**2020 Spring OOP Assignment Report**

과제 번호 : 1

학번 : 20190084

이름 : 권민재

Povis ID : mzg00

**명예서약 (Honor Code)**

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

**Problem 1. 온전한 사각형**

1. **프로그램 개요**
   * 사용자로부터 모눈종이의 가로 세로 크기를 입력받아서, 모눈종이의 한 대각선을 지나지 않는 사각형의 개수를 구하는 프로그램이다.
   * 이 프로그램은 STDIO (Standard Input Output)를 이용하여 사용자로부터 가로 세로 크기를 각각 입력받는다.
   * prob\_1 폴더 내의 prob\_1.cpp에 C++로 코드가 작성되어 있다.
2. **프로그램의 구조 및 알고리즘**

* **구조**
  + **int main()**
    - 사용자로부터 모눈종이의 가로 칸 수와 세로 칸 수를 입력받는다.
    - 알고리즘에 따라 온전한 사각형의 개수를 연산하여 출력한다.
  + **int gcd(int a, int b)**
    - 재귀적인 호출을 이용해 유클리드 호제법으로 최대공약수를 구하는 함수이다.
    - w와 h를 입력받아서 최종적으로 w와 h의 최대공약수를 반환한다.
* **알고리즘**
  + 대각선이 지나지 않는 사각형 구하기
    - 가로 세로 칸 수와 대각선이 지나는 사각형의 개수의 규칙성을 따져보자.
    - 편리하게 각 상황에서 대각선이 지나는 사각형의 개수를 구하기 위해, 좌표평면이라고 생각하고 아래와 같은 파이썬 코드를 작성하였다.

import math  
  
w = int(input("w >> ")) # 가로 길이 입력  
h = int(input("h >> ")) # 세로 길이 입력  
  
# 좌표평면이라고 생각할 것이다.  
total = 0 # 대각선을 지나는 사각형 개수  
f = lambda x: (h / w) \* x # 대각선의 1차 함수 꼴  
  
for i in range(1, w + 1): # 주어진 범위에서,  
 # f(i)의 올림 값에서 f(i)의 내림 값을 빼면 i번째 열에서 대각선이 지나는 사각형의 개수를 구할 수 있다.   
 total += (math.ceil(f(i)) - math.floor(f(i - 1)))   
  
print(total)

* + - 그 결과를 정리하면 아래와 같다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| (가로)X(세로) | 2X2 | 2X3 | 2X4 | 3X3 | 3X4 | 3X6 |
| 개수 | 2 | 4 | 4 | 3 | 6 | 6 |

* + - 이때, 대각선이 지나는 사각형의 개수는 (가로 칸 수) + (세로 칸 수) – (가로와 세로 칸 수의 최대공약수)와 같은 규칙을 가짐을 알 수 있다.
    - 고로, 대각선이 지나지 않는 사각형의 개수는 (가로 칸 수 \* 세로 칸수) - (가로 칸 수) + (세로 칸 수) – (가로와 세로 칸 수의 최대공약수) 이다.
  + (정수에 대한) 유클리드 호제법을 이용한 최대공약수 산출
    - (정수에 대한) 유클리드 호제법은 어떤 정수 의 최대공약수는 와 를 로 나눈 나머지의 최대공약수와 같다는 정리를 말한다.
    - 를 로 나눈 나머지가 0이 될 때 까지 이를 반복하면, 입력 받은 두 수 중 0이 아닌 수가 최대공약수가 됨을 알 수 있다.
    - 재귀 함수를 이용하면 C++에서 이를 간단하게 구현할 수 있다.
    - int gcd(int a, int b)의 상황에서 생각해보자. 입력 a, b에 대해서 b가 0이 될 때까지 재귀적으로 gcd(b, a%b)를 호출하면 유클리드 호제법에 따라 최대공약수를 구할 수 있음을 알 수 있다.
* 변수
  + **int w**
    - 모눈종이의 가로 칸 수를 저장한다.
  + **int h**
    - 모눈종이의 세로 칸 수를 저장한다.
  + **int a, int b**
    - 최대공약수를 구할 두 수를 입력받는다.

