

레포트 1: 마술 카드를 구현해보자.

- 작성자 – SW융합대학 컴퓨터소프트웨어공학과 20223519 전한결
- 과목 – 2023학년도 1학기 이산수학(12744) Report 1
- 과제 제시일 – 2022년 3월 7일
- 과제 마감일시 – 2022년 3월 27일 24:00

목차

1. 문제 분석
 - 1.1. 마술 카드 세션
 - 1.2. 마술 카드의 원리
2. 소스 코드
 - 2.1. 제공된 코드의 수정
 - 2.2. 일반적인 개수의 카드로 세션 수행하기
 - 2.3. 카드 장수를 늘려서 스택 overflow 발생시키기
 - 2.3.1. 스택 오버플로우
 - 2.3.2. 프로그램에서 사용한 메모리
 - 2.3.3. 스택 오버플로우의 해결 방안
 - 2.4. 동적 할당과 자동 할당의 실행 시간 비교
3. 결론
 - 3.1. 레포트를 통해 해결한 것과 배운 것
 - 3.1.1. 마술 카드 원리 이해
 - 3.1.2. 마술 카드의 구현
 - 3.1.3. 버퍼 오버플로우와 스택 오버플로우에 대한 이해
 - 3.2. 레포트를 통해서도 해결하지 못한 것
4. 참고 문헌
 - 4.1. 그래프

1. 문제 분석

1.1. 마술 카드 세션

이번 레포트를 통해 알아보고자 하는 것은 “마술 카드”이다. 마술 카드를 통한 숫자 맞추기 세션(이하 세션)에는 마술사와 관객의 두 명의 주체가 참여한다. 세션이 시작되면 다음과 같은 과정이 진행된다.

1. 관객은 마술사가 제시하는 범위 내에서 0을 포함한 자연수 중 하나를 선택하고, 이것을 마술사에게 말하지 않고 마음 속으로만 생각한다.
2. 마술사는 관객에게 카드를 보여주고, 관객은 마술사가 보여준 카드에 자신이 생각한 수가 있는지 없는지를 마술사에게 알린다. 이것을 질문이라고 한다.
3. 질문을 몇 번 반복한다.
4. 마술사는 관객이 한 대답에 따라서 관객이 마음속으로 생각한 수가 무엇인지를 맞춘다.

3월 7일에 진행한 수업에서는 3장의 카드로 0부터 7까지의 숫자 중 하나를 맞추는 세션, 5장의 카드로 0부터 31까지의 숫자 중 하나를 맞추는 세션이 진행되었다. 숫자를 맞추는 마술은 관객이 마술사에게 제공한 정보에 비해 마술사가 얻은 정보가 적다고 느껴지기 때문에 관객으로 하여금 신기함을 느끼게 한다.

1.2. 마술 카드의 원리

마술사는 총 n 장의 카드를 보여주는 세션에서 관객에게 0 이상 $2^n - 1$ 이하, 즉 2^n 개의 숫자 중 하나를 고르도록 제시하며, 이때 각각의 카드에는 2^{n-1} 개의 수가 써있게 된다. 이때 관객이 선택한 수는 2진수로 표현했을 때에 n 자리 수가 넘지 않으며, 각 카드에 있는 수는 2진수로 표현했을 때에 특정한 위치의 비트가 1인 수들이 된다. 다음은 카드 장수 $n = 4$ 인 세션에서 사용하는 카드의 예시이다.

$$\begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \\ 13 & 15 & \end{bmatrix} \begin{bmatrix} 2 & 3 & 6 \\ 7 & 10 & 11 \\ 14 & 15 & \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \\ 7 & 12 & 13 \\ 14 & 15 & \end{bmatrix} \begin{bmatrix} 8 & 9 & 10 \\ 11 & 12 & 13 \\ 14 & 15 & \end{bmatrix}$$

위 예시에서 가장 왼쪽 카드에 있는 수들은 모두 2진수로 표현했을 때 2^0 의 자리수가 1인 수들이고, 그 오른쪽 카드에 있는 수들은 2^1 의 자리수가 1인 수, 그 오른쪽 카드에 있는 수들은 2^2 의 자리수가 1인 수로, 이와 같은 패턴이 모든 카드에 진행되는 식이다.

관객이 선택한 수는 2진수로 표현했을 때 n 자리수를 넘지 않으므로, 마술사가 위와 같이 총 n 회 질문을 수행한 후에는 관객이 선택한 수를 2진수로 표현했을 때에 각각의 자리수가 무엇인지 모두 알 수 있게 된다. 따라서 마술사는 관객이 선택한 수가 무엇인지 알 수 있게 되는 것이다.

