, 30, 10, 5, 3, 1]$, we can see that it is constituted by two subsets: $S_1 = [50, 30, 10]$ and $S_2 = [5, 3, 1]$, such that $S = S_1 \cup S_2$ and $S_1 = S_2 * 10$.
As another example, if we take into consideration the Euro currency, we have:

$$\begin{aligned}[200, 100, 50, 20, 10, 5, 2, 1] &= [200, 100, 50, 20, 10, 5, 2, 1] \\ &= [200, 100] \cup [50, 20, 10] \cup [5, 2, 1] \\ &= [2 * 100, 1 * 100] \cup [5 * 10, 2 * 10, 1 * 10] \cup [5, 2, 1]\end{aligned}$$

In section 4.3 in the appendix, it is reported the 6 combinations of the elementary sets composed of three elements respecting the properties above and some sets built starting from those sets.

2.4 Comparison of the performance

We now present a comparison in terms of the time performance among all the programming approaches illustrated in the previous sections. The tests have been carried out by selecting the Euro currency, that is [200, 100, 50, 20, 10, 5, 2, 1].

All the results are expressed in terms of seconds.

Change	Technique		
	Backtrack	Dynamic programming	Greedy
1	0.000148	1.080000e-05	3.899999e-06
2	7.409999e-05	2.100000e-05	2.399999e-06
3	0.000282	7.100000e-06	3.099999e-06
4	0.000122	8.800000e-06	2.600000e-06
5	7.560000e-05	1.939999e-05	1.899999e-06
6	0.001576	1.250000e-05	2.499999e-06
7	0.001744	1.210000e-05	2.300000e-06
8	0.003432	3.969999e-05	2.799999e-06
9	0.002451	2.030000e-05	3.900000e-06
10	0.000864	2.070000e-05	1.899999e-06

It is possible to notice that the slowest algorithm is the one exploiting the backtrack approach, while the fastest program resulted to be Greedy. Such results were expected since the time complexity analysis done previously for each technique.

2.5 Divide and conquer approach overview

The coin change problem can be solved also by exploiting the Divide and Conquer approach. The recursive relationship would be as follows, where the coinSet refers to the initial set of coins and N the change to return.

$$MinCoins(N) = \begin{cases} min[(MinCoins(N - coin)] + coin, & \text{if } N \notin coinSet \\ N, & \text{otherwise} \end{cases} \tag{2.2}$$

where '*min[(MinCoins (N - coin)] + coin*' is meant $\forall coin \in coinSet$.
Hence the subproblems are the subchange (N - coin). For each subproblem we compute the optimal solution (the shortest list of coins) and we return it. Our final optimal solution would then result to be the concatenation between the optimal solutions for all the subproblems computed and the coin that led to the generation of those optimal subsolutions.
The algorithm can be expressed by the pseudocode illustrated in the next page where we highlighted the divide and conquer step. In the divide step the algorithm is solving the coin change problem for all the subproblems (change - coin). Instead in the conquer step, it is picking the optimal solution associated to the subproblem just computed and then it is returning it.

Algorithm 5: procedure MinCoins(change, coinSet)

```
if change ∈ coinSet then
    return change ;
else
    —- DIVIDE step —-
    for all the coins ∈ coinSet do
        coin_list = [ empty list ];
        if change ≤ coin then
            coin_list = coin + MinCoins(change - coin, coinSet);
        end
    end
    —- CONQUER step —-
    n = len(coinSet);
    best_coin_list = min(coin_list_1, ..., coin_list_n );
    return best_coin_list;
end
```

Chapter 3

Conclusion

In the report we could have seen different approaches to solve the coin change problem.
In designing the backtrack and dynamic programming algorithm, we could find out that the most tricky part is to set the recursion. In particular, for the backtrack procedure since the bad time complexity, we had to focus on coming up with a good pruning condition in order to observe some results without waiting too long. Indeed, several pruning conditions could be found and it is hence important to set the best one in order to make the computation as much fast as possible. For instance, a different pruning condition for the backtrack program could be to check that the last item inserted in the solutions is smaller than the previous one. This would avoid to compute some combinations already computed previously.

From this work we can conclude that for solving the coin change problem the best solution to exploit depends on our input set of coins. Indeed, the best algorithm paradigm in terms of time complexity resulted to be Greedy since

it has a linear complexity, but it is returning a correct solution only for some particular type of set of coins. On the other hand, the best procedure whose correctness is not depending on the input set is the dynamic programming (polynomial complexity). However, such last approach neglects the memory in order to improve the run time.

As we can see each algorithm has its pro and cons, and hence we should use them in scenario that let us take advantage of their pros such as the use of only Euro currency would allow us to exploit the greedy solution. Otherwise, we have to choose between performance and optimality.

