데이터 없이 R 프로그래밍 하기

이윤석
ylee@dgu.ac.kr

차 례

1 R에서 피보나치 수 찾기

피보나치 수열을 만드는 절차는 평이하면서도 다양한 접근이 가능해서 어는 프로그래밍언어든간에 입문 단계에서 알고리즘을 익히는 데 편리하다. 피보나치 수열은 0, 1로 시작해서 n번째($n > 2$) 피보나치 수는 n-1번째 수와 n-2번째 수의 합으로 정의된다. 즉, 첫 10개의 피보나치 수는 다음처럼 보일 것이다:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

이 소단원의 목표는 피보나치 수를 찾는 것, 구체적으로는 자연수 n이 주어졌을 때 n번째 피보나치 수를 찾는 R 함수를 만들되, 네 가지 상이한 접근법을 구사할 것이며 후반부에서는 단일한 n뿐 아닌 벡터 형식으로 주어진 여러 개의 n에 대해서 n번째 피보나치 수를 찾도록 `sapply` 함수로 도입하여 적용할 것이다. 다음처럼 보이게 된다:

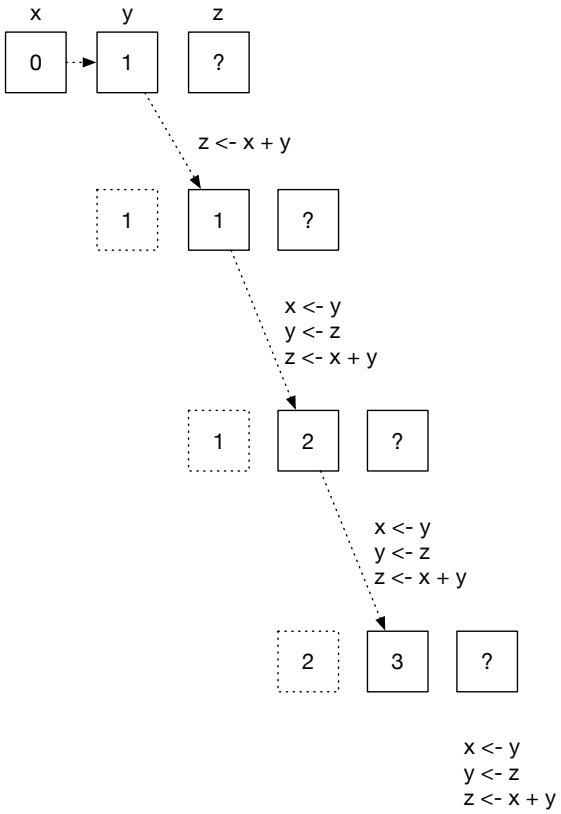
```
> your_function_name(n = 9)
[1] 21
> sapply(8:10, function(x) your_function_name(x) )
[1] 13 21 34
```

1.1 피보나치 벡터를 먼저 만드는 법

제목처럼 단순하다. 빈 숫자열에 0, 1로 시작해서 n까지 차곡차곡 피보나치 숫자를 채워 나가는 형식이다. 들여다 볼 부분이 없다.

```
> fibonacci_1 <- function(n) {

+         fibonacci <- c(0, 1)
+         for (i in 3:n) fibonacci[i] <-
+                 fibonacci[i - 1] + fibonacci[i - 2] #
+         return (fibonacci[n])
+         }
> fibonacci_1(n = 8)

## [1] 13

> sapply(11:20, function(x) fibonacci_1(x))

##  [1]   55   89  144  233  377  610  987 1597 2584 4181
```

1.2 자리이동 반복법



...
...
...

n번째 피보나치 수를 알기 위해서 n까지 피보나치 수열을 모두 보관하는 것이 낭비라는 생각이 들 때가 있다. n번째 피보나치 수를 알기 위해서 직접 필요한 값은 그 이전 값, 또 그 이전 값, 이렇게 두 개뿐이다. 이를 (조금 길지만 명시적으로) `n_minus_1`과 `n_minus_2`라는 값에 담아 차수가 증가할 때마다 한 칸씩 자리를 이동시키면 긴 벡터를 관리할 필요 없이 피보나치 수를 찾을 수 있다.

```
> fibonacci_2 <- function(n) {

+         if (n <= 2) return (n - 1)
```

```
+         n_minus_2 <- 0
+         n_minus_1 <- 1

+         for (i in 3:n) {
+                 n_0 <- n_minus_1 + n_minus_2
+                 n_minus_2 <- n_minus_1
+                 n_minus_1 <- n_0
+         }
+         return (n_0)
+         }
> fibonacci_2(n = 8)

## [1] 13

> sapply(1:30, function(x) fibonacci_2(x))

##  [1]      0      1      1      2      3      5      8     13     21     34     55
## [12]     89    144    233    377    610    987   1597   2584   4181   6765  10946
## [23]  17711  28657  46368  75025 121393 196418 317811 514229
```

