Exercise 21: Loose Change

Andreas Loibl

November 10, 2010

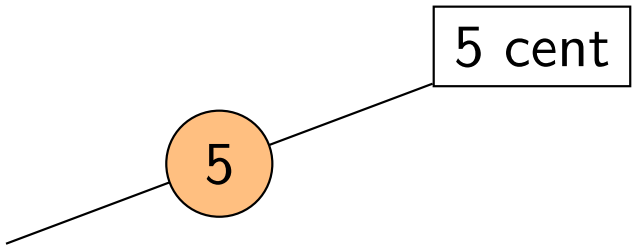
Inhaltsverzeichnis

Algorithm

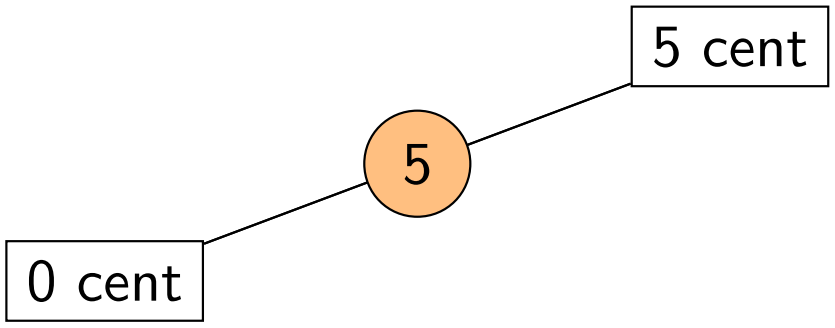
Example tree

5 cent

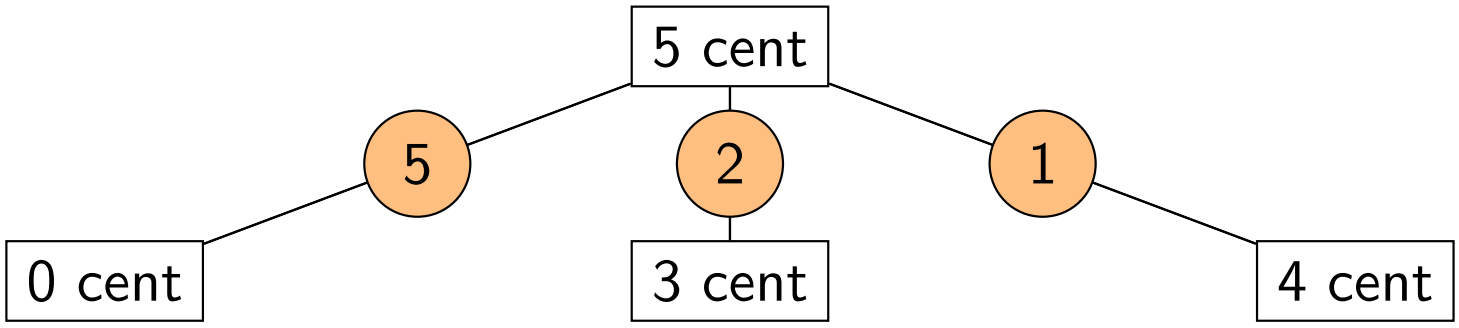
Example tree



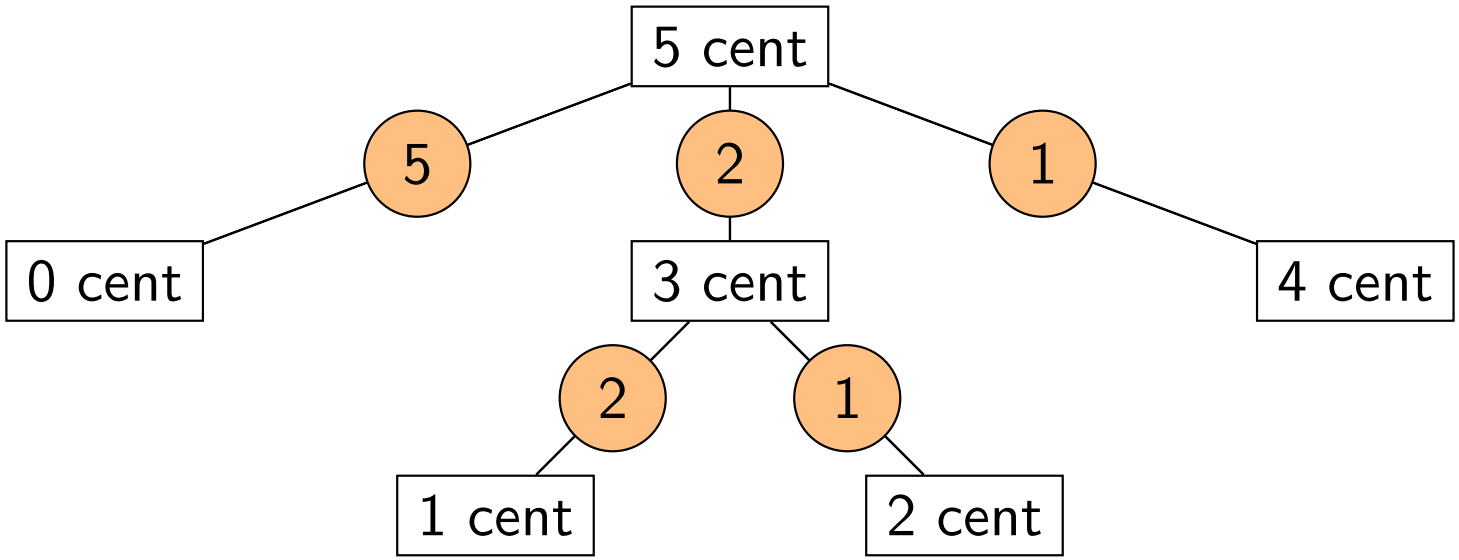
Example tree



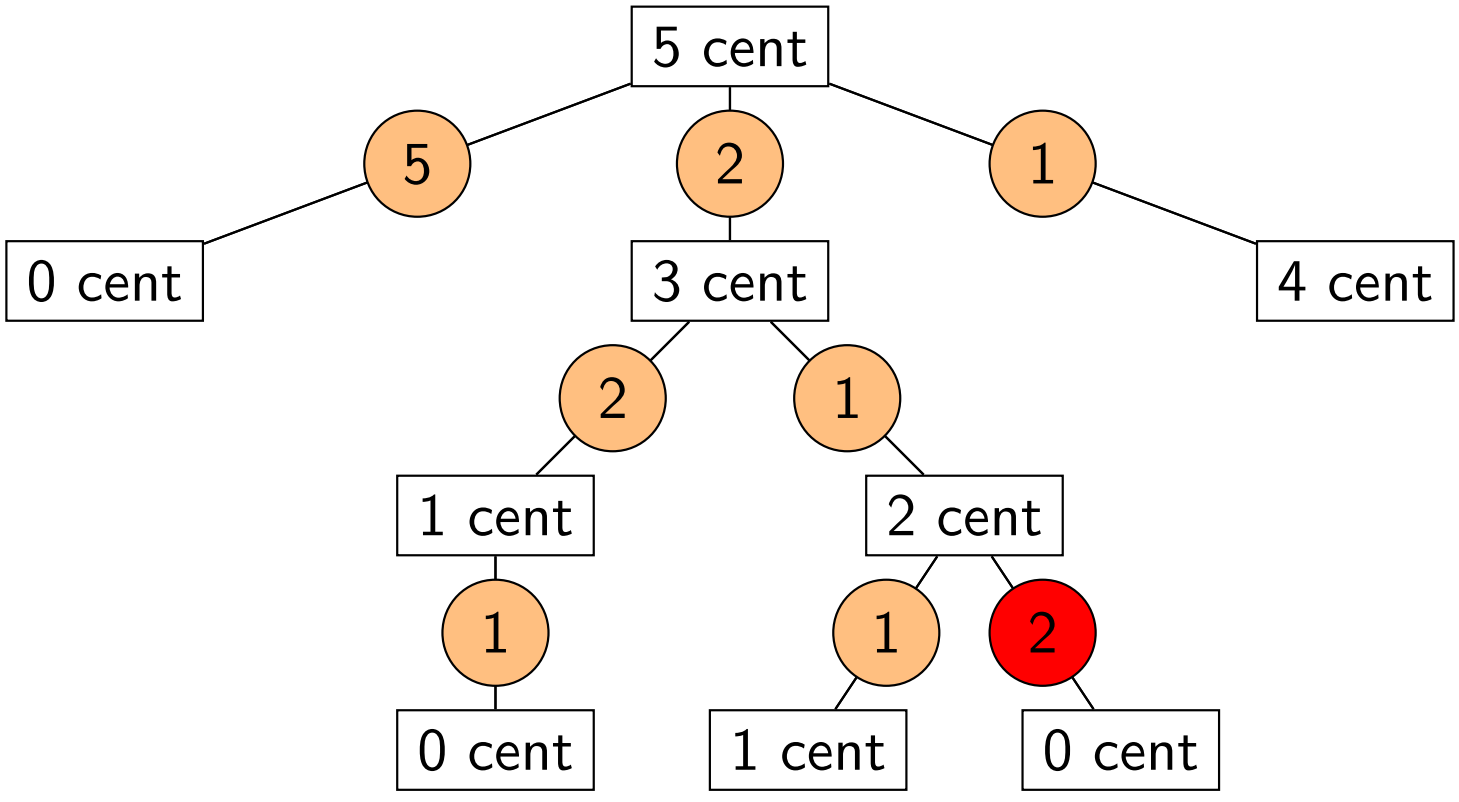
Example tree



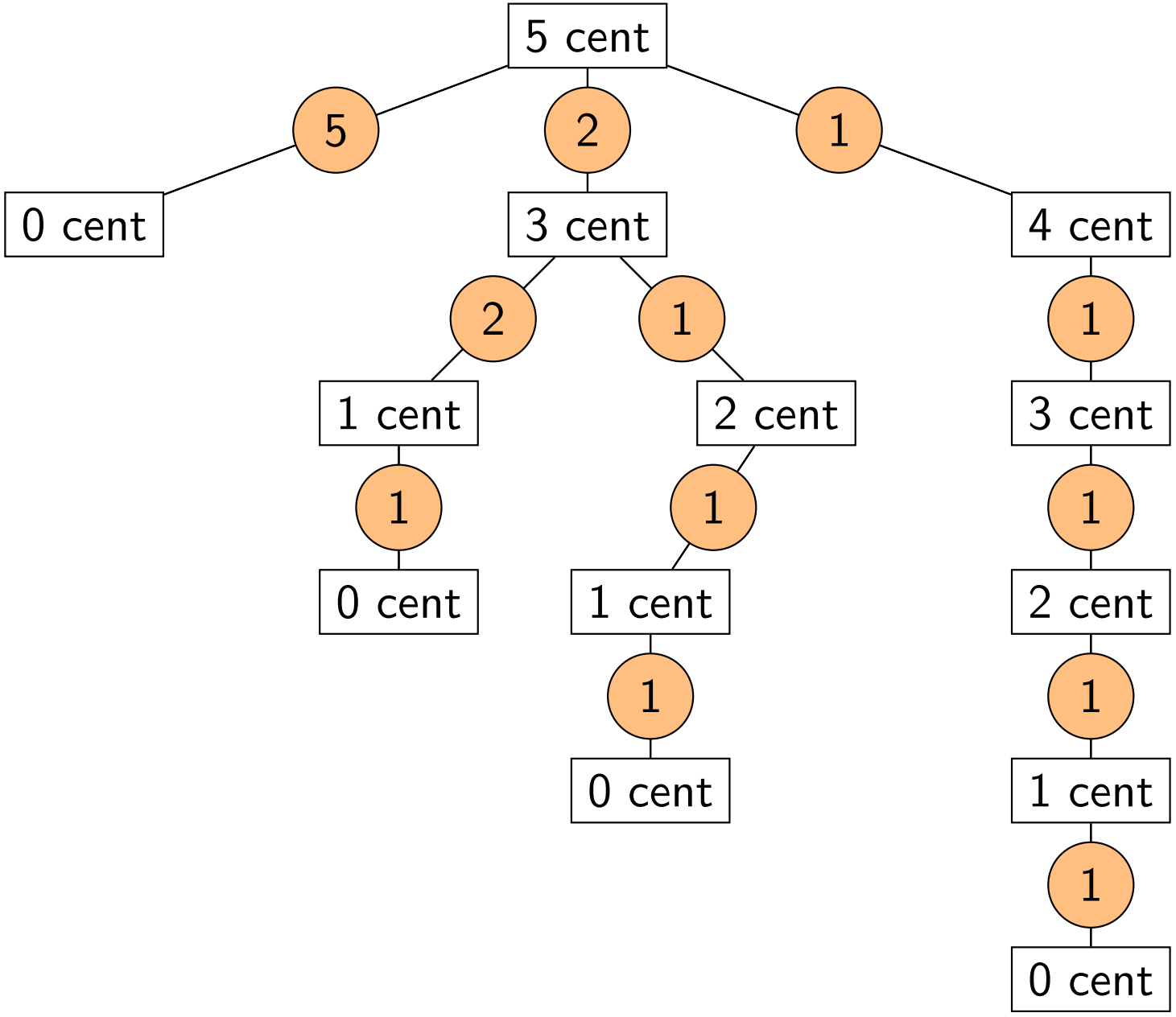
Example tree



Example tree



Example tree



in Pseudocode

function SPLITMONEY(*cents*, *lastcoin*, *path*)
    if *cents* = 0 **then** ▷ termination condition
        output *path*
    else
        for *coin* ∈ {200, 100, 50, 20, 10, 5, 2, 1} **do**
            if *cents* ≥ *coin* ∧ *lastcoin* ≥ *coin* **then**
                newpath ← *path*
