

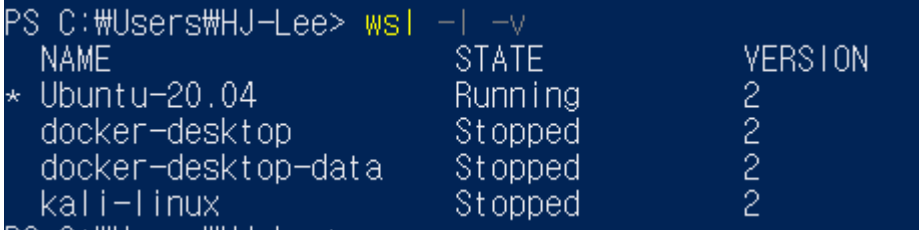
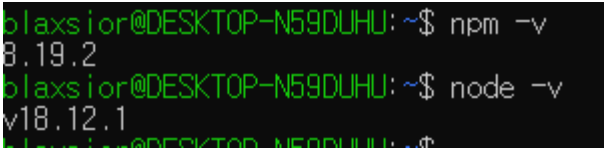
#실습 02

- brute force 구현 -

과목명	컴퓨터보안_02
교수명	
제출일	
학번	
이름	

실습 환경

실습을 진행한 환경은 다음과 같다.

장치 이름	DESKTOP-N59DUHU
프로세서	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz 3.60 GHz
설치된 RAM	16.0GB(15.9GB 사용 가능)
시스템 종류	64비트 운영 체제, x64 기반 프로세서
펜 및 터치	이 디스플레이에 사용할 수 있는 펜 또는 터치식 입력이 없습니다.
에디션	Windows 10 Home
버전	22H2
설치 날짜	2020-11-25
OS 빌드	19045.2728
경험	Windows Feature Experience Pack 120.2212.4190.0
	
사용된 언어: C++, Node.js	
- 비밀번호 생성: Node.js 환경에서 typescript 기반으로 코드를 작성. wsl2 환경.	
	
	(사용된 Node 및 npm 버전)
- brute-force: 속도를 위해 C++ 기반으로 작성, (VisualStudio 2022 사용. 환경 설정에서 utf-8, c++ 20 지정)	

위 환경에서 작성한 프로그램을 계속 구동함에 따라 이유는 알 수 없으나 성능이 점점 떨어지는 모습을 보였다. 동일한 조건에서 brute-force을 진행하는데 비교 횟수가 더 적은 비밀번호 케이스가 오히려 더 긴 시간을 소비하는 경우도 존재했다. 몇 시간 이상 컴퓨터에게 연산을 시키다 보니 컴퓨터에서 소음이 발생하여 비밀번호 7자리 이상에 대해서는 brute-force에 대한 결과를 전혀 얻지 못했다.

문제 분석

현재 과제는 키보드로 입력 가능한 ASCII 문자 중 공백을 제외한 총 94개의 문자를 이용하여 특정 조합을 가진 비밀번호를 brute-force 방식으로 푸는 것을 목적으로 한다.

ASCII 문자를 숫자, 알파벳 및 특수문자로 분류할 때 아스키 코드 분포는 다음과 같다.

- 숫자(10): 48 ~ 57
- 알파벳(52): 65 ~ 90(대문자), 97 ~ 122(소문자)
- 특수문자(32): 33 ~ 47, 58 ~ 64, 91 ~ 96, 123 ~ 126

이때 각 타입을 조합하면 7개의 비밀번호 타입이 발생한다.

- 숫자만
- 알파벳만
- 특수문자만
- 숫자 + 알파벳
- 숫자 + 특수문자
- 알파벳 + 특수문자
- 숫자 + 알파벳 + 특수문자

비밀번호 길이는 4 ~ 8로 총 5개 분류가 존재한다. 따라서 존재할 수 있는 비밀번호의 종류는 타입 개수 * 길이 개수 = $7 * 5 = 35$ 이다. 각 종류별로 10개의 비밀번호를 생성하므로 전체 비밀번호의 개수는 350개이다. 현재 과제는 이렇게 만들어진 비밀번호들에 대해 brute-force 방법을 적용하여 해결하는 것을 목표로 한다.

유저는 비밀번호의 길이와 타입을 알 수 없지만, 4 ~ 8자리 사이에 해당한다는 사실은 알고 있다는 점을 이용하여 코드를 작성한다.

프로그램에는 일정 길이의 비밀번호를 만드는 Nodejs 기반 모듈과 해당 비밀번호 목록을 받아 각 비밀번호별로 brute-force을 진행하는 C++ 기반 모듈이 요구된다. 이때 C++ 모듈의 경우 속도를 높이기 위해 스레드 등 병렬적인 동작을 구현해야 할 것으로 보인다.

프로그램 설계

비밀번호 생성 모듈

비밀번호 생성 모듈을 구상할 때 고려할 점은 비밀번호는 자신의 비밀번호 타입을 반드시 만족해야 한다는 점이다. 예를 들어 숫자 + 알파벳으로 구성되는 비밀번호가 있다고 가정하자. 단순히 랜덤 함수를 이용하여 비밀번호를 생성하면 0164처럼 숫자만 포함되는 비밀번호나 ApPle 같은 알파벳으로만 구성된 비밀번호가 발생할 수 있다. 언급된 경우는 "숫자만" 또는 "알파벳만" 포함하는 비밀번호에 해당하므로 의도와 다른 비밀번호 타입이 탄생하게 된다.

