# 보고서 : 실습과제 1, 실습과제 2

2019112096 서준원

## I. 실습과제 1

### I-1) perfect_number.m 코드 전문

```
%% 0. 완전수 찾기 스크립트
clear, clc; % 메모리 클리어, 명령 창 정리
%% 1. 입력 및 선언
n = input('찾고자 하는 완전수의 개수를 입력하시오. : '); % n 입력받음
index_perfect_num = 0; % 찾은 완전수 개수 (일련번호)
i = 0; % for문에 있던 증감 연산자 선언
%% 2. 알고리즘 실행
while index_perfect_num < n % 일련번호가 n과 같게 될 때까지 반복
    i = i + 1; % for문에 있던 i 증감 대체
    sum_val = 0;  % 약수의 합 초기화
    for j=1:(i-1) % 1부터 자신을 제외한 약수 찾기
        if (is_dividable(i,j)==true) % is_dividable.m 호출
            sum_val = sum_val + j; % 약수의 합
        end
    end
    if (sum_val == i) % 약수의 합 = 자기자신
        index_perfect_num = index_perfect_num+1; % 일련번호 1씩 증가
        perfect_num(index_perfect_num) = i; % 찾은 완전수 대입
    end
end
%% 3. 결과
perfect_num % 찾은 완전수를 화면에 표시함.
```

### I-2) is_dividable.m 코드 전문

```
%% 0. 약수인지 판단하는 함수 is_dividable
function output = is_dividable (int1, int2)
%% 1. 제수에서 피제수 차감 반복
while int1 > 0 % int가 양수일 동안만 반복, 0이 되거나 음수가 되면 반복 중지
    int1 = int1-int2; % int1에서 int2 차감
end
%% 2. 함수 참/거짓 출력
if int1 == 0
    output = 1; % 나머지가 0이면 1 출력
else
    output = 0; % 나머지가 0이 아니면 0 출력
end
```

**I-3) 실행 결과**

스크립트 실행 시 입력받은 “찾을 완전수의 개수” 에 맞게 완전수를 구하여 출력하는 프로그램이 완성되었다. 찾은 완전수들의 마지막 숫자는 6이거나 28임이 보여진다.

I-3-1) $n = 1$일 때

```
찾고자 하는 완전수의 개수를 입력하시오. : 1

perfect_num =

     6
```

I-3-2) $n = 2$일 때

```
찾고자 하는 완전수의 개수를 입력하시오. : 2

perfect_num =

     6    28
```

I-3-3) $n = 3$일 때

```
찾고자 하는 완전수의 개수를 입력하시오. : 3

perfect_num =

     6    28   496
```

I-3-4) $n = 4$일 때

```
찾고자 하는 완전수의 개수를 입력하시오. : 4

perfect_num =

          6          28         496        8128
```

I-3-5) $n \geq 5$일 때

n이 4를 넘어가면, 결과가 화면에 금방 출력되지 않는다. 이론적으로 알려진 다섯 번째 완전수는 33,550,336인데, for문을 통해 1부터 모든 수를 하나씩 검증하여 완전수인가 아닌가를 판단하기 때문에, 자리수가 기하급수적으로 커지는 다섯 번째 완전수의 계산이 금방 처리되지 않음을 추측할 수 있다.

**I-4) 완전수에 관한 가설과 알고리즘 개선 방안**

스크립트 실행 속도가 n의 값에 따라 기하급수적으로 증가하는 것은, 기본적으로 이 알고리즘이 1부터 모든 수를 검증한다는 그 한계에 기인한다. 따라서, “애초에 완전수가 될 수 없는 수들”을 검증 대상의 수들에서 제외하는 식으로 프로그래밍 한다면, 프로그램 실행 속도를 개선할 수 있으리라 판단된다.

지금까지 밝혀진 완전수들은 모두 짝수이며 맨 마지막 자리 숫자가 6 또는 28이라는 공통점이 있다. 하지만, 이는 수학적으로 증명되지 않은 귀납적 추측이다. 즉, 아래에 언급될 알고리즘 개선 방안은 완전수에 대한 추측이 사실이라는 전제 하에, 혹은 레온하르트 오일러에 의해 그 규칙이 증명된 짝수 완전수만을 찾을 때 사용 가능하다.