                append *coin* to *newpath*
                newcents ← *cents* − *coin*
                SPLITMONEY(*newcents*, *coin*, *newpath*) ▷ recursion
            end if
        end for
    end if
end function

in Pseudocode

function SPLITMONEY(*cents*, *lastcoin*, *path*)
    if *cents* = 0 **then** ▷ termination condition
        output *path*
    else
        for *coin* ∈ {200, 100, 50, 20, 10, 5, 2, 1} **do**
            if *cents* ≥ *coin* ∧ *lastcoin* ≥ *coin* **then**
                newpath ← *path*
                append *coin* to *newpath*
                newcents ← *cents* − *coin*
                SPLITMONEY(*newcents*, *coin*, *newpath*) ▷ recursion
            end if
        end for
    end if
end function

in Pseudocode

function SPLITMONEY(*cents*, *lastcoin*, *path*)
    if *cents* = 0 **then** ▷ termination condition
        output *path*
    else
        for *coin* ∈ {200, 100, 50, 20, 10, 5, 2, 1} **do**
            if *cents* ≥ *coin* ∧ *lastcoin* ≥ *coin* **then**
                newpath ← *path*
                append *coin* to *newpath*
                newcents ← *cents* − *coin*
                SPLITMONEY(*newcents*, *coin*, *newpath*) ▷ recursion
            end if
        end for
    end if
end function

in Pseudocode

function SPLITMONEY(*cents*, *lastcoin*, *path*)
    if *cents* = 0 **then** ▷ termination condition
        output *path*
    else
        for *coin* ∈ {200, 100, 50, 20, 10, 5, 2, 1} **do**
            if *cents* ≥ *coin* ∧ *lastcoin* ≥ *coin* **then**
                newpath ← *path*
                append *coin* to *newpath*
                newcents ← *cents* − *coin*
                SPLITMONEY(*newcents*, *coin*, *newpath*) ▷ recursion
            end if
        end for
    end if
end function

in Pseudocode

function SPLITMONEY(*cents*, *lastcoin*, *path*)
    if *cents* = 0 **then** ▷ termination condition
        output *path*
    else
        for *coin* ∈ {200, 100, 50, 20, 10, 5, 2, 1} **do**
            if *cents* ≥ *coin* ∧ *lastcoin* ≥ *coin* **then**
                newpath ← *path*
                append *coin* to *newpath*
                newcents ← *cents* − *coin*
                SPLITMONEY(*newcents*, *coin*, *newpath*) ▷ recursion
            end if
        end for
    end if
end function