이를 방지하기 위해서는 비밀번호가 자신의 타입에 대응되는 문자를 종류별로 최소 하나씩은 가져야 한다. 예를 들어 위 예시의 경우 0A04처럼 최소 하나의 숫자와 알파벳을 포함해야 한다. 이러한 방법을 달성하기 위해서는 문자열을 만들 때 미리 **각 타입의 문자 목록에서 하나씩 꺼내 채운 후 나머지 공간에 대해 랜덤하게 문자를 할당하면 된다.** 이후 문자 목록을 랜덤하게 섞고, 이를 문자열로 변환한다. 이와 같은 동작을 위해 각 문자 타입을 구분해서 저장하고, 이를 조합하여 비밀번호를 작성할 수 있도록 구성할 필요가 있다.

brute-force 모듈

brute-force는 무차별 대입 공격 방식을 의미한다. 동작 특성상 언어의 속도가 중요하기 때문에 C++ 언어를 사용하기로 정했다. 이때 **스레드를 이용하여** 비밀번호 검사를 병렬적으로 수행함으로써 속도를 높이고자 한다.

작업 분배는 여러가지 방식이 존재한다. 전체 업무를 스레드 개수만큼 구분한 후 각 스레드에게 할당된 작업을 진행하게 할 수도 있고 (0 ~ 100, 101 ~ 200), 전체 업무에 대해 각 스레드가 작업이 겹치지 않도록 전체 스레드 개수만큼 인덱스를 더하면서 진행할 수도 있다(0 -> 8 -> 16, 1->9->17). 현재 프로젝트에서는 후자 방식으로 프로그램을 작성했다.

문자열 비교의 경우 프로그램 사용자가 비밀번호를 알 수 없는 상황을 가정한다. 따라서 비밀번호와 동일한 길이의 문자열을 만든 후 문자열끼리 비교한다. 단순히 문자열의 순서를 찾는 것이 목적이라면 비밀번호의 각 자리에 있는 문자 단위로 비교하면서 맞지 않는 경우 이상의 과정을 모두 무시하고 넘어갈 수 있지만, 현재 경우에는 반드시 N자리의 문자열을 만든 후 실제 비밀번호에 대입해야 하므로 반드시 문자열을 완성한 후에 비교를 진행해야 한다.

소스코드 및 부가 설명

비밀번호 생성 모듈

비밀번호 생성 모듈은 컴퓨팅 자원을 많이 소모하지 않으므로 문자열 처리 방식이 C++보다 편리하고, 본인이 자주 사용하는 언어인 타입스크립트를 이용하여 작성했다.

```
import {appendFile} from 'fs/promises';

const num_idxlist: [number,number][] = [[48,57]];
const alpha_idxlist: [number,number][] = [[65,90],[97,122]];
const special_idxlist: [number,number][] =
[[33,47],[58,64],[91,96],[123,126]];

// 33 ~ 47, 58 ~ 64, 91 ~ 96, 123 ~ 126
// 숫자 48 ~ 57
// 알파벳 65 ~ 90, 97 ~ 122
```

숫자, 알파벳, 특수문자에 대응되는 시작, 끝 인덱스를 담은 튜플에 대한 배열 정의

우선 숫자, 알파벳 및 특수문자를 얻기 위해 ASCII 코드 상에서 각 타입의 문자가 존재하는 인덱스 값들을 배열로 나타냈다. 예를 들어 숫자는 ASCII 기준 48 ~ 57번 인덱스에 존재한다. 이런 방식으로 총 94개의 문자가 존재하는 범위를 각각 분류했다. 이는 타이핑 실수로 특정 문자를 누락하는 경우를 방지하기 위한 과정이다.

```
/**
 * 구간에 대응되는 문자 배열을 반환한다.
 * @param chlist [start,end] 배열
 * @returns 해당 구간에 대응되는 문자 목록
 */
function charCodeGenerator(chlist: [number,number][]): string[] {
    const retarr: string[] = [];

    for(const [start, end] of chlist) {
        for(let num = start; num <= end; num++) {
            retarr.push(String.fromCharCode(num));
        }
    }

    return retarr;
}
```

구간에 대응되는 문자 배열을 반환하는 함수

charCodeGenerator 함수는 위에서 정의한 인덱스 튜플을 담은 리스트를 받아 대응되는 문자 배열을 반환하는 함수이다. for문 내부에서 튜플의 인덱스를 구조분해하고, 해당 인덱스 범위에 있는 변수들을 반환할 배열에 넣는다.

```

/**
 * 비밀번호 목록을 입력된 문자 배열 기반으로 랜덤으로 생성하고, '\n'으로 붙인
 문자열을 반환한다.
 * @param count 비밀번호의 길이. 4 ~ 8 개로 지정하자.
 * @param chlist 입력하는 문자 배열
 * @param total_count 만드는 비밀번호의 개수. 기본 값은 10
 * @returns 비밀번호 목록을 \n 으로 이어 붙인 문자열
 */
function randomPasswordGenerator(count: number, chlist: string[][],
total_count: number = 10) {
    const retarr: string[] = [];

    const remained = count - chlist.length;
    if(remained < 0) { // 해당하는 비밀번호가 성립할 수 없음.
        return "";
    }

    for(let c = 0; c < total_count; c++) {
        const charr: string[] = [];

        for (const arr of chlist) { // 각 타입을 우선 하나씩 넣음.
            charr.push(arr[Math.trunc(Math.random() * arr.length)]);
        }

        for(let i = 0; i < remained; i++) { // 남은 비밀번호 문자 채워넣기.
            const lidx = Math.trunc(Math.random() * chlist.length); //
리스트의 인덱스
            charr.push(chlist[lidx][Math.trunc(Math.random() *
chlist[lidx].length)]);
        }

        const password = charr.sort(() => Math.random() - 0.5).join(''); //
배열 한번 더 섞기.
        retarr.push(password);
    }

    return retarr.join('\n') + '\n'; // 섞은거 문자열로 바꿔서 엔터키 붙여서
출력
}

```

입력된 문자 배열을 기반으로 비밀번호를 작성하는 함수

randomPassordGenerator 함수는 문자 배열의 배열, 비밀번호 길이 및 비밀번호의 개수를 입력 받아 대응되는 비밀번호 목록을 문자열 형태로 결합하여 반환하는 함수이다.