1.3 재귀호출법

정의를 재확인하자면 n번째 피보나치 값은($Fibonacci_n$) n - 1번째, n - 2번째 값의 합이고 n - 1번째 피보나치 값은 n - 2번째와 n - 3번째 값의 합이고... $n > 2$까지 풀어서 쓸 수 있다. 이를 다음처럼 묘사하면 이해하기 쉽다.

$$
\begin{aligned}
Fibonacci_n &= Fibonacci_{n-1} + Fibonacci_{n-2} \\
Fibonacci_{n-1} &= Fibonacci_{n-2} + Fibonacci_{n-3} \\
Fibonacci_{n-2} &= Fibonacci_{n-3} + Fibonacci_{n-4} \\
&\cdots \\
&\cdots \\
&\cdots
\end{aligned}
\tag{1}
$$

즉, n번째 피보나치 값을 구하는 데 필요한 값은 n-1번째, n-2번째 피보나치 값일 뿐이므로 피보나치 계산 절차(함수)를 공통 자원으로 활용할 수 있다면 복잡한 반복문이 필요치 않다. 이를 실제로 구현하려면 함수 안에서 자기 함수를 호출하는 기법을 동원하게 되는데 이를 재귀적호출(recursive call)이라고 부른다. 재귀적호출을 구현한 세 번째 피보나치 계산 함수 `fibonacci_3()`의 코드는 다음처럼 간결하다. 코드는 지나치게 간결해서 어느 부분에서 답을 계산해내고 있는지조차 쉽게 식별되지 않지만 함수를 호출하면 답이 나온다.

```
> fibonacci_3 <- function(n) {

+         if (n <= 2) return (n - 1)
+         return (fibonacci_3(n - 1) + fibonacci_3(n - 2)) #
+ }
> fibonacci_3(10)

## [1] 34

> sapply(1:30, function(x) fibonacci_3(x))

##  [1]      0      1      1      2      3      5      8     13     21     34     55
## [12]     89    144    233    377    610    987   1597   2584   4181   6765  10946
## [23]  17711  28657  46368  75025 121393 196418 317811 514229
```

논리적으로는 명징한 접근법이지만 재귀 연산의 과정에서 중첩되는 동일한 연산의 양이 기하급수적으로 늘기 때문에 n이 커질수록 시스템의 부하가 막대하다. 따라서 재귀호출의 알고리즘 훈련에만 국한하여 사용하는 것이 좋겠다.

1.4 근사식

앞서 나온 절차들은 n번째 피보나치 수를 얻기 위해서 그 앞의 모든 피보나치 수를 계산해야 한다. 다음의 근사식으로 n번째 피보나치 수를 바로 얻을 수 있다고 제안하고 있다. 이 근사식은 과제를 위해서 의도적으로 불완전함을 개량하지 않는 채 소개하고 있음에 주의해야 한다.

$$Fibonacci_n = \phi^n/\sqrt{5}$$

여기서 $\phi = (1+\sqrt{5})/2$이다. 주의할 점은 이 근사식에서 계산된 수열은 0을 제외한 1부터 나온다는 점이다. 지금까지 얻은 피보나치 수와 바로 비교하려면 n

을 보정해야 한다. 우선, n을 보정하지 않고 책에서 소개되었던 대로 계산하면 다음처럼 나온다.

```
> fibonacci_4 <- function(n) {
+         # Knuth's "The Art of Computer Programming - Book 1"
+         # pp 79-84.
+         phi <- (sqrt(5) + 1) / 2
+         return (round(phi^n / sqrt(5))) #
+         }
> sapply(1:10, function(x) fibonacci_4(x))

##  [1]  1  1  2  3  5  8 13 21 34 55
```