in Pseudocode

function SPLITMONEY(*cents*, *lastcoin*, *path*)
    if *cents* = 0 **then** ▷ termination condition
        output *path*
    else
        for *coin* ∈ {200, 100, 50, 20, 10, 5, 2, 1} **do**
            if *cents* ≥ *coin* ∧ *lastcoin* ≥ *coin* **then**
                newpath ← *path*
                append *coin* to *newpath*
                newcents ← *cents* − *coin*
                SPLITMONEY(*newcents*, *coin*, *newpath*) ▷ recursion
            end if
        end for
    end if
end function

in R code

```
split_money <- function(cents, lastcoin=cents, path=
    NULL)
{

  if(cents == 0)
    print(path)
  else
  for(coin in c(200, 100, 50, 20, 10, 5, 2, 1))
  {
    if(cents >= coin && lastcoin >= coin)
    {
      split_money(cents-coin, coin, c(path, coin))

    }

  }

}
```

in R code (returning a list)

```
split_money <- function(cents, lastcoin=cents, path=
    NULL)
{
  res <- list(NULL); res[[1]] <- NULL
  if(cents == 0)
    return(list(path))

  for(coin in c(200, 100, 50, 20, 10, 5, 2, 1))
  {
    if(cents >= coin && lastcoin >= coin)
    {
      for(sequence in split_money(cents-coin, coin,
          c(path, coin)))
        res[[length(res)+1]] <- sequence
    }
  }
  return(res)
}
```

format list more human readable (collapsed)

```
format_output <- function(data)
{
  res <- list(NULL); res[[1]] <- NULL
  for(coins in data)
  {
    output <- character()
    coins.rle <- rle(coins)
    coins.data <- data.frame(coin = coins.rle$values, count = coins.rle$lengths)
    for(i in 1:length(coins.data$count))
      output <- c(output, paste(c(coins.data$coin[i], coins.data$count[i]),
          collapse=" cent x "))
    res[[length(res)+1]] <- output
  }
  return(res)
}
```

output example

"5 cent x 3" "2 cent x 2" "1 cent x 1" instead of "5 5 5 2 2 1"

Answer

Question

How many different ways can €2.50 be made using any number of coins?

```
> source("ex21_loose_change.R")
> a <- split_money(250)
> length(a)
[1] 200187
```

Answer

There are **200187** different ways €2.50 can be made of coins.

Performance

- So you see that this algorithm works, but you should have a look at the performance
- If you try to compute the number this way you will notice how slow it is and how long it takes to finish
- On this machine[1] it took almost **12 minutes**
- This bad performance is caused by the recursion which calls itself very often (exponentially!)
- In order to speed up the calculation I rewrote the algorithm:
  - Step 1: only calculate the number of possible ways, do not create a list of them
  - Step 2: replace for-loop with R's apply-function
  - Step 3: add a cache for the calculated values (very important!)

[1] ThinkPad X200 with Intel Core2 Duo P8600

- So you see that this algorithm works, but you should have a look at the performance
- If you try to compute the number this way you will notice how slow it is and how long it takes to finish
- On this machine[1] it took almost **12 minutes**
- This bad performance is caused by the recursion which calls itself very often (exponentially!)
- In order to speed up the calculation I rewrote the algorithm:
  - Step 1: only calculate the number of possible ways, do not create a list of them
  - Step 2: replace for-loop with R's apply-function
  - Step 3: add a cache for the calculated values (very important!)

[1] ThinkPad X200 with Intel Core2 Duo P8600

- So you see that this algorithm works, but you should have a look at the performance
- If you try to compute the number this way you will notice how slow it is and how long it takes to finish
- On this machine[1] it took almost **12 minutes**
- This bad performance is caused by the recursion which calls itself very often (exponentially!)
- In order to speed up the calculation I rewrote the algorithm:
  - Step 1: only calculate the number of possible ways, do not create a list of them
  - Step 2: replace for-loop with R's apply-function
  - Step 3: add a cache for the calculated values (very important!)

[1] ThinkPad X200 with Intel Core2 Duo P8600

- So you see that this algorithm works, but you should have a look at the performance
- If you try to compute the number this way you will notice how slow it is and how long it takes to finish
- On this machine[1] it took almost **12 minutes**
- This bad performance is caused by the recursion which calls itself very often (exponentially!)
- In order to speed up the calculation I rewrote the algorithm:
  - Step 1: only calculate the number of possible ways, do not create a list of them
  - Step 2: replace for-loop with R's apply-function
  - Step 3: add a cache for the calculated values (very important!)