본격적인 작업을 진행하기 전에 입력된 문자 배열의 개수와 비밀번호의 길이를 비교한다. 만약 문자배열의 개수가 비밀번호의 길이보다 길다면, 서로 다른 문자 배열에서 최소 하나의 문자를

입력 받아야 한다는 비밀번호 특성이 망가지기 때문에 빈 문자열을 반환한다.

이후 입력된 비밀번호의 개수(total_count)만큼 순회하며 랜덤하게 비밀번호를 생성한다. 비밀번호는 반드시 서로 다른 타입의 문자 배열에 속한 문자를 최소 하나씩 가져야 하므로 해당 문자 배열을 순회하면서 (const arr of chlist 부분) 각 타입의 문자를 하나씩 넣는다.

남은 문자 remained 개는 입력 받은 문자열 목록 및 선택된 배열의 문자들을 랜덤하게 선택하여 배열에 넣는다. 하나의 비밀번호에 대한 문자 배열(charr)이 다 채워지면 랜덤하게 정렬한 후 join 함수로 붙여 비밀번호를 생성한다.

```
async function generate() {
  const nl = charCodeGenerator(num_idxlist);
  const al = charCodeGenerator(alpha_idxlist);
  const sl = charCodeGenerator(special_idxlist);
  //0 1 2 01 02 12 012
  console.log(nl, nl.length);
  console.log(al, al.length);
  console.log(sl, sl.length);
  for(let count = 4; count <= 8; count++) {
    let pwstr="";
    pwstr += randomPasswordGenerator(count,[nl]);
    pwstr += randomPasswordGenerator(count,[al]);
    pwstr += randomPasswordGenerator(count,[sl]);
    pwstr += randomPasswordGenerator(count,[nl, al]);
    pwstr += randomPasswordGenerator(count,[nl, sl]);
    pwstr += randomPasswordGenerator(count,[al, sl]);
    pwstr += randomPasswordGenerator(count,[nl,al,sl]);

    await appendFile(`password${count}.txt`, pwstr,{
      encoding: 'utf-8'
    });
  }
}

generate();
```

생성된 비밀번호 목록을 파일에 쓰는 함수

generate 함수에서는 앞서 언급한 함수들을 이용, 각 타입에 맞는 비밀번호를 10개씩 생성하여 파일에 작성하는 작업을 진행한다. for문 내부에서는 길이가 4 ~ 8인 비밀번호에 대해 숫자, 알파벳 및 특수문자 조합을 달리 한 비밀번호 목록을 만들고, 이를 appendFile을 통해 파일에 저장하고 있다.

코드를 실행하기 위해서는 컴퓨터 환경에 Node.js 및 타입스크립트 컴파일러와 언어 지원을 npm으로부터 설치해야 한다. 해당 코드를 실행하면 나오는 결과는 다음과 같다.

```
[
  '0', '1', '2', '3',
  '4', '5', '6', '7',
  '8', '9'
] 10
[
  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
  'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
  'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
  'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
  'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
  'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
  'w', 'x', 'y', 'z'
] 52
[
  '!', '"', '#', '$', '%', '&',
  "'", '(', ')', '*', '+', ',',
  '-', '.', '/', ':', ';', '<',
  '=', '>', '?', '@', '[', '\\',
  ']', '^', '_', '`', '{', '|',
  '}', '~'
] 32
65 N,6\ 65 p19\ 65 5V45)Z 65 _+i07A\ 65 5%nR}@c=
66 3t\h 66 3S23< 66 F'u91% 66 921*I+x 66 9k|d0wi$
67 U91) 67 5q6@b 67 3}r6oX 67 1C~>M27 67 7_8fw'8Q
68 3o?" 68 1-3yC 68 a7R.Zh 68 n"@I609 68 6T`5[xNq
69 1q"J 69 (]"i4 69 9n9|{h 69 {0b-uF8 69 $U{025X=
70 ;\G4 70 #w0=0 70 Q&7`R8 70 4-s@A!0 70 7["s~216
71
```

콘솔에 출력되는 결과 및 생성된 비밀번호 목록들

비밀번호는 프로그램을 실행할 때마다 다른데, 위 비밀번호들은 brute-force 이후에 보고서 첨부
를 위해 새롭게 실행한 것이므로, brute-force에 사용된 것은 아니다.


```

/**
 *
 * @param filenames 섞을 비밀번호가 담긴 파일 이름 목록
 */
async function passwordShuffle(filenamees: string[]) {
    const passwords: string[] = [];
    for (const filename of filenamees) {
        const buffer = await readFile(filename, { encoding: 'utf-8' });
        passwords.push(...buffer.trim().split('\n'));
    }
    passwords.sort(() => Math.random() - 0.5);
    console.log(passwords);
    await appendFile('shuffled_password.txt', passwords.join('\n'));
}

```

비밀번호가 담긴 파일들을 받아 해당 비밀번호들을 섞는 코드

4 ~ 8자리로 구성된 비밀번호 목록을 섞어 반환하기 위한 함수도 존재한다. generate를 통해 생성한 비밀번호 파일 목록을 받아 비밀번호들을 읽어 들여 랜덤하게 섞고 파일로 내보낸다.

brute-force 모듈

C++로 작성된 brute-force 수행 모듈은 3개 파일로 구성된다.