1.5 벡터를 인수로 받는 피보나치 함수 만들기

여기까지 만든 피보나치 수를 찾는 함수들은 모두 인수 n으로 단일값만 전달할 수 있었다. 여러 개의 숫자를 n에 대입하면 경고가 발생하면서 오답을 내기에, 여러 값을 전달하기 위해서는 `sapply` 함수를 동원해야 했다. `fibonacci_2` 함수를 일례로 들면 이러하다.

```
> fibonacci_2(1:10)

## Warning in if (n <= 2) return(n - 1): the condition has length > 1
and only the first element will be used

##  [1] 0 1 2 3 4 5 6 7 8 9
```

이 경고의 원천은 함수 내부에 들어 있는 `if` 문과 `for` 문형이 벡터를 인수로 받지 못하는 한계에서 비롯한다. 즉, 알고리듬이 다소 복잡해지는 약점 없이 `if` 문과 `for` 문에 벡터가 들어가지 않도록 함수를 다시 작성해 보았다.

```
> fibonacci_2v <- function(n) {
+         fibonacci_2s <- function(n) {

+                 if (n <= 2) return (n - 1)

+                 n_minus_2 <- 0
```

```
+                 n_minus_1 <- 1

+                 for (i in 3:n) {
+                         n_0 <- n_minus_1 + n_minus_2
+                         n_minus_2 <- n_minus_1
+                         n_minus_1 <- n_0
+                 }
+                 return (n_0)
+                 }

+         answer <- integer()
+         for (i in n)
+                 answer <- c(answer, fibonacci_2s(n[i]))

+         return(answer)
+ }
> fibonacci_2v(n = 1:15)

##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

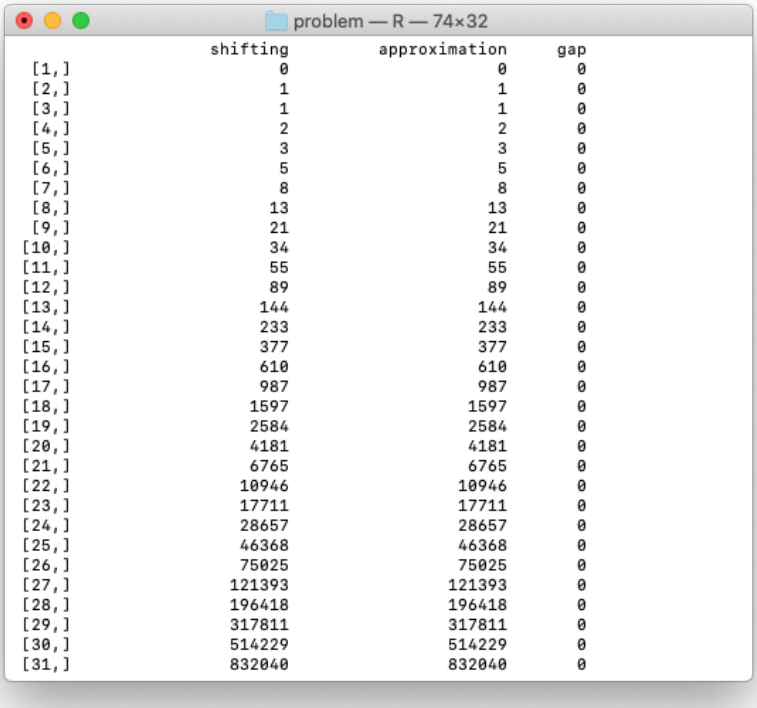
함수 `fibonacci_2v` 내부에서 선언된 함수 `fibonacci_2s`는 `fibonacci_2`와 똑같은 모습을 가졌지만 `fibonacci_2v` 함수의 바깥에서는 사용할 수 없다. 앞선 만들었던 여타의 함수들을 글로벌 함수로 부를 때 `fibonacci_2s`를 로컬 함수라고 부른다(local to the function `fibonacci_2v`). 함수와 변수들의 작동 공간에 대해서는 다른 시간에 더 다룰 것이다. 다만, 사용자가 실수로 `fibonacci_2s`에 벡터를 인수로 전달할 때 초래되는 에러를 원천적으로 막기 위해서 로컬로 만들었다.