대부분의 경우에 카드에는 오름차순으로 수를 정렬해 두는데, 이 경우 카드에 있는 가장 작은 수가 무엇인지 쉽게 알 수 있다. 이 수는 2진수로 표현했을 때 해당 카드에서 확인하고자 하는 자릿수만을 1로 하고 나머지 자릿수는 모두 0인 수가 된다. 따라서 마술사는 관객이 고른 수가 포함되어있다고 한 카드들에서 가장 작은 수들을 모두 더한 합이 관객이 고른 수가 된다는 것을 알 수 있다. 이 방법을 통해 마술사는 카드의 조합에 대한 수의 관계를 하나하나 외우고 있지 않아도 어떤 수를 골랐는지 알아낼 수 있으며, 암산만 할 수 있다면 몇 장의 카드를 가지고 세션을 진행하는지에 상관 없이 그 수가 무엇인지 알 수 있게 되는 것이다.

관객이 마술사에게 제공하는 정보의 양이 충분하다는 것은 경우의 수를 고려해보는 것으로 쉽게 이해할 수 있다. 총 n 장의 카드를 보여주는 세션에서 관객이 선택할 수 있는 수의 경우의 수는 2^n 이고, 각 카드에 있는 수는 그의 절반인 2^{n-1} 이므로, 마술사는 한 번의 질문을 통해서 관객이 선택한 수의 경우의 수를 반씩 줄여나갈 수 있다. 이때, 경우의 수를 반씩 줄여나가는 행위를 n 번 반복한 후에는 관객이 선택한 수의 경우의 수가 $2^n \cdot \left(\frac{1}{2}\right)^n = 1$ 이므로, 마술사는 관객이 어떤 수를 선택했는지 특정할 수 있게 되는 것이다.

2. 소스 코드

이 프로젝트를 진행하면서 작성한 코드는 모두 다음과 같은 컴파일러를 통해 실행할 수 있음이 확인되었다.

- gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
- Apple clang version 12.0.0 (clang-1200.0.32.29)

2.1. 제공된 코드의 수정

교재 1장 실습 부분에서 이번 프로젝트에서 사용할 수 있는 소스 코드의 예제를 찾을 수 있었다. 하지만 해당 코드는 중복되는 내용이 많고, 필요 이상으로 많은 함수가 선언되어 있어서 카드의 수를 늘리거나 일반적인 개수의 카드를 사용하는 데에 어려움이 있었다. 따라서 `magic1.c` 와 같이 코드를 수정했다.

`magic1.c`에서는 기존의 `my_card1`, `my_card2`, ..., `my_card5` 함수를 일반적으로 사용할 수 있도록 `my_card` 함수로 통합해서 사용하게 했고, 나머지 부분은 출력 형식이 조금 다르거나 입력을 처리하는 방법이 조금 다를 뿐, 카드를 보여주고 수를 계산하는 매커니즘 자체는 동일하게 했다.

과제에서 제시된 요구사항은 `magic1.c`를 가지고 수행해보고자 한다.

```
26 27 30 31
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 0
----- 카드 -----
4 5 6 7
12 13 14 15
20 21 22 23
28 29 30 31
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 0
----- 카드 -----
8 9 10 11
12 13 14 15
24 25 26 27
28 29 30 31
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 0
----- 카드 -----
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 1
당신이 생각한 수는 17입니다.
[16:09:05] sch:magiccard $
[0] 0:zsh* "sch-All-Series" 16:10 09-Mar-23
```

magic1.c 를 실행한 화면

2.2. 일반적인 개수의 카드로 세션 수행하기

프로젝트를 진행한 컴파일러에서는 정수 변수의 내용만큼의 길이를 가진 2차원 배열을 선언하는 것이 가능했고, 따라서 일반적인 개수로 카드를 생성하려면 해당 카드의 목록을 생성하는 함수를 통해 배열을 채우는 것이 좋겠다고 판단했다.

일반적인 개수의 카드로 세션을 수행하는 과제는 magic2.c 에서 수행했다.