1. **토론 및 개선**
   * + 이번 문제를 해결하면서 좌표평면 위에서 생각해보는 알고리즘 또한 생각해보면서, 조금만 더 생각하면 최대공약수를 이용하는 것과 같이 더 효율적인 알고리즘을 설계할 수 있음을 알았다.
     + int gcd(int a, int b)는 간단한 재귀함수지만, 만에 하나 과도한 재귀 호출로 인해 스택 오버플로우가 발생할 수도 있다. 해당 경우에는 반복문을 이용하여 유클리드 호제법을 구현하여 개선해야 할 것이다.
2. **참고 문헌**

* 유클리드 호제법 (개념 및 코드)
  + <https://ko.wikipedia.org/wiki/%EC%9C%A0%ED%81%B4%EB%A6%AC%EB%93%9C_%ED%98%B8%EC%A0%9C%EB%B2%95>
* 대각선을 지나는 사각형의 개수 (개념)
  + <https://m.blog.naver.com/orbis1020/220664563768>

**Problem 2. 리코더**

1. **프로그램 개요**
   * 여분의 리코더를 가진 친구들이 리코더를 도난 당한 친구들에게 리코더를 빌려줄 수 있는 상황에서, 리코더를 가져갈 수 있는 최대 학생 수를 출력시키는 프로그램이다.
   * 이 프로그램은 STDIO (Standard Input Output)를 이용하여 사용자로부터 전체 학생 수, 리코더를 도난당한 학생 수, 여분 리코더를 가지고 있는 학생 수, 리코더를 도난당한 학생들의 번호, 그리고 여분의 리코더를 가지고 있는 학생들의 번호를 각각 입력받는다.
   * prob\_2 폴더 내의 prob\_2.cpp에 C++로 코드가 작성되어 있다.
2. **프로그램의 구조 및 알고리즘**

* **구조**
  + **int main()**
    - 사용자로부터 전체 학생 수, 리코더를 도난당한 학생 수, 여분 리코더를 가지고 있는 학생 수, 리코더를 도난당한 학생들의 번호, 그리고 여분의 리코더를 가지고 있는 학생들의 번호를 입력받는다.
    - 학생 수 크기의 벡터를 만든다.
    - 함수 greedy\_max를 호출하여 얻은 리코더를 가져갈 수 있는 최대의 학생 수를 출력한다.
  + **int greedy\_max(std::vector<int>& students)**
    - Greedy algorithm을 이용해 리코더를 가져갈 수 있는 최대의 학생 수를 반환한다.
    - main 함수의 students 벡터를 참조를 통해 이용한다.
    - students 벡터를 순회하면서, 도난 당한 친구에게 앞 번호 친구나 뒷 번호 친구에게 있는 여분의 리코더를 빌려주는 작업을 수행한다.
    - 각 작업의 수행 이후, students 벡터에서 현재 인덱스로 참조한 값이 0 이상이면 수업에 리코더를 가져갈 수 있는 학생으로 +1 한다.
    - 최종적으로 리코더를 가져갈 수 있는 최대의 학생을 반환한다.
* **알고리즘**
  + Greedy algorithm으로 리코더를 가져갈 수 있는 최대의 학생 수 구하기
    - Greedy algorithm은 매 순간마다 부분적으로 최적인 선택을 통해 최적의 전체 결과 값을 근사하는 알고리즘이다.
    - 이 프로그램에서는 학생들에 대한 정보가 담겨있는 벡터 students를 순회하는 각 순간에서 최적의 선택을 통해 리코더를 가져갈 수 있는 최대의 학생 수를 구하고자 한다.
    - students 벡터는 기본적으로 0으로 초기화 되어 있고, 각 값이 학생이 리코더를 도난 당한 경우에는 -1을, 여분의 리코더를 가진 경우에는 +1이 더해진다.
    - 이제 students 벡터를 순회하면서 생각해보자.
      * 순회 중 0보다 작은 값을 가지는 학생을 만났을 경우,
        + 우선 앞 번호의 학생에게 여분의 리코더가 있는지 확인한다.
        + 왜냐하면, 앞 번호의 학생은 이미 순회를 마친 상태이기에, 앞 번호와 뒷 번호 친구 둘 다에게 모두 여분의 리코더가 있다면 미래를 위해 앞 번호의 친구에게서 여분의 리코더를 빌리는 것이 부분적인 최적 해이기 때문이다.
        + 앞 번호의 친구에게 여분의 리코더가 있다면, 앞 번호의 친구에게서 리코더를 빌리고, 앞 번호의 친구에게는 여분의 리코더가 없고 뒷 번호의 친구에게는 있을 경우 뒷 번호의 친구에게 리코더를 빌린다.
      * 원래 0 이상의 값을 가지는 학생이거나 0보다 작은 값을 가지는 학생이 리코더를 빌리는 모든 과정이 종료된 경우, 이제 학생의 값이 0 이상인지 확인한다.
      * 학생의 값이 0일 경우, 앞으로 뒷 번호에게 리코더를 빌려줄 일이 없고, 여분을 가지고 있을 경우 (값이 0보다 클 경우) 리코더를 빌려주어도 수업시간에 리코더를 가져간다는 사실은 자명하므로, 학생의 값이 0 이상이기만 하면 최종적으로 리코더를 가져갈 학생 수로 추가하는 것이 합리적임을 알 수 있다.
* 변수
  + **int student\_num, int loss\_num, int extra\_num, int result\_num**
    - 각각 전체 학생 수, 리코더를 잃어버린 학생 수, 여분의 리코더를 가지고 있는 학생 수, 수업시간에 리코더를 가져갈 수 있는 학생 수를 저장한다.
  + **vector<int> students**
    - 전체 학생 수 만큼의 크기를 가지는 벡터로, 각 학생의 리코더 관련 정보를 저장한다.
    - 기본적으로 각 학생의 초기화된 값은 0이며, 0보다 작으면 도난당했음을, 0보다 크면 여분의 리코더를 가지고 있음을 의미한다.