- brute_force.h: brute-force을 수행하는 코드가 담긴 헤더 파일
- timer.h: 각 비밀번호를 발견하는데 걸린 시간을 측정하기 위해 정의된 클래스
- main.cpp: 메인 엔트리가 존재하는 함수

brute_force.h 파일

```

#pragma once
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <thread>    // 스레드
#include <vector>    // 벡터
#include "timer.h" // 타이머
#define LISTLEN 94

// 연구 기반으로 빈도를 설정해도 괜찮을 듯?
// https://www.researchgate.net/figure/Character-count-and-frequency-within-50-547-passwords-of-eight-characters-or-less_tbl4_357876614

const char chlist[94] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                        'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
                        'J', 'K', 'L', 'M', 'N', 'O', 'P',
                        'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
                        'Z',

```

```

        'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
        'j', 'k', 'l', 'm', 'n', 'o', 'p',
        'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
        'z',

        '!', '"', '#', '$', '%', '&',
        '\'', '(', ')', '*', '+', ',',
        '-', '.', '/', ':', ';', '<',
        '=', '>', '?', '@', '[', '\\',
        ']', '^', '_', '`', '{', '|',
        '}', '~'};

// 전체 문자 94 개 리스트

```

하드 코딩 되어있는 94개의 문자 목록

C++ 모듈에서는 94개의 문자를 따로 구분하지 않고 배열 형태로 나타냈다. 사용자가 비밀번호의 길이는 알고 있어도 어떤 구조로 되어 있는지는 알 수 없으므로 하나의 배열에 문자들을 몰아도 상관없으며, 각 문자의 경우 패스워드를 생성할 때 출력된 문자 목록을 복사했으므로 누락이 발생하지 않았다. 현재 비밀번호 목록은 실제 사용자가 아니라 컴퓨터가 랜덤하게 생성하기 때문에 규칙성이 존재하지 않으므로 모든 문자는 동등한 비율로 등장하게 된다. 그러나 실제 사용자들은 자신 나름대로 가진 규칙을 통해 비밀번호를 구성하는 경우가 많으므로 실제 상황에서는 각 문자에 대한 빈도 분석을 통해 사람들이 자주 사용하는 문자일수록 낮은 인덱스에 배치하여 효율적인 brute-force을 진행할 수 있을 것으로 생각된다. 주석의 링크는 비밀번호 내에서 등장하는 문자들의 빈도 표를 보여준다.

동일한 맥락에서 실제 사용자의 비밀번호에 대해 brute-force을 진행할 때는 많은 사용자들이 자주 사용하는 단어 목록을 미리 파일 형태로 저장해둔 후 이를 먼저 테스트해볼 수 있었으나, 현재 프로젝트의 비밀번호는 경향성 없이 완전히 랜덤으로 생성되었으므로, 자주 사용하는 비밀번호 목록은 따로 구성하지 않는다.

```

/**
 * 입력받은 문자열에 대해 brute-force 을 진행하는 함수
 * @param pc 프로세서가 지원하는 개수
 * @param size 문자 경우의 수
 * @param str 비교 대상이 되는 문자열
 * @return 발견한 비밀번호의 인덱스(전체 경우의 수 기준)
 */
long long brute_force(const unsigned& pc, const long long& size, const
std::string& str)
{
    std::cout << str << " "; // 현재 비교 대상인 문자열
    const int length = str.length(); // 문자열의 길이
    //long long gap = size / pc; // 이 숫자만큼 연산 수행할 예정
    std::vector<std::thread> thread_list; // 스레드 리스트
    bool find = false; // 결과를 발견했는지 -> 스레드 for 문 탈출 조건
    long long find_idx = -1; // 발견한 인덱스 값.

```

```

    for (unsigned i = 0; i < pc; i++) // 환경이 가진 스레드 개수만큼 돌린다.
    {
        // 스레드 추가 ( 람다로 작업 시키기 )
        std::thread t([&find, &length, &pc, &str, &find_idx, &size](int
sidx)
        {
            for (long long idx = sidx; idx < size; idx+= pc) {
                if (find) {
                    break; // 발견하면 나간다.
                }
                std::vector<char> fvec(length);
                long long cur = idx; // 현재 인덱스 길이...
                bool cmp_check = true;
                // 생성될 문자열 각 자리의 인덱스 처리
                for (int j = 0; j < length; j++) {
                    fvec[length - j - 1] = chlist[cur % LISTLEN];
                    cur /= LISTLEN;
                }
                std::string result(fvec.begin(), fvec.end()); // 문자열
생성

                if (result.compare(str) == 0) {
                    find_idx = idx; // 찾은 인덱스 얻기
                    find = true; // 발견했으므로 모든 스레드 탈출
                }
            } }, i);
        thread_list.push_back(std::move(t));
    }

    for (auto& th : thread_list)
    {
        th.join(); // 모든 스레드 끝날때까지 대기하기
    }

    return find_idx;
}

```

입력 받은 문자에 대해 brute-force를 수행하고, 발견된 인덱스를 알리는 함수

brute_force 함수는 입력 받은 패스워드 str에 대해 brute-force를 수행하는 주요 함수이다. 일반적인 컴퓨팅 파워로는 brute-force를 진행하는데 애로사항이 있으므로, 스레드를 이용하여 병렬적인 탐색을 진행한다.