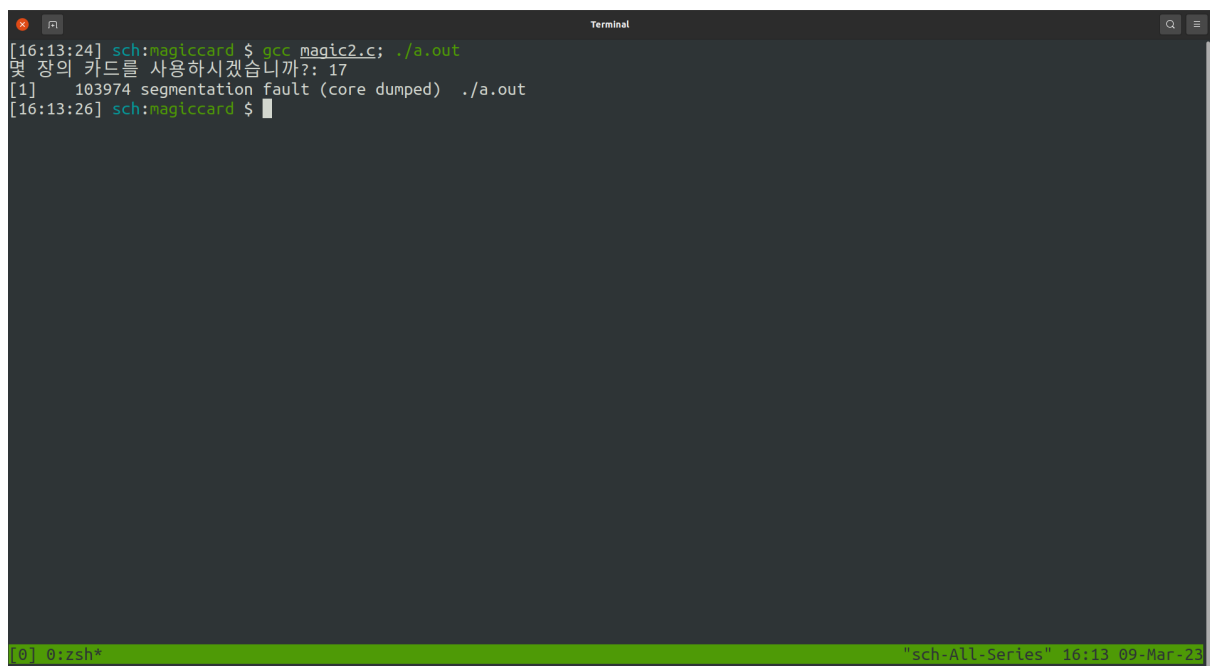
```
[16:11:03] sch:magiccard $ gcc magic2.c; ./a.out
몇 장의 카드를 사용하시겠습니까?: 4
----- 카드 -----
1 3 5 7
9 11 13 15
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 1
----- 카드 -----
2 3 6 7
10 11 14 15
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 0
----- 카드 -----
4 5 6 7
12 13 14 15
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 1
----- 카드 -----
8 9 10 11
12 13 14 15
생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 1
당신이 생각한 수는 13입니다.
[16:11:12] sch:magiccard $
[0] 0:zsh* "sch-All-Series" 16:12 09-Mar-23
```

magic2.c 를 실행한 화면

2.3. 카드 장수를 늘려서 스택 overflow 발생시키기

원래 스택 오버플로우에 대해 알고 있던 것에 따르면, 재귀함수가 자신을 호출하는 횟수가 너무 많아지게 되면 함수의 콜 스택이 감당할 수 있는 것 이상의 정보가 들어오게 되면서 더 이상 콜 스택에 자료를 추가할 수 없어 오류가 발생하는 것이 바로 스택 오버플로우인 것으로 알고 있었다. 하지만 `magic2.c`에는 재귀함수를 사용하지 않았고, 심지어는 `magic1.c`나 실습 자료로 제공된 코드에도 재귀함수를 실행하는 부분이 없었기 때문에 어떻게 스택 오버플로우가 발생할 수 있을지 의아했다.

내가 개발하는 데에 사용한 환경에서, `magic2.c`를 사용하면 16개의 카드까지는 문제 없이 세션을 실행할 수 있지만, 17개의 카드를 사용하는 세션을 시작하려고 하면 다음과 같이 오류가 뜨면서 프로그램의 실행이 중지되었다.



```
[16:13:24] sch:magiccard $ gcc magic2.c; ./a.out
몇 장의 카드를 사용하시겠습니까?: 17
[1] 103974 segmentation fault (core dumped) ./a.out
[16:13:26] sch:magiccard $
```

`magic2.c`를 실행하고 17을 입력해 오류가 뜬 화면

```
$ gcc magic2.c; ./a.out
몇 장의 카드를 사용하시겠습니까?: 17
[1] 61088 segmentation fault (core dumped) ./a.out
```

스택 overflow라는 이름으로 오류가 발생하지는 않았으나, 몇 번의 자료조사를 통해 이것이 스택 overflow 에러라는 것을 알 수 있게 되었다.

2.3.1. 스택 오버플로우

버퍼 오버플로우란 일정한 크기의 자료를 저장할 수 있도록 저장된 자료공간에 해당 자료공간이 저장할 수 있는 최대 크기의 자료보다 더 큰 자료를 저장하려고 했을 때에 저장이 성공

적으로 이루어지지 않거나, 자료공간 바깥의 추적되지 않는 부분까지 영향을 미치게 되는 현상을 의미한다.

