Problem 1

HW 4

Hanyuan Chi(chixx105), Zhi Shen(shenx704)

February 25, 2017

```
suppressPackageStartupMessages({
  library(microbenchmark)
  library(memoise)
  library(assertthat)
  library(ggplot2)
  library(dplyr)
})
```

Dynamic Programming

Assume our local post office in Minneapolis sells stamps in three different denominations: 1 cent, 7 cents, and 10 cents. Design a dynamic programming algorithm that will find the minimum number of stamps necessary to pay N cents of postage fee.

General Requirements

- `N = 37L` is the desired problem size. Feel free to experiment with it but make sure you produce a submission with `N >= 37L` for bot feedback purposes.
- Please review the resulting PDF and make sure that all code fits into the page. If you have lines of code that run outside of the page limits we will deduct points for incorrect formatting as it makes it unnecessarily hard to grade.
- Please make sure your PDF can be knitted in *at most* 3 minutes.

Question 1

Think about the Dynamic Programming formulation for that problem.

- **Hints**:
  - Consider subproblems $S(n)$ where $S(n)$ is the minimum number of stamps needed to pay n cent fee.
  - Greedy approach is not a correct approach here just adding the biggests stamps first and then going for smaller stamps will not always produce an optimal solution in general.
  - Since the largest denomination is 10 cents, feel free to pre-compute S manually for subproblems of sizes up to 10. This is considered an acceptable practice in DP solutions.

Please describe the logic of your dynamic programming solution here: If the number of stamps necessary to pay N cents of postage fee is optimally minimized, then you must have optimally decided which denotation of stamps to pick everytime you pick a stamp, 1 cent, 7 cents or 10 cents. If you decide to pick 1 cent this time, it means the number of stamps necessary to pay (N-1) cents of postage fee has also been minimized, so the minimun number of stamps for N cents is S(n-1)+1.

If you decide to pick 7 cents this time, it means the number of stamps necessary to pay (N-7) cents of postage fee has also been minimized, so the minimun number of stamps for N cents is S(n-7)+1. If you decide to pick 10 cents this time, it means the number of stamps necessary to pay (N-10) cents of postage fee has also been minimized, so the minimun number of stamps for N cents is S(n-10)+1. When you pick a stamp this time, among 1 cent, 7 cents and 10 cents, you pick the one that gives you the minimum number of stamps, that is min(S(n-1)+1,S(n-7)+1,S(n-10)+1).

```
# N is the desired problem size
# Feel free to experiment with it
# but make sure you produce a final submission with N >= 37L
N <- 37L
```

Question 2

Solve the problem using naive recursive approach

- **Hint**:
  - think carefully about the initial conditions for S
- Please produce a function named `S`

```
# Please write your code here
S <- function(n) {
  if (!is.integer(n)) stop("n must be an integer")
  if (n==0L) return (0L)
  if (n>0L && n<7L) return (S(n-1L)+1L)
  if (n>=7L && n<10L) return (min(S(n-1L)+1L, S(n-7L)+1L))

  pick_one <- S(n-1L)+1L
  pick_seven <- S(n-7L)+1L
  pick_ten <- S(n-10L)+1L

  (min(pick_one, pick_seven, pick_ten))
}

S(N)
```

```
## [1] 4
```

Question 3

Add memoisation and solve the problem using the Dynamic Programming top-down approach.

- Please produce a function named `mS`

```
# Please write your code here
mS <- memoise(function(n) {
  if (!is.integer(n)) stop("n must be an integer")
  if (n==0L) return (0L)
  if (n>0L && n<7L) return (mS(n-1L)+1L)
  if (n>=7L && n<10L) return (min(mS(n-1L)+1L, mS(n-7L)+1L))

  pick_one <- mS(n-1L)+1L
  pick_seven <- mS(n-7L)+1L
  pick_ten <- mS(n-10L)+1L
```

```
  (min(pick_one, pick_seven, pick_ten))
})

mS(N)
```

```
## [1] 4
```

Unit tests for Question 3

Let's take a look at the concept of a "unit test" in software engineering. The idea of the unit test is to confirm that a unit of our code (in this case, function `mS`) works as expected on particular inputs that we verified manually. I recommend using `assertthat` package in R. It offers a collection of functions like `validate_that()`, `see_if()` and `assert_that()` that are of great help when design unit tests as below.