파라미터의 pc는 현재 프로세서가 지원하는 스레드의 개수를 의미한다. 해당 수치만큼 for문을 돌면서 스레드를 생성한다. 스레드는 C++ lambda 문법을 통해 명시된 작업을 진행하는데, 내부적으로 현재 인덱스에 대응되는 문자 배열을 구성하여 문자열로 만든 후 입력된 비밀번호인 str과 비교하는 작업을 수행한다. 이때 비밀번호가 맞다면 find = true로 지정하여 즉시 반복문을 빠져나온다. find는 스레드 외부에 정의되어 있으며 생성된 모든 스레드에 대하여 공유되므로 특정 스레

드가 비밀번호를 발견하고 해당 값을 true로 변경하면 다른 스레드들도 이에 따라 반복문을 탈출할 수 있게 된다.

비밀번호를 발견하면 해당 비밀번호가 위치한 인덱스(전체 비밀번호 가짓수에 대한 인덱스)를 반환한다. 해당 인덱스가 0 이상이라면 brute_force_all에서 관련된 내용을 출력한다.

```
/*
 * 문자열 길이와 비밀번호 리스트가 담긴 파일 이름을 입력받고,
 * 해당 비밀번호들에 대해 각각 브루트포스를 수행.
 * 파일 읽기, 시간 측정 등의 작업을 수행한다.
 * @param length 문자열 길이
 * @param path 비밀번호 파일 이름
 * @param min_len 비밀번호의 최소 길이
 * @param max_len 비밀번호의 최대 길이
 */
void brute_force_all(const std::string& path, const int& min_len = 4, const
int& max_len = 8)
{
    const auto processor_count = std::thread::hardware_concurrency();
    // 코어 개수 가져오기. 이거 기반으로 스레드 돌릴 예정.

    if (min_len > max_len)
    {
        return; // 조건에 맞을때만 처리한다.
    }

    std::ifstream stream;
    stream.open(path, std::ios_base::in); // 읽기 모드로 오픈

    long long default_size = 1; // 길이 지정하는 코드
    for (unsigned i = 0; i < min_len; i++) // 최소 길이만큼 곱해준다.
    {
        default_size *= LISTLEN; // 나중에 하드코딩에서 바꿀 수 있음.
    }
    long long inner_size = default_size;

    MyTimer timer;

    int count = 0; // 현재 어디까지 진행했는지?
    long long sum = 0; // 평균 시간 측정 용도
    int word_len = min_len;
    std::string curstr;
    while (stream >> curstr)
    {
        count++;
        if (count % 10 == 1)
        {
```

```

        cout << "case: " << count << "#####" << '\n';
        sum = 0;
    }
    timer.start(); // 작업 시작
    long long idx = -1;
    while (idx < 0 && word_len <= max_len) {
        idx = brute_force(processor_count, inner_size, curstr);
        inner_size *= LISTLEN;
        word_len++;
    }

    timer.end(); // 작업 끝
    /*다음 단계를 위해 값 초기화*/
    inner_size = default_size;
    word_len = min_len;
    /**/
    if (idx >= 0) {
        std::cout << "find[ " << curstr << " : "
            << idx << " ] "
            << timer.getTimeLapse() << std::endl; // 찾았으면 위치 출력하고
    }
    else {
        std::cout << "cannot find " << curstr << std::endl;
    }

    sum += timer.getTimeLapse();
    if (count % 10 == 0)
    {
        std::cout << "average time: " << sum / 10 << '\n';
        cout << "#####" << '\n';
    }
}
}

```

특정 파일에 담긴 패스워드 목록에 대해 brute-force을 수행, 결과를 출력하는 함수

brute_force_all 함수는 패스워드 목록이 담긴 파일을 입력 받고, 해당 비밀번호들에 대해 각각 brute-force을 진행, 각 비밀번호가 소요한 밀리 세컨드 단위 시간, 타입 별 평균 시간 등을 출력한다. 시간 측정은 timer.h에 정의된 타이머 클래스를 이용하여 측정하며, 대입 공격은 brute_force 함수를 통해 수행한다.

C++은 std::thread::hardware_concurrency() 함수를 통해 현재 컴퓨터 환경의 하드웨어 스레드 개수를 알려주는데, 현재 프로그램에서는 이 수치를 병렬가능한 수치로 간주한다. processor_count 변수는 이에 대응되는 수치로, brute_force 함수의 pc 파라미터로 전달된다.

시간 측정은 각 비밀번호에 대해 brute_force 함수가 동작하는데 걸린 시간에 대해 수행되며, 동일 타입의 비밀번호인 10개 단위로 평균 시간을 측정하기 위해 % 연산으로 현재 인덱스를 검

사한다. $\text{count} \% 10 = 1$ 인 경우 각 타입의 첫번째 인덱스이므로 sum 값을 초기화, $\text{count} \% 10 = 0$ 인 경우 각 타입의 마지막 인덱스이므로 시간의 평균을 출력한다.

입력으로 받은 min_len과 max_len을 이용하여 가장 짧은 길이부터 brute-force을 진행한다. 특정 길이의 비밀번호를 발견했다면 while문을 탈출하고, 관련된 메시지를 출력한다. 특정 길이에 대한 brute-force에 실패하는 경우 탐색 범위에 LISTLEN을 곱해 길이가 1 더 길어진 비밀번호 목록에 대해 다시 시도한다. 만약 max_len까지도 비밀번호를 찾지 못하는 경우 $\text{idx} = -1$ 로 탈출하므로 cannot find 메시지를 출력한다.

timer.h 파일

```
#pragma once
#include <chrono>

using namespace std;

class MyTimer
{
private:
    std::chrono::steady_clock::time_point st;
    std::chrono::steady_clock::time_point et;

public:
    void start()
    {
        this->st = std::chrono::steady_clock::now();
    }

    void end()
    {
        this->et = std::chrono::steady_clock::now();
    }

    /**
     * @return 밀리세컨드 단위의 시간 간격 (long long)
     */
    auto getTimeLapse() {
        auto timelapse =
std::chrono::duration_cast<std::chrono::milliseconds>(this->et - this->st).count();
        return timelapse;
    }
};
```