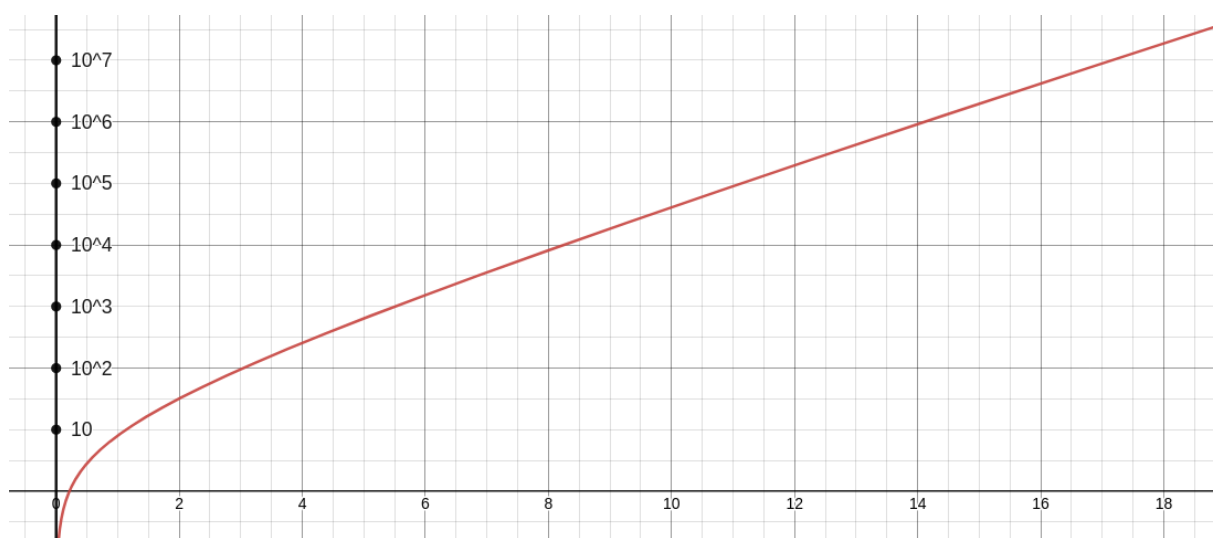
스택 오버플로우(stack overflow)는 일종의 버퍼 오버플로우이다.[출처: 1] 프로그램을 실행하면 해당 프로그램에서 실행하는 함수, 그리고 해당 함수 내에서 실행한 다른 함수를 재귀적으로 추적하기 위해 콜 스택(call stack)을 사용한다. 콜 스택에는 함수에 대한 정보와 함수에서 사용하는 변수가 저장되어있고, 일반적인 경우에 하나의 프로그램에서 사용할 수 있는 콜 스택의 크기는 정해져있다. 스택 오버플로우는 프로그램에 할당된 콜 스택의 최대 크기를 벗어나는 자료의 저장을 시도할 때 발생하는 오류이다.

위에서 작성한 것과 같이 기존에는 재귀함수를 통해서만 스택 오버플로우 오류가 발생한다고 알고 있었는데, 콜 스택에 함수에서 사용하는 지역 변수들도 모두 저장된다는 것을 알고 스택 오버플로우가 발생할 때에는 굳이 재귀함수가 필요하지 않을 수도 있다는 것을 알게 되었다.

2.3.2. 프로그램에서 사용한 메모리

`magic2.c` 에 17을 입력했을 때에 스택 오버플로우를 발생시킨 가장 큰 원인은 `main` 함수의 `cards` 변수인 것으로 보인다. 왜냐하면 다른 변수에 비해 사용자가 입력하는 수에 따라서 그 크기가 큰 폭으로 변화하기 때문이다. 사용자가 수를 입력하게 되면 `count` 변수에 그 값이 정수형으로 담기게 되고, `cards` 는 $count \times 2^{count}$ 개의 `int` 형 변수를 담는 2차원 배열을 선언한다. 이때 `count` 가 17이라면 총 2,228,224개의 정수를 담는 배열이 선언되므로, `cards` 변수 하나가 차지하는 메모리 상의 용량은 $4 \times 17 \times 2^{17}$ Byte, 대략적으로 8.9MB 정도가 된다.

같은 방식으로 `count` 가 16인 경우의 `cards` 의 용량을 구하면 4.1MB 정도가 되는 것을 알 수 있는데, 따라서 내가 개발하는 환경에서 `magic2.c` 를 컴파일한 프로그램에 할당되는 콜 스택의 크기는 4.1MB 이상 8.9MB 이하인 어떤 용량이 된다는 것을 알 수 있다.



위 그래프[그래프 1]는 `count`의 값(x 축)에 따라서 `cards`의 용량(y 축, Byte, 로그 스케일)이 어떻게 변화하는지를 나타낸다. 그래프에서 x 의 값이 증가함에 따라 y 의 값이 지수적으로 증가하는 것을 확인할 수 있다.

실제로 높은 수를 대입해가며 콜 스택의 크기를 구해본 바에 따르면, `int numbers[2096096]` 부터는 스택 오버플로우 에러가 뜨면서 프로그램이 종료된다는 것을 알 수 있었고, 이를 통한 함수에서 변수 선언으로 사용 가능한 콜 스택의 크기는 약 8.384MB 정도라는 것을 알 수 있었다.