I-4-1) 기존 스크립트에서 $n = 4$인 경우의 실행 속도

프로파일 요약

*05-Apr-2019 01:31:58에 performance 시간을 사용하여 생성됨*

| 함수 이름 | 호출 | 총 시간 | 자체 시간* | 총 시간 플롯 (짙은 띠 = 자체 시간) |
|---|---|---|---|---|
| perfect_number | 1 | 56.900 s | 18.831 s | |
| is_dividable | 33028128 | 38.069 s | 38.069 s | |

프로파일러도 같이 동작하기에, 그냥 실행했을 때에 비해 10배 정도 느려졌다.

I-4-2) 검증 대상에서 홀수를 제외할 경우, $n = 4$일 때의 실행 속도

***perfect_number.m***의 9번째 줄은 자체 선언된 카운터 변수 i를 통한 증감을 실행한다. 이 때, $i = 0$부터 시작하여 1씩 증가되는데, 이는 즉 홀수를 포함한 모든 수를 검증한다는 의미이다.

***perfect_number.m***의 9번째 줄을 다음과 같이 수정한다면 홀수를 검증 대상에서 제외할 수 있다.

```
i = i + 2; % for문에 있던 i 증감 대체
```

그 뒤, 프로그램 실행 속도는 다음과 같음을 알 수 있다.

프로파일 요약

*05-Apr-2019 01:36:45에 performance 시간을 사용하여 생성됨*

| 함수 이름 | 호출 | 총 시간 | 자체 시간* | 총 시간 플롯 (짙은 띠 = 자체 시간) |
|---|---|---|---|---|
| perfect_number | 1 | 29.265 s | 10.426 s | |
| is_dividable | 16516096 | 18.839 s | 18.839 s | |

홀수를 검증 대상에서 제외했기에, 실행 시간이 절반 가까이 줄어들었음을 파악할 수 있다.

여담으로, 재미 삼아 CPU의 성능에 따른 실행 속도도 테스트 해보았다. 약 29초의 실행 시간은 i7-9700K(4.5GHz)가 장착된 데스크톱에서의 결과이다.

프로파일 요약

*05-Apr-2019 02:11:14에 performance 시간을 사용하여 생성됨*

| 함수 이름 | 호출 | 총 시간 | 자체 시간* | 총 시간 플롯 (짙은 띠 = 자체 시간) |
|---|---|---|---|---|
| perfect_number | 1 | 42.732 s | 15.255 s | |
| is_dividable | 16516096 | 27.477 s | 27.477 s | |

약 42초가 소요된 위 테스트 결과는 i7-6700HQ(3.2GHz)가 장착된 랩톱에서의 결과이다. 이를 통해, CPU의 클럭, 성능 차이가 코드 실행 시간에 유의미한 영향을 줌을 고찰할 수 있다.

I-4-3) 끝자리가 6이거나 28인 수만을 검증할 경우, $n = 4$일 때의 실행 속도

**I-4-1**에서 사용한 방법과 비슷하지만 약간 다르게 처리하면, 끝자리가 6이거나 28인 수 만을 대상으로 검증을 실행할 수 있다. 카운터 변수 i를 증가 처리할 때, 시작점을 6으로 잡은 뒤 10씩 증가하도록 하면 끝자리가 6인 수만을 골라낼 수 있다. 마찬가지로 시작점을 28로 잡은 뒤 100씩 증가하도록 하면 끝자리가 28인 수만을 골라낼 수 있다.

다만, 코딩 실력의 한계로 별개의 루프문에서 도출되는 결과를 하나의 벡터에 통합시키지는 못하였다.