타이머와 관련된 기능을 담은 클래스

시간 측정과 관련된 기능을 가진 클래스이다. 시간은 `std::chrono::steady_clock`을 사용하고 있다. 해당 클래스를 이용하면 정확한 시간 간격을 반환하므로 시간 측정에 적합하다. 초기에는

std::chrono::high_resolution_clock을 이용하여 더 정밀한 시간을 측정하려고 했으나, brute-force가 정밀한 시간 안에 동작하지 않으므로 정확성을 보장하는 steady_clock을 이용했다.

getTimeLapse 함수는 타이머가 멈출 때까지 흐른 시간을 밀리 세컨드 단위로 반환한다. std::chrono에는 다양한 시간 단위가 있는데, 현재 프로그램에서는 밀리 세컨드 수준의 정밀도면 충분할 것으로 판단하여 std::chrono::milliseconds로 타입 캐스팅을 진행하고 있다.

main.cpp 파일

```
#pragma once
// #include "timer.h"
#include "brute_force.h"
#include <iostream>
int main() {
    brute_force_all("password4.txt"); // 파일, 최소 길이, 최대 길이 변경 가능
    return 0;
}
```

현재 모듈의 엔트리 역할을 수행하는 main 함수가 정의되어 있다. 비밀번호의 길이 및 비밀번호가 포함된 파일을 전달하여 brute-force을 수행할 수 있다.

실행 결과

길이 = 4인 경우

출력 포맷을 다듬기 전이라 코드 상의 출력과는 다를 수 있으나, 중심 기능은 동일하다.

```
case: 1#####
0196
find[ 0196 : 9688 ]
idx: 9688
49
4508
find[ 4508 : 3366524 ]
idx: 3366524
11858
6022
find[ 6022 : 4983694 ]
idx: 4983694
15205
0711
find[ 0711 : 61947 ]
idx: 61947
173
0227
find[ 0227 : 17867 ]
idx: 17867
67
2624
find[ 2624 : 1714376 ]
idx: 1714376
4994
9613
find[ 9613 : 7528369 ]
idx: 7528369
22851
7680
find[ 7680 : 5867856 ]
idx: 5867856
16338
2976
find[ 2976 : 1741356 ]
idx: 1741356
4840
3719
find[ 3719 : 2553707 ]
idx: 2553707
7305
average time: 8368

#####
case: 11#####
UIDZ
find[ UIDZ : 25077825 ]
idx: 25077825
75596
GwAp
find[ GwAp : 13802823 ]
idx: 13802823
38902
GddZ
find[ GddZ : 13637649 ]
idx: 13637649
40789
pBRr
find[ pBRr : 42459571 ]
idx: 42459571
235160
Rlyd
find[ Rlyd : 22846739 ]
idx: 22846739
66904
WVbF
find[ WVbF : 26856097 ]
idx: 26856097
81182
zcfw
find[ zcfw : 51005304 ]
idx: 51005304
154756
DNRI
find[ DNRI : 11003376 ]
idx: 11003376
33351
XLyP
find[ XLyP : 27600493 ]
idx: 27600493
77974
DNXE
find[ DNXE : 11003936 ]
idx: 11003936
33056
average time: 83767

#####
case: 21#####
++>>
find[ ++>> : 60445935 ]
idx: 60445935
181804
}!(~
find[ }!(~ : 76968139 ]
idx: 76968139
226030
,>$>
find[ ,>$> : 61354539 ]
idx: 61354539
185407
<>|+
find[ <>|+ : 66340478 ]
idx: 66340478
185100
%|'[
find[ %|'[ : 55629096 ]
idx: 55629096
166459
&'&)
find[ &'&) : 56256344 ]
idx: 56256344
168339
*>][
find[ *>][ : 59695348 ]
idx: 59695348
180572
;<!]
find[ ;<!] : 65489510 ]
idx: 65489510
189490
~.}}
find[ ~.}} : 77915752 ]
idx: 77915752
235551
[^=,
find[ ^=, : 70545381 ]
idx: 70545381
202568
average time: 192132
```

현재 프로그램은 94개의 인덱스를 가진 N자리 문자열을 생성하여 비밀번호와 비교하는데, 숫자를 먼저 진행하는 등의 과정이 존재하지 않으므로 문자들이 섞이기 시작하는 경우 전체 시간은 비밀번호의 초반부를 이루는 문자들의 타입이 무엇인지에 따라 시간이 달라진다.

첫 3개 부류의 경우 명백하게 시간 차이가 발생한다. 인덱스로 비교하면 숫자는 0~10을, 알파벳은 11 ~ 62, 특수문자는 63~94에 분포하므로 비밀번호의 초반부를 구성하는 문자 중 특수문자의 비율이 높아질수록 비밀번호를 발견하는데 까지 걸리는 시간이 길어진다. 실제로도 각 타입은 배 이상의 평균 시간을 소모하고 있는 모습을 볼 수 있다.