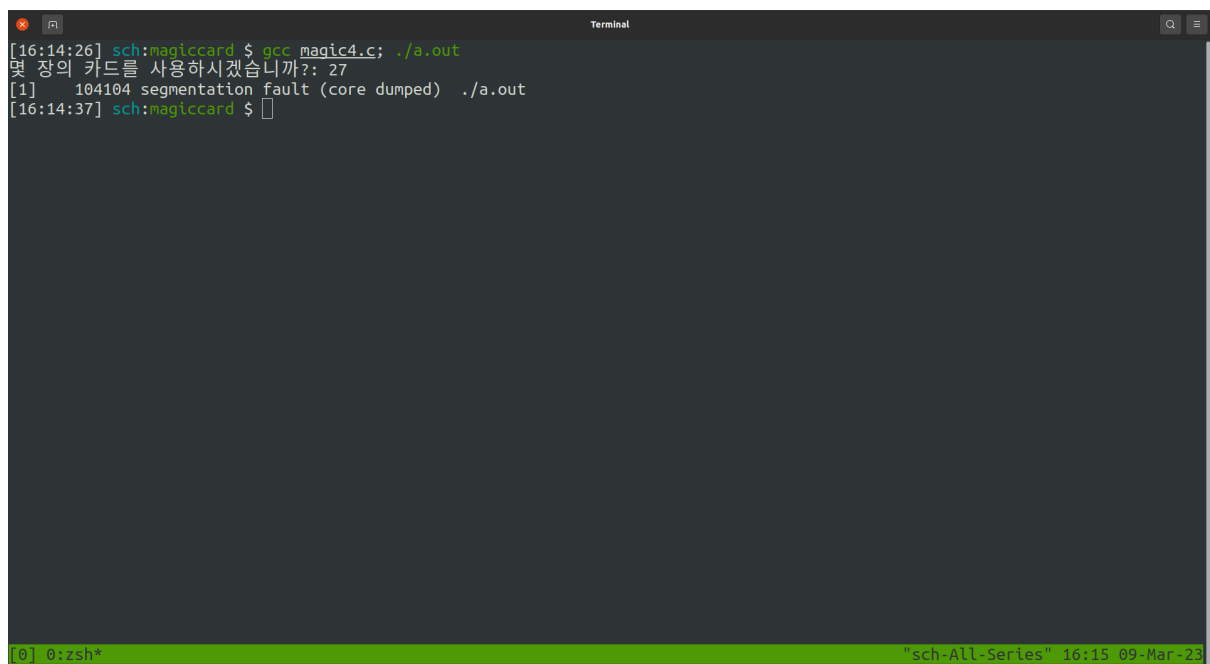
2.3.3. 스택 오버플로우의 해결 방안

- 동적 할당을 사용하는 방법

`cards` 변수를 만들 때에 동적 할당을 사용하게 되면 자동 할당으로 지역 변수를 만드는 것보다 더 큰 크기의 배열을 만들 수 있다.

`magic4.c`에서 동적 할당을 사용하여 `cards`를 할당했다. 원래 2차원 배열로 만들었던 것을 이번에는 일차원 배열로 만들어, 1차원 배열에서 인덱스를 잘 조절하여 2차원 배열처럼 사용하도록 했다. 이렇게 하면 17 이상의 수를 입력해도 정상적으로 작동되었다.

하지만 동적 할당에도 한계가 있었는데, 26을 초과하는 수를 할당하려고 하자 또 스택 오버플로우 때와 같은 에러 코드를 출력하며 프로그램이 정지했다. `count`를 27로 설정하면 할당해야 하는 메모리 크기는 14.4GB로, 프로젝트를 진행한 컴퓨터의 메모리 용량인 12GB를 넘긴 크기였다. 따라서 `magic4.c`와 같은 방법을 사용하더라도 현재 프로젝트 환경을 사용하는 한 27장 이상을 사용한 세션을 진행할 수 없었다.

A terminal window titled "Terminal" showing the execution of a program. The user runs `gcc magic4.c; ./a.out`. The program prompts "몇 장의 카드를 사용하시겠습니까?: 27". The user enters "1". The program then outputs "104104 segmentation fault (core dumped) ./a.out". The terminal prompt returns to `sch:magiccard $`. The terminal window has a green status bar at the bottom with the text `[0] 0:zsh*` and `"sch-All-Series" 16:15 09-Mar-23`.

`magic4.c`를 실행하고 오류가 뜬 화면

이 문제를 해결하는 가장 단순한 방법은 메모리 카드를 구매하여 장착하는 것이겠지만, 마술 카드 세션의 카드 장 수가 1장 늘어날 때마다 필요한 메모리의 양이 2배씩 커지기 때문에 실질적으로 바람직한 해결 방법은 아니다.

- 변수를 사용하지 않는 방법

만든 변수의 크기가 너무 커서 메모리에 담지 못하는 것이라면 값을 변수에 담지 않으면 된다.