1. **토론 및 개선**
   * + 이 프로그램은 전체 학생 모두에 대한 데이터를 벡터에 넣어놓고 계산하지만, 도난 당한 학생 벡터를 따로 만들어서 학생 번호만을 저장하고, 여분의 학생 번호가 1개 입력될 때마다 도난당한 학생 벡터를 탐색하여 빌려줄 수 있는지 판단하는 방법으로 저장 공간을 많이 차지하는 문제를 개선할 수 있을 것으로 보인다.
     + 이번 문제를 통해서 greedy algorithm을 통해서도 충분히 의미있는 결과값을 근사할 수 있음을 알았다.
2. **참고 문헌**

* Greedy algorithm의 정의 (개념)
  + <https://en.wikipedia.org/wiki/Greedy_algorithm>

**Problem 3. 위스키 시음**

1. **프로그램 개요**
   * 이 프로그램은 연속으로 3개 이상의 위스키를 마실 수 없는 상황에서 최대한 마실 수 있는 위스키의 양을 구하는 프로그램이다.
   * 이 프로그램은 STDIO (Standard Input Output)를 이용하여 사용자로부터 위스키 잔의 수와 각각의 위스키 잔에 들어있는 위스키의 양들을 입력받는다.
   * prob\_3 폴더 내의 prob\_3.cpp에 C++로 코드가 작성되어 있다.
2. **프로그램의 구조 및 알고리즘**