Chapter 4

Appendix

4.1 Execution backtrack algorithm

Figure 4.1: Combinations of the best solutions computed

```
START BACKTRACKING PROGRAM

Set of coins:  [200, 100, 50, 20, 10, 5, 2, 1]

Combination # 1   [100, 50, 20, 20]
Combination # 2   [100, 20, 50, 20]
Combination # 3   [100, 20, 20, 50]
Combination # 4   [50, 100, 20, 20]
Combination # 5   [50, 20, 100, 20]
Combination # 6   [50, 20, 20, 100]
Combination # 7   [20, 100, 50, 20]
Combination # 8   [20, 100, 20, 50]
Combination # 9   [20, 50, 100, 20]
Combination # 10   [20, 50, 20, 100]
Combination # 11   [20, 20, 100, 50]
Combination # 12   [20, 20, 50, 100]

The best way to give back the change:  190  is:  [20, 20, 50, 100]
END program
```

4.2 Different backtracking implementation

Here it is a different backtracking implementation for the coin change problem. This results to be better than the one presented in the section 2.1. The improvements has been achieved by taking advantage of the descending order of the set of coins and using a pruning condition.

Algorithm 6: procedure min_change(change, return_coins)

```
for coin in the list of coins that is worth a visit do
    change = change - coin;
    return_coins = return_coins + coin;
    if change > 0 then
        if change == 0 then
            End recursion: Solution found! ;
        end
        min_change(change, return_coins);
    end
    change = change + coin;
    return_coins = return_coins - coin;
end
```

4.2.1 Optimality property

By setting a descending order over the set of coins, the algorithm is picking always the largest coin for building the solution. This property ensures that the first solution found is the shortest one and so the best one.

4.2.2 Time complexity analysis

We are now going to analyse the algorithm to give an order of time complexity.

The first consideration to be done regards the **pruning condition**. In the 'for' loop we set a condition that needs to be satisfied in order to let the recursive tree expands. For implementing the loop condition it is possible to adopt a global Boolean variable that is initialize 'True' at each recursive call and is assigned the value 'False' only when the solution is found. This strategy let us stop the generation of the recursive tree and hence avoid to check nodes (hence coins) that we already know are not going to build an optimal solution. Indeed without such condition we are going to build all the possible solutions and so generating all the combinations of coins we can use to give back the change. This would result in a really expensive computational effort and lead to a $O(n^k)$, with n = *cardinality of the input coins set* and k = *change in a no-decimal type* (e.g. change = 1.90€ -> k = 190).

By using the pruning condition, we could obtain a backtrack algorithm running in an order of time $O(n^2)$. This estimations can be achieved by doing the following considerations focusing on the number of comparisons done for each interaction:

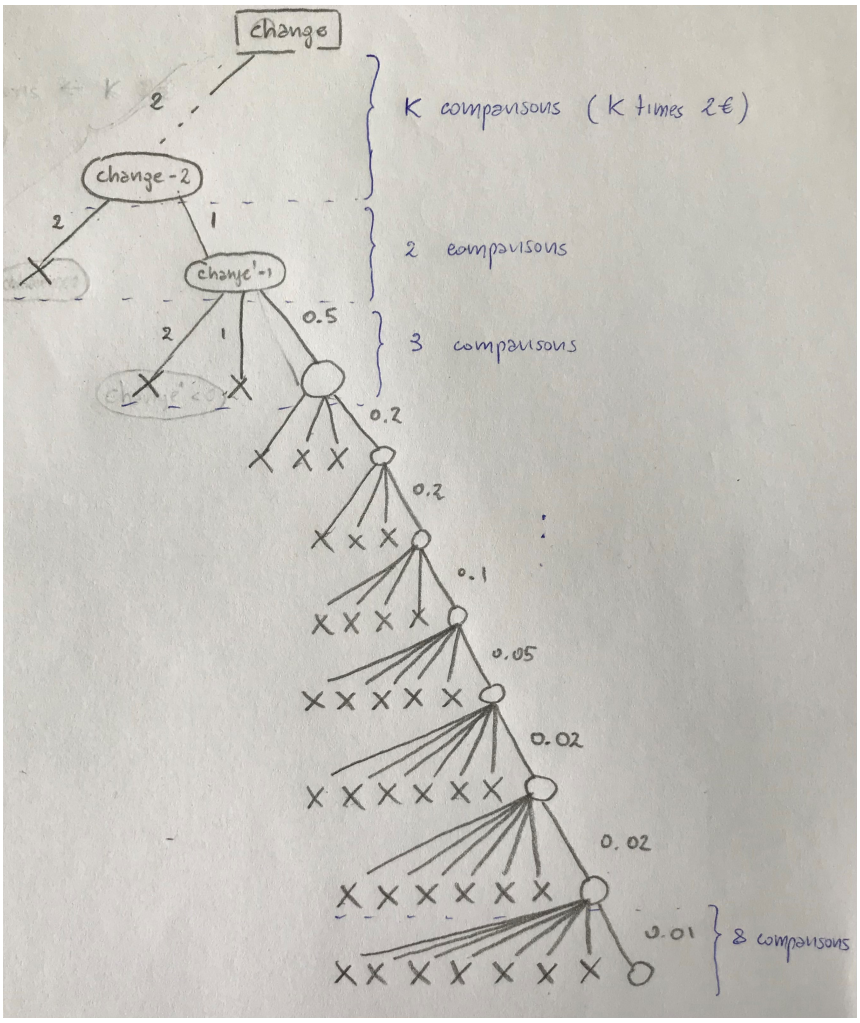
* For each coin added to the result we do at most 2 comparisons: the 'satisfaisant' and 'soltrouvée' conditions; and at least 1 comparison: the 'satisfaisant'.
* Because of the descending order on the set of the input coins, not all the coins in such set can be repeated in the solution.
  - we can have $k, k \geq 0$ repetitions for 2€.