[1] ThinkPad X200 with Intel Core2 Duo P8600

- So you see that this algorithm works, but you should have a look at the performance
- If you try to compute the number this way you will notice how slow it is and how long it takes to finish
- On this machine[1] it took almost **12 minutes**
- This bad performance is caused by the recursion which calls itself very often (exponentially!)
- In order to speed up the calculation I rewrote the algorithm:
  - Step 1: only calculate the number of possible ways, do not create a list of them
  - Step 2: replace for-loop with R's apply-function
  - Step 3: add a cache for the calculated values (very important!)

[1] ThinkPad X200 with Intel Core2 Duo P8600

Step 1 & 2

```
coins.available <- c(200, 100, 50, 20, 10, 5, 2, 1)

ways_to_split_money <- function(cents,lastcoin=0)
{
  if(!lastcoin)
  {
    lastcoin <- cents
  }
  if(cents < 2) return(1)
  coins <- coins.available
  coins <- coins[which(cents>=coins)]
  coins <- coins[which(lastcoin>=coins)]
  return(sum(sapply(coins, function(coin) ways_to_split_money(cents-coin, coin))
      ))
}
```

Step 3: Caching

```
coins.available <- c(200, 100, 50, 20, 10, 5, 2, 1)

ways_to_split_money <- function(cents,lastcoin=0)
{
  if(!lastcoin)
  {
    cache <<- matrix(0,cents,length(coins.available))
    lastcoin <- cents
  }
  use_cache <- function(cents,lastcoin,value=0)
  {
    if(! lastcoin || ! lastcoin %in% coins.available) return(value)
    if(value) cache[cents,which(coins.available==lastcoin)] <<- value
    return(cache[cents,which(coins.available==lastcoin)])
  }
  if(cents < 2) return(1)
  if(use_cache(cents,lastcoin)>0) return(use_cache(cents,lastcoin))
  coins <- coins.available
  coins <- coins[which(cents>=coins)]
  coins <- coins[which(lastcoin>=coins)]
  return(
    use_cache(cents,lastcoin,
    sum(sapply(coins, function(coin) ways_to_split_money(cents-coin, coin)))
    ))
}
```

Step 3: Caching

```
> cache
        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]
  [1,]     0     0     0     0     0     0     0     0
  [2,]     0     0     0     0     0     0     2     1
  [3,]     0     0     0     0     0     0     2     1
  [4,]     0     0     0     0     0     0     3     1
  [5,]     0     0     0     0     0     4     3     1
  [6,]     0     0     0     0     0     0     4     1
  [7,]     0     0     0     0     0     0     4     1
  [8,]     0     0     0     0     0     0     5     1
  [9,]     0     0     0     0     0     0     5     1
 [10,]     0     0     0    11    11    10     6     1
 [11,]     0     0     0     0     0     0     6     1
 [12,]     0     0     0     0     0     0     7     1
 [13,]     0     0     0     0     0     0     7     1
 [14,]     0     0     0     0     0     0     8     1
 [15,]     0     0     0     0     0    18     8     1
 [16,]     0     0     0     0     0     0     9     1
 [17,]     0     0     0     0     0     0     9     1
 [18,]     0     0     0     0     0     0    10     1
 [19,]     0     0     0     0     0     0    10     1
 [20,]     0     0     0    41    40    29    11     1
 (...)
 [50,]   451   451   451   450   341   146    26     1
 (...)
[250,]     0     0     0     0     0     0     0     0
```

```
$ time ./ex21_loose_change.R 250  # 2.50 EUR in coins
[1] 200187
real    0m0.269s
$ time ./ex21_loose_change.R 1000  #  10 EUR in coins
[1] 321335886
real    0m0.598s
$ time ./ex21_loose_change.R 10000 # 100 EUR in coins
[1] 1.133873e+15
real    0m4.538s
$ time ./ex21_loose_change.R 50000 # 500 EUR in coins
[1] 7.963173e+19
real    0m22.145s
```

Result

Huge speed improvement by implementing the cache