보완하여 작성한 코드의 전문은 다음과 같다.

```
%% 0. 완전수 찾기 스크립트
clear, clc; % 메모리 클리어, 명령 창 정리
%% 1. 입력 및 선언
n = input('찾고자 하는 완전수의 개수를 입력하시오. : '); % n 입력받음
index_perfect_num_6 = 0; % 찾은 끝자리 6 완전수 개수 (일련번호)
index_perfect_num_28 = 0; % 찾은 끝자리 28 완전수 개수 (일련번호)
i_6 = -4; % for문에 있던 증감 연산자 선언, 6-10 = -4
i_28 = -72; % for문에 있던 증감 연산자 선언, 28-100 = -72
%% 2. 알고리즘 실행 - 끝자리가 6인 경우
while index_perfect_num_6 + index_perfect_num_28 < n % 루프 기준 : 일련번호의 합
    i_6 = i_6 + 10; % for문에 있던 i 증감 대체
    sum_val_6 = 0;  % 약수의 합 초기화
    for j=1:(i_6-1) % 1부터 자신을 제외한 약수 찾기
        if (is_dividable(i_6,j)==true) % is_dividable.m 이용
            sum_val_6 = sum_val_6 + j; % 약수의 합
        end
    end
    if (sum_val_6 == i_6) % 약수의 합 = 자기자신
        index_perfect_num_6 = index_perfect_num_6+1; % 일련번호 1씩 증가
        perfect_num_6(index_perfect_num_6) = i_6; % 찾은 완전수 대입
    end
%% 2. 알고리즘 실행 - 끝자리가 28
    i_28 = i_28 + 100; % for문에 있던 i 증감 대체
    sum_val_28 = 0;  % 약수의 합 초기화
    for j=1:(i_28-1) % 1부터 자신을 제외한 약수 찾기
        if (is_dividable(i_28,j)==true) % is_dividable.m 이용
            sum_val_28 = sum_val_28 + j; % 약수의 합
        end
    end
    if (sum_val_28 == i_28) % 약수의 합 = 자기자신
        index_perfect_num_28 = index_perfect_num_28+1; % 일련번호 1씩 증가
        perfect_num_28(index_perfect_num_28) = i_28; % 찾은 완전수 대입
    end
end
%% 3. 결과
if n == 1
  perfect_num_6 % 예외처리 : n이 1이면 끝자리가 6인거만 표시
else
  perfect_num_6
  perfect_num_28
end
```

## 프로파일 요약

*05-Apr-2019 11:55:04에 performance 시간을 사용하여 생성됨*

| 함수 이름 | 호출 | 총 시간 | 자체 시간* | 총 시간 플롯 (짙은 띠 = 자체 시간) |
|---|---|---|---|---|
| perfect_number_even | 1 | 3.322 s | 1.999 s |  |
| is_dividable | 367934 | 1.323 s | 1.323 s |  |

그 결과 I-4-2에서 홀수만 검증하는 스크립트에 42초가 걸렸던 랩톱에서 1.999초만에 4개의 완전수를 찾아낼 수 있었다.

```
찾고자 하는 완전수의 개수를 입력하시오. : 4

perfect_num_6 =

     6    496


perfect_num_28 =

          28        8128
```

실행 결과는 위와 같다.

### I-5) 결론

MatLab을 통해 완전수를 구하는 스크립트를 프로그래밍 하였고, 완전수의 규칙을 실제로 확인해 볼 수 있었다. **I-4)**에서 언급했듯이, "완전수가 될 수 없는 수"를 검증 대상에서 제외하는 식으로 알고리즘을 구축한다면, $n \geq 5$일 때도 완전수를 더 적은 시간을 들여 구해나갈 수 있으리라 기대한다. 아래에 각 환경에서 $n = 4$에서의 스크립트 실행 속도를 비교한 도표를 첨부하겠다. 프로파일러가 켜진 채 실행 속도를 측정하는 것이라, 실제 프로그램 구동 속도와는 큰 차이가 있으니 비례 관계를 알아보는 차원에서 참고할 수 있는 자료라고 판단된다.

| | 랩톱(i7-6700HQ, 3.2GHz) | 데스크톱(i7-9700K, 4.6GHz) |
|---|---|---|
| 모두 검증 | 86.194s | 57.703s |
| 짝수만 검증 | 44.834s | 29.120s |
| 끝자리 검증 | 2.031s | 1.813s |