- we can have $c, 0 \leq c \leq 2$ repetitions for 0.20€ and 0.02€.
- we cannot have repetitions of all the other coins. For instance, a repetition of 0.10€ is included in one coin of value 0.20€.

Hence, in the worst scenario (Figure 4.2) we are checking all the coins and we have k 2€ and $c = 2$ both for 0.20€ and 0.02€. In the figure 4.1 in the appendix, we report the drawing of the recursive tree related to this case.
Hence, taking into account the consideration above, we can determine a time recurrence for such case. In the following recurrence, n is meant to be the size of the list coins inputted.

$$\begin{aligned} T(n) &= k + 2 + 3 + 4 + 4 + 5 + 6 + 7 + 7 + 8 \\ &= k - 1 + 1 + 2 + 3 + 4 + 4 + 5 + 6 + 7 + 7 + 8 \\ &= k - 1 + \sum_{n=1}^{n}(i) + 4 + 7 \\ &= k + \frac{n(n+1)}{2} + 10 \in O(n^2) \end{aligned}$$

Figure 4.2: Recursive tree for the worst case for the optimized backtrack algorithm



4.3 Particular sets for the Greedy approach

In the following table we present the elementary sets that let the greedy program return the right solution.

[1, 2, 5]
[1, 2, 4]
[1, 2, 3]
[1, 3, 5]
[1, 3, 4]
[1, 4, 5]

In the following table we present some sets built starting from the elementary sets that respect the properties explained in the greedy section and hence let the greedy program return the right solution.

Elementary sets	New sets
[1, 2, 5]	[1, 2, 5, 10, 20, 50]
[1, 2, 4]	[1, 2, 4, 10, 20, 40, 100, 200]
[1, 2, 3]	[10, 20, 30, 100, 200, 300]
[1, 3, 5]	[1, 3, 5, 10, 30]
[1, 3, 4]	[1, 3, 4, 10, 30, 40]
[1, 4, 5]	[1, 4, 5, 10, 40, 50]

4.4 Coding implementation

Here we illustrate the code implementation done by using Python as programming language.

4.4.1 Backtrack algorithm

```
#****************************************
#*********** BACKTRACKING ***************
#****************************************

# Function to find the minimum list of coins to get a
   change of `N` from an limited supply of coins in set `
   list_coins`
def bkOpt(change, return_coins):
    global best_return_coins
    global size
    global list_coins
    global N
    global N_combination

    for i in range(len(list_coins)):
        curr_coin = list_coins[i]
        return_coins.append(curr_coin)
        change = change - curr_coin

        #If element can lead to a best solution
        if change >= 0 and len(return_coins) <= size:
```

```
            #candidate solution found
            if change == 0:
                #update best solutions
                best_return_coins = return_coins[:]
                size = len(best_return_coins)

            else:
                bkOpt(change, return_coins)

        #backtrack step
        return_coins.pop()
        change = change + curr_coin
```

4.4.2 Dynamic programming algorithm

```
#*******************************************
#********** DYNAMIC PROGRAMMING ************
#*******************************************

def NBP(S,N):
# initialization of the array
  store = [0]*(N+1)

# start dynamic programming algorithm
  return RLMMO_dyn(S, N, store)

def RLMMO_dyn(S, N, storeMin):

# If change = 0
  if N==0:
    # No coins to give back
    return 0

# if the last item in the array is positive
  elif storeMin[N]>0:
    # return it
    return storeMin[N]

# else search for the miminum coins
  else:
  # init minimum
    minimum = 9000
```

```
    for i in range(len(S)):

    # if a new ways is found
      if S[i]<=N:
        #increment number of ways (combinations)
        nbWays= 1 + RLMMO_dyn(S, N-S[i], storeMin)

        # if it is the best one
        if nbWays<minimum:
        # update minimum
          minimum = nbWays
          storeMin[N] = minimum

  return minimum
```

4.4.3 Greedy algorithm

```
#****************************************
#**************** GREEDY ****************
#****************************************

# Function to find the the best way to get a change of N
def greedy(change, start, return_coins):
    global list_coins

    #find the next candidates for the next choice
    while (change < list_coins[start]):
      start = start + 1

    #greedy choice
    coin = list_coins[start]
    return_coins.append(coin)

    change = change - coin

    if change == 0:
        return #stop recursion

    greedy(change, start, return_coins)
```