* **구조**
  + **전처리기**
    - **CACHE\_INIT**
      * 이 프로그램에서 쓰이는 캐시 벡터의 초기화 값을 지정한다.
    - **MAX(X, Y)**
      * X와 Y 중 큰 수를 반환하는 매크로이다.
      * std에 이미 max 함수가 지정되어 있으나, 편의를 위해 매크로로 재구성하였다.
  + **int main()**
    - 사용자로부터 위스키 잔의 개수와, 각 잔에 들어있는 위스키의 양을 입력받는다.
    - 위스키 잔 개수만큼의 크기를 가지는 벡터 whiskey를 만들고, 같은 크기의 벡터 cache를 만든다.
      * cache 벡터는 memoization을 위해 사용되며, cache[idx]는 whiskey\_max(idx, whiskey, cache)의 값을 가진다.
    - 벡터 whiskey에 각 위스키 양을 입력받을 때마다 push\_back 한다.
    - 함수 whiskey\_max를 호출하여 최대로 먹을 수 있는 위스키의 양을 구하고 출력한다.
  + **int whiskey\_max(const int idx, const std::vector<int>& whiskey, std::vector<int>& cache)**
    - 인덱스 값을 입력 받고, 벡터 whiskey와 cache를 참조한다.
    - 이 함수는 whiskey 벡터에서 idx 부터 끝까지의 구간에 대해 최대로 먹을 수 있는 위스키의 양을 반환한다.
    - 우선 주어진 idx로부터 끝까지의 구간의 길이를 구해서 partial\_length에 저장한다.
    - partial\_length가 2 이하인 경우,
      * 주어진 구간에서 모든 원소의 합을 반환한다.
    - partial\_length가 2보다 클 경우,
      * 아래의 3가지 케이스를 비교해서 최대의 값을 반환한다.
        + whiskey[idx + 0] + whiskey[idx + 1] + whiskey\_max(idx + 3, whiskey, cache)
        + whiskey[idx + 0] + whiskey\_max(idx + 2, whiskey, cache)
        + whiskey\_max(idx + 1, whiskey, cache)
    - idx가 cache 벡터의 범위 밖에 위치할 경우, 0을 반환한다.
* **알고리즘**
  + Dynamic Programming으로 최대의 위스키 양 구하기
    - Dynamic programming이란 큰 문제를 간단한 여러 개의 문제로 쪼개어 푸는 방법을 일컫는다.
    - k번째 위스키 잔에 담긴 위스키 양을 , n번째 이후 먹을 수 있는 최대의 위스키 양을 이라고 해보자.
    - 이때, 최대의 위스키 양 는 3가지 경우 중 최대의 값을 취하는 방법으로 구할 수 있다.
      * 각각의 에서 을 필요로 하면서 모든 경우를 커버하므로, 우리는 큰 문제를 작게 쪼개서 해결하는 Dynamic Programming 방식으로 처리하기에 적합한 것을 알 수 있다.
    - 위와 같은 식을 통해 수열에서 3번 이상 연속하지 않는 최대의 합을 구할 수 있다.
    - 하지만, 각 과정에서 함수의 잦은 호출과 연산으로 계산 과정이 엄청나게 느려지기 때문에, 우리는 memoization을 할 필요가 있다. 그래서 cache 벡터를 따로 만들어서 cache[k]에 를 저장함으로서 효율적인 연산을 할 수 있다. 위와 같은 수식에서, 같은 를 연산하는 과정이 엄청나게 많다는 사실에서, memoization을 통해 상당한 성능 개선이 있을 수 있다.
* 변수
  + **int size, int input**
    - size는 위스키 잔의 개수를 저장하며, input은 입력을 위한 임시변수이다.
  + **vector<int> whiskey**
    - 위스키 잔에 담긴 위스키의 양을 저장하는 벡터이다.
  + **vector<int> cache**
    - 함수 whiskey\_max의 결과 값을 memo하는 벡터이다.
  + **int case\_list[3]**
    - 알고리즘 문단에서 설명한 각 3가지 케이스의 결과 값을 저장하는 배열이다.
  + **int idx, int partial\_sum, int partial\_max**
    - idx로 함수 whiskey\_max가 연산할 whiskey 벡터 구간의 시작점을 설정하고, partial\_max에 현재 구간에서의 최대 위스키 양을 저장하며, partial\_length는 현재 구간의 길이를 저장한다.

1. **토론 및 개선**
   * + 이번 문제를 통해 memoization으로 효율적인 계산을 할 수 있음을 알았다.
     + 이 프로그램은 함수 whiskey\_max를 재귀적으로 호출하여 결과값을 구하는데, 스택 오버플로우를 대비해 반복문을 이용하여 재귀적인 호출 없이 결과값을 구할 수 잇도록 설계해볼 필요가 있을 것 같다.
2. **참고 문헌**

* Dynamic Programming의 정의 (개념)
  + <https://ko.wikipedia.org/wiki/%EB%8F%99%EC%A0%81_%EA%B3%84%ED%9A%8D%EB%B2%95>