For the purposes of this HW, I picked `validate_that()` instead of `assert_that()` since `assert_that()` will return an ERROR if assertion failed and thus, will not even let you produce the PDF until you fix the mistake. Instead, `validate_that()` will be nicer and will just report a message while still letting you knit the file.

Clearly, before you actually implement `mS` your unit tests will fail. But once you implement everything correctly they should all return `TRUE`.

```
validate_that(mS(10L) == 1L) # only 1 stamp is required to pay 10 cents
validate_that(mS(14L) == 2L) # only 2 stamps are required to pay 14 cents
validate_that(mS(16L) == 4L) # only 4 stamps are required to pay 16 cents
validate_that(mS(17L) == 2L) # only 2 stamps are required to pay 17 cents
validate_that(mS(20L) == 2L) # only 2 stamps are required to pay 20 cents
validate_that(mS(24L) == 3L) # only 3 stamps are required to pay 24 cents
# Feel free to add a couple more unit tests but don't erase the tests above
```

```
## [1] TRUE
## [1] TRUE
## [1] TRUE
## [1] TRUE
## [1] TRUE
## [1] TRUE
```

Question 4

Time the two solutions. Verify that memoisation indeed brings you the promised performance benefits.

- **Hint:**
  - use `microbenchmark(S(N), {forget(mS); mS(N)}, times = 10L)`
  - we use `times=10L` instead of the default `times=100L` to make sure the estimation is repeated 10 times. It is faster but less precise.
  - make sure you `forget()` the cached version first. Otherwise, your benchmarking is not doing testing it correctly

```
forget(mS) # Erase the cache first
```

```
## [1] TRUE
```

```
# Please write your code here
microbenchmark(S(N), {forget(mS); mS(N)},times = 10L )
```

```
## Unit: milliseconds
```

```
##                          expr       min        lq      mean    median
##                          S(N) 82.500232 83.781512 93.035225 85.411378
##  {     forget(mS)     mS(N) }  5.506132  5.769385  8.019709  7.569278
##          uq       max neval cld
##  97.849559 134.42938    10   b
##   9.555623  12.11221    10  a
```

Question 5

Re-implement the problem using bottom-up approach.

- Please produce an integer array `S_` that contains the solutions to all the subproblems up to N.

```
# Please write your code here
S_ <- vector("numeric", length=N)

# For bottom-up approach you need to fill the array S_ with your solutions
# Here is the bottom-up loop
for(i in 1:N) {
  S_[i] <- 0L
  S_[1] <- 1L
  S_[7] <- 1L
  S_[10] <- 1L

  if (i>1L && i<7L) (S_[i] <- S_[i-1L]+1L)
  if (i>7L && i<10L) (S_[i] <- min(S_[i-1L]+1L, S_[i-7L]+1L))
  if (i>10L) (S_[i] <- min(S_[i-10L]+1L, S_[i-7L]+1L, S_[i-1L]+1L))
}

# Finally, display the result
S_[N]
```

```
## [1] 4
```

Unit tests for Question 5

As before, the idea of the unit test is to confirm that a unit of our code works as expected on particular inputs that we verified manually. I recommend using `assertthat` package in R. It offers a collection of functions like `validate_that()`,`see_if()` and `assert_that()` that are of great help when design unit tests as below.

Clearly, before you actually populate the array `S_` your unit tests will fail. But once you implement everything correctly they should all return `TRUE`.

```
validate_that(S_[10L] == 1L) # only 1 stamp is required to pay 10 cents
validate_that(S_[14L] == 2L) # only 2 stamps are required to pay 14 cents
validate_that(S_[16L] == 4L) # only 4 stamps are required to pay 16 cents
validate_that(S_[17L] == 2L) # only 2 stamps are required to pay 17 cents
validate_that(S_[20L] == 2L) # only 2 stamps are required to pay 20 cents
validate_that(S_[24L] == 3L) # only 3 stamps are required to pay 24 cents
# Feel free to add a couple more unit tests but don't erase the tests above
```

```
## [1] TRUE
## [1] TRUE
## [1] TRUE
## [1] TRUE
```

```
## [1] TRUE
## [1] TRUE
```