```
> fibonacci_2s(10)

## Error in fibonacci_2s(10): could not find function "fibonacci_2s"
```

1.6 과제 1

마지막에 소개한 근사식은 n이 작을 때엔 정답을 내지만 n이 커질수록 오차가 커지므로 추가적인 보정이 필요한 것으로 알려져 있다. 과제는 다음과 같다.

자리이동 반복법으로 구한 피보나치 수를 맨 왼쪽 열에 두고, 근사식으로 구한 피보나치 수를 가운데 열에 두고, 두 값의 차이를 오른쪽 열에 두는 행렬을 구하되, 두 값이 차이를 내기 시작하는 n번째 피보나치 수를 찾는 코드를 작성한다. 행렬을 출력했을 때 다음처럼 보이도록 열 이름을 배정하고, 소수점 아래 숫자들이 모두 보이도록 유효숫자를 조정한다. 출력할 때 유효숫자를 조정하는 인수는 digits = ??이다.



	shifting	approximation	gap
[1,]	0	0	0
[2,]	1	1	0
[3,]	1	1	0
[4,]	2	2	0
[5,]	3	3	0
[6,]	5	5	0
[7,]	8	8	0
[8,]	13	13	0
[9,]	21	21	0
[10,]	34	34	0
[11,]	55	55	0
[12,]	89	89	0
[13,]	144	144	0
[14,]	233	233	0
[15,]	377	377	0
[16,]	610	610	0
[17,]	987	987	0
[18,]	1597	1597	0
[19,]	2584	2584	0
[20,]	4181	4181	0
[21,]	6765	6765	0
[22,]	10946	10946	0
[23,]	17711	17711	0
[24,]	28657	28657	0
[25,]	46368	46368	0
[26,]	75025	75025	0
[27,]	121393	121393	0
[28,]	196418	196418	0
[29,]	317811	317811	0
[30,]	514229	514229	0
[31,]	832040	832040	0

다음은 1부터 100번째까지 피보나치 수를 구하는 R 코드의 일부이다. 원하는 출력물을 얻도록 코드를 완성하되, 출력할 최대 n값은 두 열의 값이 차이(gap)가 1이상으로 벌어질 때까지로 배정한다.

```
> f1 <- sapply(1:100, function(x) fibonacci_2(x))
> f2 <- sapply(...)
> ...
> ...
> ...
```

2 숫자열 정렬하기

주어진 숫자열을 순서대로 정렬하는 절차는 간단한 알고리듬부터 복잡한 알고리듬까지 다양하며, 이 내용만으로 두꺼운 책 한 권을 쓸 수 있을 만큼 다양한 논리수학적 문제를 배태하고 있다. 이 소단원의 목표는 두 개의 간단한 정렬 알고리듬을 소개하고 만드는 과정에서 프로그래밍에 익숙해지는 것이다. 먼저 밝혀둘 점은 R에는 이미 자동정렬 함수인 sort가 내장되어 있다는 점이다. 당연히 우리는 이 함수의 힘을 빌리지 않고 R의 기초 연산자와 조건, 반복 함수만으로 숫자열 정렬을 구현한다.

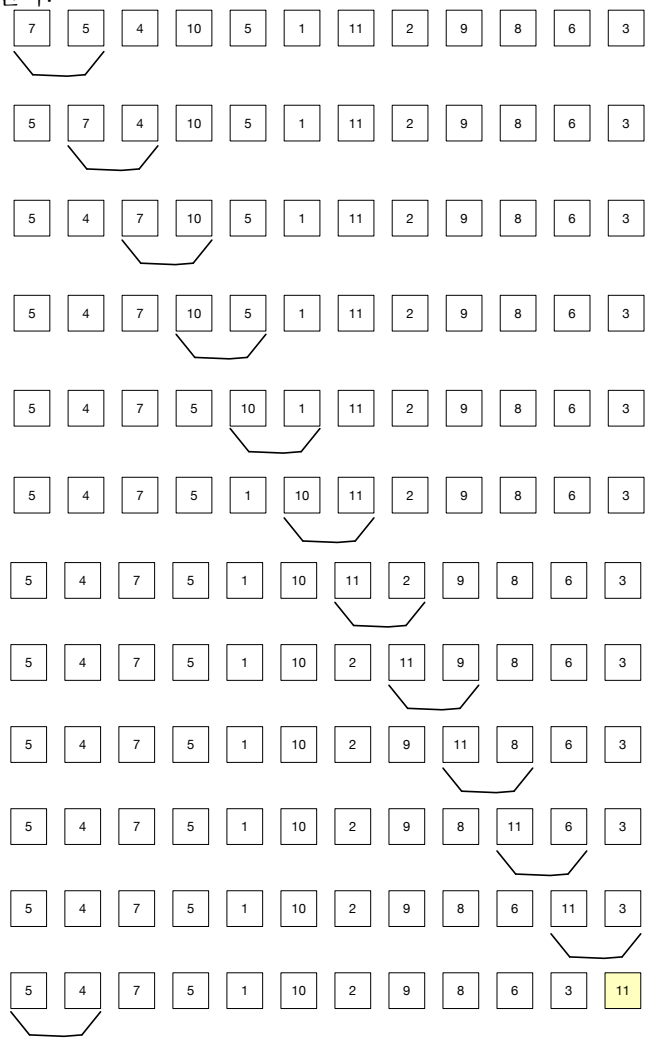
```
> sort(c(7, 5, 4, 10, 5, 1, 11, 2, 9, 8, 6, 3))