<pre> case: 31##### 73M0 find[73M0 : 5842664] idx: 5842664 17862 2qP5 find[2qP5 : 2122995] idx: 2122995 6479 1kQ2 find[1kQ2 : 1239486] idx: 1239486 3596 8KD3 find[8KD3 : 6822617] idx: 6822617 19913 szb8 find[szb8 : 45394018] idx: 45394018 130305 U53Y find[U53Y : 24962016] idx: 24962016 75910 kT4x find[kT4x : 38463543] idx: 38463543 116973 3006 find[3006 : 2706266] idx: 2706266 7845 6DY4 find[6DY4 : 5101572] idx: 5101572 15326 LUw2 find[LUw2 : 17712798] idx: 17712798 51113 average time: 44532 </pre>	<pre> case: 41##### 4`9+ find[4`9+ : 4109658] idx: 4109658 11752 -5,% find[-5,% : 61514324] idx: 61514324 186223 2[0 find[2[0 : 2411852] idx: 2411852 7179 1@48 find[1@48 : 1564356] idx: 1564356 4628 :7)2 find[:7)2 : 64023402] idx: 64023402 177955 +}&1 find[+}&1 : 60621259] idx: 60621259 180603 89@2 find[89@2 : 6732000] idx: 6732000 20103 8`!2 find[8`!2 : 7436906] idx: 7436906 20664 %5^5 find[%5^5 : 54870907] idx: 54870907 163751 3<80 find[3<80 : 3190548] idx: 3190548 9516 average time: 78237 </pre>	<pre> case: 51##### KY_# find[KY_# : 16920440] idx: 16920440 50686 w%]p find[w%]p : 48765183] idx: 48765183 146288 pD.Y find[pD.Y : 42481736] idx: 42481736 127475 %*}t find[%*}t : 55454603] idx: 55454603 166637 ;/IF find[;/IF : 65461521] idx: 65461521 184558 ng,k find[ng,k : 41076636] idx: 41076636 123631 Y\$HP find[Y\$HP : 28815819] idx: 28815819 86730 rx-; find[rx-; : 44549310] idx: 44549310 125478 Ee&d find[Ee&d : 11987953] idx: 11987953 34531 a'c* find[a'c* : 30505515] idx: 30505515 92621 average time: 113863 </pre>	<pre> case: 61##### 5R!5 find[5R!5 : 4397325] idx: 4397325 12209 (?c5 find[(?c5 : 58038425] idx: 58038425 160793 c0,V find[c0,V : 31569085] idx: 31569085 86995 W+R5 find[W+R5 : 71238375] idx: 71238375 215370 7Q(5 find[7Q(5 : 6050315] idx: 6050315 18293 `5H% find[`5H% : 73967820] idx: 73967820 228632 ^SW6 find[^SW6 : 72516212] idx: 72516212 226513 Oo`t find[Oo`t : 450221] idx: 450221 1409 5P]7 find[5P]7 : 4381911] idx: 4381911 14021 }u5 find[}u5 : 76401325] idx: 76401325 239563 average time: 120379 </pre>
---	---	---	---

특수 문자가 섞이는 41번째 이상의 비밀번호에 대해서는 특수문자가 어디에 위치하는지에 따라 전체적인 시간 비율이 달라지므로 이를 비교하는 것은 큰 의미가 없다.

길이 = 4인 비밀번호 70개에 대해 걸린 밀리 초는 6,412,780초로, 분으로 환산하면 107분 정도가 소요되었다. 비밀번호를 발견한 인덱스 위치는 해당 비밀번호를 발견할 때까지 수행한 비교 연산수라고 생각할 수 있으므로, 1 ~ 10번째 비밀번호에 대한 밀리 초당 연산수를 구하면 (27845384 / 8368 / 10) = 333이 나온다. 이 정보를 기반으로 하여 현재 컴퓨터가 밀리 초당 330번의 연산력을 가졌다고 간주하자.

4자리 비밀번호를 찾기 위해 최악의 경우에 연산해야 하는 횟수는 $94^4 = 78,074,896$ 으로, 이를 기준으로 걸리는 시간은 $78,074,896 / 330 = 236591$ 밀리 초이며, 전체 70개 비밀번호를 찾기 위해 걸리는 시간을 분으로 환산하면 $(236591 * 70) / 1000 / 60 = 276$ 분이라는 결과가 나온다. 실제 시간은 해당 수치의 절반 이하로 걸렸다는 것을 감안하면 최악의 시간을 볼 일은 없는 편이다.

따라서 중위 값인 138분을 기준으로 비밀번호의 길이가 5 이상인 경우에 대해 걸리는 시간을

생각해보자. 비밀번호의 자릿수가 1씩 증가할 때마다 연산이 94배 증가한다는 것을 감안하면 각 타입의 비밀번호를 전부 찾는 데 걸리는 시간의 중위 값은 다음과 같다.

- 길이 5: 12972분 = 9일
- 길이 6: 1219368분 = 846일 = 2.3년
- 길이 7: 114,620,592분 = 79528일 = 217년 (길이가 8인 경우는 언급할 필요도 없음.)

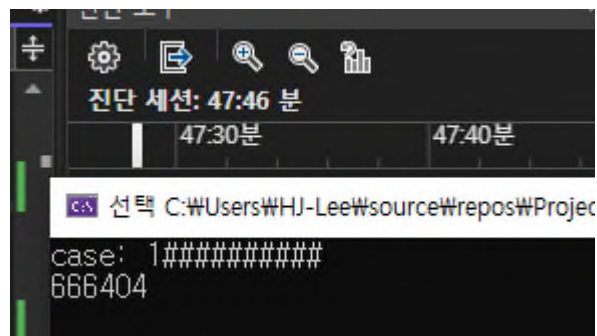
이러한 추세를 감안하면, 현재 환경에서는 길이 5 이상의 비밀번호에 대해서는 24시간 동안 컴퓨팅 자원을 전부 사용하더라도 다음 실습 전까지 비밀번호를 전부 찾아내는 것이 불가능함을 알 수 있다. 더 좋은 컴퓨팅 자원이나 병렬성 또는 효과적이고 참신한 알고리즘을 발견하지 않는 이상 이 이상 탐색을 진행하는 것은 의미가 없는 수준이라고 할 수 있다. 따라서 $n=5$ 이상의 비밀번호 일부에 대해서만 brute-force을 시도해본다.