`magic3.c`에서는 `magic2.c`의 함수 `make_cards` 함수를 수정하여, 변수를 선언하지 않고 카드를 출력하는 함수 `print_card`를 만들었다. `magic3.c`는 사용자에게 `magic2.c`와 동일한 경험을 주어주고, 17 이상의 수를 입력해도 정상적으로 작동하도록 만들어졌다.

`magic3.c`를 실행한 사진. 28을 입력하고 첫 번째 카드를 출력했다.

위 사진에서 `magic3.c`의 실행 시연을 확인해볼 수 있다. 카드가 28장인 세션을 진행했고, 스크린샷은 1번째 카드를 보여주고 관객에게 카드에 있는 수의 여부에 대해 질문하는 시점이다. 카드 장수가 28인 세션에서 나타날 수 있는 가장 큰 수인 $268435455 = 2^{28} - 1$ 이 카드의 가장 마지막에 써있는 것을 볼 수 있다.

하지만 `magic3.c`도 모든 자연수 입력을 받을 수 있는 것은 아니다. 프로젝트를 진행한 환경에서는 `int`가 32비트 2의 보수로 사용되는데, 이 형식은 $2^{31} - 1$ 을 초과하는 수를 표현할 수 없다. 즉, 31을 초과하는 입력을 처리할 수 없다. 물론 카드에 있는 수는 음수가 아니기 때문에 `unsigned int` 형식을 사용하면 표현할 수 있는 수의 범위가 $2^{32} - 1$ 까지로 늘어날 수는 있지만, 이렇게 하면 32까지는 처리할 수 있고 32를 초과하는 수는 처리할 수 없게 된다. 이 경우에는 더 큰 정수를 표현하는 표현 방식이 필요하다.


```

[16:24:37] sch:magiccard $ gcc magic3.c; ./a.out
몇 장의 카드를 사용하시겠습니까?: 32
----- 카드 -----

생각한 수가 이 목록에 있다면 1, 없다면 0을 입력해주세요: 

```

`magic3.c`를 실행하고 32를 입력한 화면. 첫 번째 카드를 보여준 화면이다. 원래라면 `print_card` 함수의 `left_limit` 값이 정수 2,147,483,648로 설정되어야 하지만 `int`형 변수의 최대 범위에 의해 오버플로우가 일어나 `-2147483648`로 초기화되어있다. 따라서 해당 함수 안의 `for`문이 실행되지 않았다.

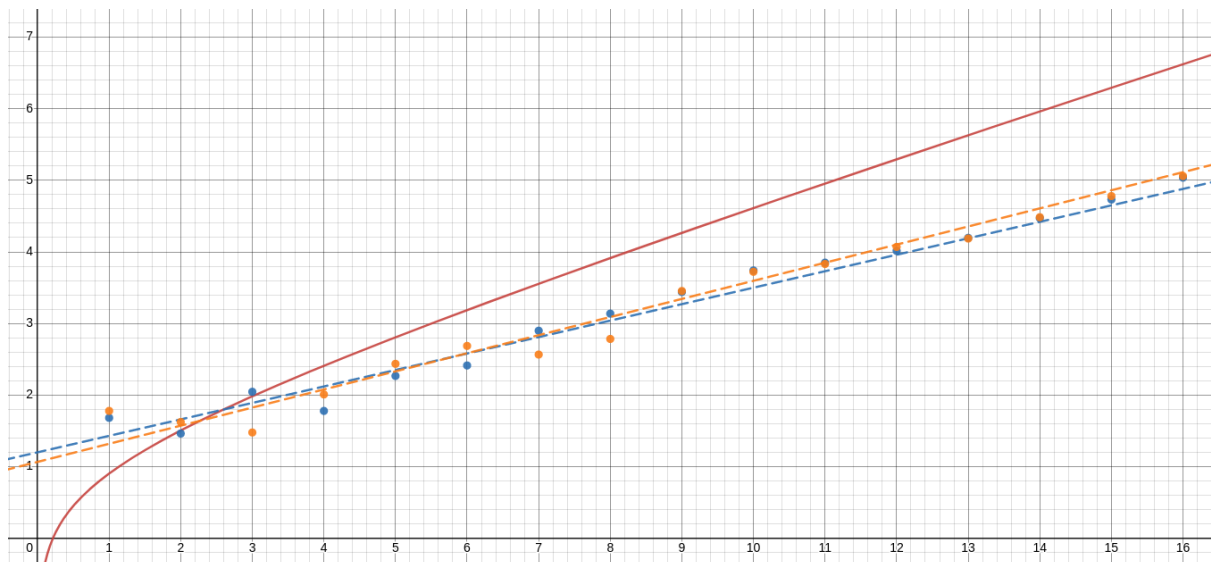
2.4. 동적 할당과 자동 할당의 실행 시간 비교

다음 표는 카드 장 수에 따른 동적 할당과 자동 할당의 프로그램 수행 시간을 비교한다. 수행 시간은 같은 컴퓨터에서 각각 `magic2_1.c`와 `magic4_1.c` 파일을 통해 진행되었다. 시간의 단위는 마이크로초이다.