##  [1]  1  2  3  4  5  5  6  7  8  9 10 11
```

```
> ?sort
sort                    package:base                    R Documentation

Sorting or Ordering Vectors

Description:

     Sort (or _order_) a vector or factor (partially) into ascending or
     descending order.  For ordering along more than one variable,
     e.g., for sorting data frames, see ‘order’.

Usage:

     sort(x, decreasing = FALSE, ...)

     ## Default S3 method:
     sort(x, decreasing = FALSE, na.last = NA, ...)

     sort.int(x, partial = NULL, na.last = NA, decreasing = FALSE,
              method = c("auto", "shell", "quick", "radix"), index.return = FALSE)
```

2.1 숫자를 두 개씩 일일히 비교해서 순서를 바꾸기

숫자 벡터 x의 원소를 앞에서부터 차례로 두 개씩 꺼내서 비교해서 순서대로 재배열하는 절차를 왼쪽에서 오른쪽으로 시행하는 과정이다. 그림에서 보듯 한 주

기마다 맨 마지막 값이 제자리를 찾게 되므로 주기를 되풀이할 때마다 비교의 조합을 하나씩 줄일 수 있다. 반복 구문 안에 반복 구문을 구현하는 방식으로 작성한다.




```
> easySort <- function(x) {

+         n <- length(x)
+         for (i in 0:(n - 2)) {
+                 # i = 0, 1, 2, ...., n - 2
+                 for (j in 2:(n - i)) {
```

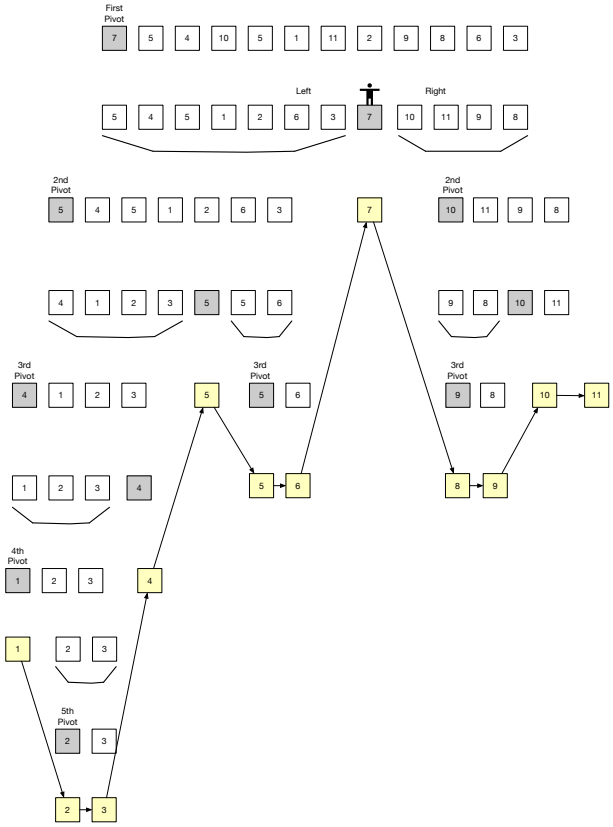
```
+                         # j = 2, 3, 4, .... n - i
+                         if (x[j - 1] > x[j]) {
+                                 temp <- x[j - 1]
+                                 x[j - 1] <- x[j]
+                                 x[j] <- temp
+                         }
+                 }
+         }
+         return(x)
+ }
> easySort(c(7, 5, 4, 10, 5, 1, 11, 2, 9, 8, 6, 3))

##  [1]  1  2  3  4  5  5  6  7  8  9 10 11

> x <- rnorm(100, 10, 100)
> easySort(x)