## II. 실습과제 2

### II-1) eratosthenes_sieve.m 코드 전문

```
%% 0. 에라스토테네스의 체 알고리즘
clear, clc; % 명령 창 초기화, 메모리 초기화
%% 1. 입력 및 선언
n = input('표현 가능한 소수 개수를 찾기 위한 비트수 N을 입력하시오 : '); % 입력
index_prime=0; % 소수 인덱스 선언
is_prime=0; % 소수 판단 인자 선언
%% 2. 정수가 소수인지 아닌지 판단
for i=2:((2^n)-1) % 2부터 2^n-1까지 체크
    if i == 2 % i가 2일때에 한하는 연산법
    for j=1:i
        if is_dividable(i,j)==1 % i가 j에 나누어 떨어질 때
            is_prime = is_prime + 1; % 나누어 떨어진 횟수 1 증가
        end
    end
    if is_prime == 2 % 1과 자기 자신, 2회 나누어 떨어졌을 때
        index_prime = index_prime + 1; % 소수 인덱스 1 증가
        prime_number_result(index_prime)=i; % 소수 벡터에 i 대입
        is_prime = 0; % 나누어 떨어진 횟수 초기화
    else
        is_prime = 0; % 나누어 떨어진 횟수 초기화
    end
    else % i가 2가 아닐 때
        for j=prime_number_result % 지금까지 나온 소수의 행렬
        if is_dividable(i,j)==1 % i가 j에 나누어 떨어질 때
            is_prime = is_prime + 1; % 나누어 떨어진 횟수 1 증가
        end
    end
    if is_prime == 0 % 한번도 나누어 떨어지지 않았을 때
        index_prime = index_prime + 1; % 소수 인덱스 1 증가
        prime_number_result(index_prime)=i; % 소수 벡터에 i 대입
        is_prime = 0; % 나누어 떨어진 횟수 초기화
    else
        is_prime = 0; % 나누어 떨어진 횟수 초기화
    end
    end
end
%% 3. 결과 출력
disp(n + "비트로 표현될 수 있는 소수 개수 : " + index_prime);
% 소수 개수 출력
disp(n + "비트로 표현될 수 있는 최대 소수 : " + prime_number_result(index_prime));
% 최대 소수 출력
```

### II-2) 실행 결과

```
표현 가능한 소수 개수를 찾기 위한 비트수 N을 입력하시오 : 12
12비트로 표현될 수 있는 소수 개수 : 564
12비트로 표현될 수 있는 최대 소수 : 4093

표현 가능한 소수 개수를 찾기 위한 비트수 N을 입력하시오 : 8
8비트로 표현될 수 있는 소수 개수 : 54
8비트로 표현될 수 있는 최대 소수 : 251
```

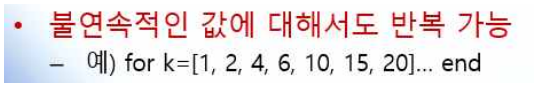
### II-3) 알고리즘 설계 과정 및 후기

처음에는 문제를 잘못 이해해, “에라스토테네스의 체”가 아닌 “소수 찾기 알고리즘”을 설계하였다. 코드는 아래와 같다.

```
%% 0. 에라스토테네스의 체 알고리즘
clear, clc; % 명령 창 초기화, 메모리 초기화
%% 1. 입력 및 선언
n = input('표현 가능한 소수 개수를 찾기 위한 비트수 N을 입력하시오 : '); % 입력
index_prime=0; % 소수 인덱스 선언
is_prime=0; % 소수 판단 인자 선언
%% 2. 정수가 소수인지 아닌지 판단
for i=2:((2^n)-1) % 2부터 2^n-1까지 체크
    for j=1:i
        if is_dividable(i,j)==1 % i가 j에 나누어 떨어질 때
            is_prime = is_prime + 1; % 나누어 떨어진 횟수 1 증가
        end
    end
    if is_prime == 2 % 1과 자기 자신, 2회 나누어 떨어졌을 때
        index_prime = index_prime + 1; % 소수 인덱스 1 증가
        prime_number(index_prime)=i; % 소수 벡터에 i 대입
        is_prime = 0; % 나누어 떨어진 횟수 초기화
    else
        is_prime = 0; % 나누어 떨어진 횟수 초기화
    end
end
%% 3. 결과 출력
disp(n + "비트로 표현될 수 있는 소수 개수 : " + index_prime);
% 소수 개수 출력
disp(n + "비트로 표현될 수 있는 최대 소수 : " + prime_number_result(index_prime));
% 최대 소수 출력
```