카드 장수	자동 할당 (<code>magic2_1.c</code>) [μ s]	동적 할당 (<code>magic4_1.c</code>) [μ s]	<code>cards</code> 메모리 용량 [Byte]
1	48	60	8
2	29	42	32
3	111	30	96
4	60	102	256
5	185	273	640
6	259	486	1536
7	792	368	3584
8	1374	608	8192
9	2753	2842	18432
10	5507	5275	40960
11	7077	6780	90112

카드 장수	자동 할당 (magic2_1.c) [μ s]	동적 할당 (magic4_1.c) [μ s]	cards 메모리 용량 [Byte]
12	10337	11739	196608
13	15728	15376	425984
14	29682	30549	917504
15	53925	60296	1966080
16	109085	114815	4194304

아래의 그래프[그래프 2]는 세션에서 사용하는 카드의 수(x)에 따라 자동 할당 시 실행 시간[μ s](파란색), 동적 할당 시 실행 시간[μ s](주황색), cards 메모리 사용 용량[Byte](빨간색)을 구하고 각각의 자연로그를 나타낸 것이다. 그래프 1을 통해서 확인해 보았던 것과 마찬가지로, 메모리 할당 방법에 상관 없이 프로그램의 실행 시간은 자료의 크기(개수)에 따라서 지수적으로 증가하는 것을 알 수 있었다.



3. 결론

3.1. 레포트를 통해 해결한 것과 배운 것

3.1.1. 마술 카드 원리 이해

이번 레포트를 통해 마술 카드의 원리에 대해서 이해했다. 각 카드는 특정한 비트 수로 이루어진 2진수의 특정한 비트가 1인 수들을 모두 보여주며, 한 카드에 대한 정답을 들을 때마다 관객이 선택했다고 의심되는 수의 경우의 수가 반씩 줄어들어 결국 수를 특정하게 되는 것이다.

마술 카드 세션의 원리에 대해 알아보면서 파동함수 붕괴를 통한 이미지 생성 알고리즘을 상기했다. 파동함수 붕괴를 통한 이미지 생성 알고리즘이 진행되는 과정에서는 자료가 되는 이미지의 부분들이 특정한 방향에 연속될 수 있는 다른 조각의 종류를 가지거나 특정한 이미지를 기반으로 배치되는 등의 수학적 규칙을 가지는데, 이것에 기반하여 모든 확률이 중첩된 자료공간에서 무작위한 초기 상태에 따른 경우의 수 감소로 입력된 규칙에 따라 자동적으로 이미지를 생성하게 된다.

프로그램을 통해 마술 카드를 구현하면서 총 n 장의 카드로 진행되는 세션에는 모든 카드에 2^{n-1} 개의 수가 써있다고 했는데, 사실 카드에 있는 수는 이것보다 더 적을 수 있다. 마술 카드는 실물로 존재하는 카드를 가지고 세션을 진행하는 것을 상정하고 디자인된 것이므로 일반적인 경우를 상정하고 카드를 디자인하는 것이 카드의 수를 줄일 수 있어 바람직하다. 하지만 이 프로젝트를 진행하면서와 같이 마술 카드 세션을 컴퓨터 프로그램으로 구현할 수 있다면 관객이 선택했을 것으로 의심되는 모든 경우의 수의 카드 중 절반만 보여주고 이 중에 카드가 있는지 없는지를 물어보면 되므로 굳이 특정한 비트가 1인 수만을 골라서 만드는 등의 수학적 규칙이 있을 필요도 없고, 경우의 수는 점점 줄어들을 것이므로 세션의 마지막에 보여주는 카드일수록 적은 수를 포함하게 할 수 있다.

이것은 정렬되어있는 배열에서 특정한 요소의 위치를 탐색하는 이진 탐색 알고리즘에서도 적용되는 전략이다. `search.c` 파일에서는 기존의 수를 제시하는 이진 탐색 알고리즘과 달리, 관객이 생각했을 것으로 예상되는 수의 절반 중 아랫 부분의 수를 제시하며 유무를 질의한다. 질의를 통해 경우의 수를 반씩 줄여나간다는 부분에 있어서 마술 카드와 이진 탐색은 본질적으로 같은 것이다.

```

sch@sch-All-Series: ~/Desktop/magiccard
$ gcc search.c; ./a.out
질문의 횟수를 입력해주세요 (자연수): 6
0 이상 64 미만의 수 중 하나를 생각해주세요.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
당신이 생각한 수가 이 안에 있으면 1, 없으면 0을 입력해주세요.: 1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
당신이 생각한 수가 이 안에 있으면 1, 없으면 0을 입력해주세요.: 0

16 17 18 19 20 21 22 23
당신이 생각한 수가 이 안에 있으면 1, 없으면 0을 입력해주세요.: 0

24 25 26 27
당신이 생각한 수가 이 안에 있으면 1, 없으면 0을 입력해주세요.: 1

24 25
당신이 생각한 수가 이 안에 있으면 1, 없으면 0을 입력해주세요.: 0

26
당신이 생각한 수가 이 안에 있으면 1, 없으면 0을 입력해주세요.: 0
당신이 생각한 수는 27입니다.

sch@sch-All-Series: ~/Desktop/magiccard
$
  