##   [1] -175.1854458 -174.9204150 -167.1018721 -150.1596781 -131.9928269 -121.0671595
##   [7] -119.9319999 -115.7881803 -109.7622777 -108.0473534 -102.3627592 -101.1513078
##  [13]  -97.9664967  -87.0281350  -80.7591383  -73.1367365  -72.8067605  -72.3330621
##  [19]  -71.0955952  -70.8587108  -64.4056219  -63.4951211  -60.5402952  -59.7391176
##  [25]  -56.3894585  -54.4344511  -51.9907639  -49.9640759  -49.7018682  -48.2638723
##  [31]  -46.1163568  -41.8996341  -38.6204672  -38.5844092  -30.3967121  -28.0889530
##  [37]  -27.7887643  -25.3518498  -21.6904543  -10.9454288   -8.8843780   -4.2450181
##  [43]   -3.4486741   -0.2482748    0.6018435    0.8102651    3.0524031    5.0194014
##  [49]    8.3477079    8.5299904    9.1675149   10.4883791   17.5514343   22.3055461
##  [55]   22.4079579   22.9506597   23.2748491   24.8362514   28.1228109   31.0296962
##  [61]   35.7727680   37.7862317   39.1511702   39.4252514   40.4064608   44.9535054
##  [67]   47.1533049   49.7676731   52.6301673   65.3576995   68.8529923   69.2758710
##  [73]   71.3345834   73.5918724   78.8593419   79.8602503   87.8811774   92.4000803
##  [79]   96.9292615   97.1171913   97.3975182   98.4816601  101.4622685  103.7998139
##  [85]  106.3727278  110.0462953  114.4433196  115.8354158  116.9762316  123.5149359
##  [91]  125.5369559  135.3910417  137.9728916  144.7059792  149.7340416  159.3151750
##  [97]  163.5778412  167.2263655  178.7750888  228.2114294
```

지루하고 따분해 보이지만 컴퓨터는 지치지 않고 몇 만개의 숫자라도 정렬해 낼 것이며, 이해하기도 쉽다.

2.2 피봇을 중심으로 한 좌우 정렬법

숫자 중에서 임의의 값을 골라서 그 값을 중심으로 좌우 정렬을 거듭하는 방식으로 숫자열을 정렬하는 방법이 있다. 임의의 값을 피봇이라고 부르며 그림에서는 최초의 값을 피봇으로 삼고 있다. 즉, 첫 피봇인 7을 중심으로 7보다 작은 값들은 7의 왼쪽에 두고 7보다 크거나 같은 값들은 7의 오른쪽에 둔다. 왼쪽, 오른쪽 숫자열에서 다시 피봇을 고른 뒤 절차를 재귀적으로 반복하는 수단임을 궤뚫어 보았다면 코딩이 실로 쉽다. 쪼개진 숫자열의 길이가 하나일 때 반복을 중단하면 된다. 이 함수는 당연히 왼쪽 숫자열, 피봇, 오른쪽 숫자열을 연결해서 값을 되돌리면 된다.

```
> quickSort <- function(x) {

+         if (length(x) <= 1) return(x)

+         pivot <- x[1]
+         remained <- x[-1]

+         left <- remained[remained < pivot]
+         right <- remained[remained >= pivot]

+         left <- quickSort(left)
+         right <- quickSort(right)

+         return( c(left, pivot, right) )
+ }
> quickSort(c(7, 5, 4, 10, 5, 1, 11, 2, 9, 8, 6, 3))

##  [1]  1  2  3  4  5  5  6  7  8  9 10 11
```



2.3 과제 2

피봇을 중심으로 하는 좌우 정렬법은 일명 퀵소트로 불리며 R의 내장함수 `sort`에서 사용되는 알고리듬이기도 하다. 피봇으로 어떤 숫자를 선정하는가에 따라 코드의 효율이 조금씩 달라진다. 위에서 만든 `quickSort`는 항상 첫 번째 값을 피봇으로 삼기 때문에 이미 반쯤은 정렬이 되어 있는 숫자열에서 극도의 비효율을 자랑하게 될 것이다. 주어진 숫자열의 중간에 놓인 값을($n/2$번째) 피봇으로 삼는 `quickSort2`를 만들어 보라. 그리고 나서, 평균 100, 표준편차 25를 가지는 임의의 정수 난수 101개를 만들어서 벡터 `x`에 저장한 뒤 위에서 만든 `easySort`, `quickSort`, `quickSort2`로 정렬하는 데 걸리는 시간을 측정해 본다.