하지만, 위 코드는 결정적으로 모든 수를 1부터 나누어 보며 소수인가 아닌가를 판단한다는 결점을 가지고 있었고, 이는 실행 속도적 측면에서 효율성이 떨어지는 코드이다. 주어진 과제였던 “에라토스테네스의 체”를 설계하기 위해서는 약수인가 비교해보는 대상을 “이전까지 산출되었던 소수들(이하 기존 소수)”로 제한해야 했다.

어찌 문제를 풀어야 할지 감이 안잡히던 와중, 강의 자료에서 기적과도 같은 문구를 발견하였다.




위 말인 즉슨, for문에서 약수 여부 판단 대상에 기존 소수들의 행렬을 대입할 수 있다는 의미였다. 결론적으로, 새로운 숫자를 기존 소수들에게 한번이라도 나누어 떨어지지 않는다면 이 새로운 숫자가 곧 새로운 소수가 되는 것이었다.

하지만 문제는 2였다. 2는 코드의 시작 지점으로서, 2를 소수인지 아닌지 판단할 때는 기존 소수가 존재하지 않았다. 2의 소수 여부를 판단할 때는 1을 사용해야 하는데, 그렇다고 기존 소수 행렬에 1을 포함시킨다면 이후 코드가 꼬여버리게 된다.

필자의 해결 방안이 효율적인가는 모르겠으나, 필자는 "예외 처리"를 통해 문제를 해결하였다. 과제의 조건을 해석해 본다면, 가장 작은 소수 "2"도 소수 판별법을 통해 도출해야 한다고 판단되었다. 다시 말해 기존 소수 행렬에 "2"를 인덱스 "1"로서 강제로 지정해 둘 수도 있지만, 이는 과제의 조건에 알맞지 않다는 판단이다.

따라서, **II-3**에서 언급된 소수 찾기 알고리즘을 이용하였다. **II-3**에서의 방법은 "1과 자기 자신으로만 나누어지는가?" 인데, 이는 2의 소수 판별에 사용할 수 있는 방법이기 때문이다. if~else 구문을 통하여 최초 연산일 때와, 그렇지 않을 경우를 구분하여 과제의 조건에도 부합하며 순수한 "에라토스테네스의 체"를 구현하는 데에 성공하였다.

### II-4) 결론 : 알고리즘 간 실행 시간 비교

프로파일 요약

*06-Apr-2019 02:20:11에 performance 시간을 사용하여 생성됨*

| 함수 이름 | 호출 | 총 시간 | 자체 시간* | 총 시간 플롯 (좁은 띠 = 자체 시간) |
|---|---|---|---|---|
| prime number | 1 | 14.977 s | 5.564 s | |
| is dividable | 8386559 | 9.413 s | 9.413 s | |

프로파일 요약

*06-Apr-2019 02:20:33에 performance 시간을 사용하여 생성됨*

| 함수 이름 | 호출 | 총 시간 | 자체 시간* | 총 시간 플롯 (좁은 띠 = 자체 시간) |
|---|---|---|---|---|
| eratosthenes sieve | 1 | 3.595 s | 1.754 s | |
| is dividable | 1239491 | 1.841 s | 1.841 s | |

좌측은 전수 검증을 통한 소수 판별법, 우측은 에라토스테네스의 체를 통한 소수 판별법 알고리즘의 실행 시간이다. 예측한 대로, 거쳐야 할 계산이 적은 에라토스테네스의 체 쪽이 실행 속도 면에서 우월하다는 결론을 얻을 수 있다.