```

`search.c` 를 실행한 화면

3.1.2. 마술 카드의 구현

각 마술 카드는 일정한 길이의 2진수 수에서 특정한 비트가 1인 모든 수를 적어 만든다. 이것을 프로그램으로 구현해야 일반적인 장수에 대한 마술 카드 세션을 진행할 수 있기 때문에, 프로젝트를 진행하면서 마술 카드를 구현하는 것은 상당히 중요한 부분이었다.

마술 카드에 있는 수는 특정한 비트가 1이라는 것이 상당히 중요한 사실이었다. 따라서 마술 카드의 수를 2진수로 표현하고, 1인 비트를 기준으로 왼쪽과 오른쪽으로 나누는 것이 가능했다. n 장짜리 세션에서 k 번째 카드가 가진 수가 모두 2^k 의 자리를 1로 한다면 1의 왼쪽에는 $n - 1 - k$ 비트, 오른쪽에는 k 비트의 2진수 수가 형성된다. 따라서 각각을 0부터 2^{n-1-k} 까지, 0부터 2^k 까지 순회하는 반복문을 중첩하여 모든 경우의 수를 나타낼 수 있었다. 중첩된 반복문 안에서는 각각의 수를 시프트 연산과 논리합 연산으로 결합하여 후보 수를 계산해 낼 수 있었다.

학교에서 컴퓨터의 자료가 2진수로 구현되기 때문에 2진수의 성질을 이용한 연산이 계산 속도가 빠르다고 배운 바 있다. 따라서 2의 거듭제곱 연산을 1을 왼쪽으로 시프트하는 연산으로 바꾸거나, 2진수로 각 비트가 상호 배타적이라는 것이 전제된 경우에 두 수를 덧셈하는 것보다 논리합 연산을 수행하는 것이 연산 복잡도 면에서 효율적이라고 판단했다.

3.1.3. 버퍼 오버플로우와 스택 오버플로우에 대한 이해

위에서 언급한 바와 같이, 원래 스택 오버플로우는 재귀 함수에 의해서만 일어나는 것이라고 알고 있었다. 하지만 이번 실습을 통해 지역 변수 자체가 콜 스택에 저장된다는 것을 알게 되었다. 따라서 재귀함수를 사용할 때에 발생하는 편의나 개발상의 성능과 하드웨어 조건 사이에서 적합한 결정을 내려야 하겠다고 생각하게 되었다.

3.2. 레포트를 통해서도 해결하지 못한 것

- 32장 이상의 마술 카드 세션 구현

이번 레포트에서 아쉬운 부분 중에 하나는 32장 이상의 마술 카드 세션을 구현하지 못했다는 것이다. 물론 n 장짜리 마술 카드 세션에서는 2^{n-1} 개의 수를 화면에 표시해야 하므로 실질적으로 세션을 진행하는 데에 어려움이 있지만, 일반적인 수의 자연수에 대해 마술 카드 세션을 구현하겠다는 목표를 이루지 못한 것이다.

정수형 변수 2개를 활용하면 32장을 초과하는 세션도 진행할 수 있다. 하지만 그 경우에도 역시 64장을 초과하는 세션은 진행할 수 없게 되며, 정수형 변수를 3개 활용하면 64장을 초과하는 세션을 진행할 수 있으나 96장을 초과하는 세션은 진행할 수 없다.

마술 카드 세션을 진행하지 못한 것에 대한 아쉬움보다는 C언어로 32비트 정수 이상의 수를 표현하지 못했다는 것에 대한 아쉬움이 더 크다. 실제로 구현하려고 했으나, 두 수 사이에서 유동적으로 연산을 하거나 10진수로 표현하는 등의 기능을 수행하기는 상당히 어려운 것이었다.

4. 참고 문헌

1. Robert Sheldon, «stack overflow»,
<https://www.techtarget.com/whatis/definition/stack-overflow>, 2023. 3. 8 방문

4.1. 그래프

1. 전한결, Desmos를 사용해 그림, <https://www.desmos.com/calculator/eaume01mvo>,
2023. 3. 8 방문
2. 전한결, Desmos를 사용해 그림, <https://www.desmos.com/calculator/fcyk5lrbfi>,
2023. 3. 9 방문