길이 = 5인 경우

```
case: 1#####
53129 find[ 53129 : 392875265 ] 1197562
43950 find[ 43950 : 314871330 ] 1387829
74127
```

가장 간단한, 숫자로만 구성된 비밀번호를 찾는 데 대략 20분의 시간이 소요되었다.

길이 = 6인 경우



위 경우의 2배 이상 시간을 들였지만 나오지 않는다. 당연한 것이, 길이가 6인 비밀번호를 발견하는데 걸리는 시간의 기대값은 길이가 5인 경우의 94배이므로 대충 생각해도 $20 * 94 / 60 = 31.3$ 시간 정도가 소요된다. 이 시점부터는 탐색 자체가 무의미하므로 brute-force을 중단한다.

실제로 시간을 구하기 위해서 94진법의 숫자 666404를 10진법으로 바꿔 인덱스를 확인하자.

$666404(94) = 6 * 94^5 + 6 * 94^4 + 6 * 94^3 + 4 * 94^2 + 4 = 44,507,709,572$ 이므로, 이를 330으로 나누고 분에 대해 변환해보면 $A / 330 / 1000 / 60 = 2248\text{분} = 37\text{시간}$ 정도가 소요된다. 따라서 현재 조건에서 전체 비밀번호는 사실 상 찾을 수 없다.

소감

brute-force을 구현하면서 느낀 점은 확실히 비밀번호에 사용되는 문자 도메인의 범위가 넓어질 수록 비밀번호를 알아낼 가능성이 낮아진다는 점이다. 대부분의 비밀번호 해제 툴들은 효율성 문제로 일단 숫자 + 알파벳 조합으로 비밀번호에 대한 brute-force을 진행할 것으로 보이는데, 이 경우에는 비밀번호가 한자리 증가할 때마다 경우의 수가 62배 증가한다. 반면 특수문자 32개를 추가하는 경우 문자 하나 당 94배씩 증가하므로 탐색해야 하는 비밀번호의 수가 기하급수적으로 증가한다고 볼 수 있다.

현재 알고리즘에서 간단한 비밀번호를 찾는 것을 목적으로 하는 경우 각 비밀번호의 타입처럼 숫자 / 숫자 + 알파벳 조합 등 도메인을 제한하여 brute-force을 순서대로 진행할 수 있다. 다만 이 방식은 특수문자가 포함되기 시작하는 순간부터 오히려 탐색 횟수를 증가시키므로 비효율적인 방식이 될 수 있다. 이런 점을 고려하면 비밀번호에 상상하지 못한 특수문자를 추가하는 것이 비밀번호 보안에 큰 도움이 될 수 있을 것으로 보인다. 만약 emoji나 한글 등이 허용되는 환경이라면 예측하기 매우 어렵기 때문에 사실상 뚫을 수 없는 비밀번호가 될 수 있다.

다만 사람들은 자신과 관련된 정보로 비밀번호를 구성하는 경우가 많으며, 사용하는 특수문자가 어느 정도 정해져 있다는 점에 있어서 실제 환경에 대한 brute-force는 생각보다 빠르게 종료될 수도 있다. 여전히 비밀번호에 대한 특수문자를 추가를 강제하지 않는 기업이 많으므로 이러한 기업에 대해서는 도메인 자체를 줄여서 brute-force을 수행할 수 있으므로 현재 걸리는 시간의 절반 이하로 감소할 수 있다.

C++ 측면의 경우, 정말 새로운 경험이 많았다. 나는 웹 관련 공부를 주로 하기 때문에 C++ 코드를 거의 1년 만에 만져봤다. 그렇다 보니 iostream, chrono, thread 등을 어떻게 사용하는지 거의 처음부터 다시 배운 셈이 됐다. 특히 스레드나 lambda 문법의 경우 알고는 있었지만 실제로 사용해본 적은 없었는데, 고학년이 돼서 사용해보니까 과거와는 달리 이러한 기능이 왜 필요한지 느껴졌다. 특히 과거에는 lambda 식에 왜 캡처와 인자를 위한 공간이 따로 있는 것인가 이해가 되지 않았는데, 지금 사용해보니 정말 편하다는 생각이 든다. 내가 자주 사용하는 언어인 자바스크립트는 현재 환경에서 호출하는 변수가 현재 환경에 없으면 상위 환경으로 이동하여 찾는데, lambda의 캡처를 이용하니 이런 기능을 유사하게 사용할 수 있었다. 또한 std::move나 std::ref 등의 함수를 왜 사용해야 하는지 동의하기 힘들었는데, 복사 생성자를 제거한 std::thread와 람다식을 사용하다 보니까 이걸 왜 만든 것인지 느껴졌다. 언어적 측면에서는 C++은 알면 알수록 할 수 있는 범위가 매우 넓은 것 같다.

그럼에도 C++을 사용하고 싶지 않은 이유는 컴파일 문제가 크다. 당장 이번 과제를 진행할 때 컴파일 과정에서 이유를 알 수 없는 링커 문제가 발생하여 1시간 이상을 끙끙 앓았는데, 컴퓨터를 껐다 켜니 해당 증상이 사라졌다. 이런 열불나는 상황만 발생하지 않으면 매력적인 언어인데, 당장 visual studio 같은 IDE도 이런 상황이라, 좀처럼 손이 가지 않는다. 이런 점을 감안해도 꽤 많은 경험이었지만, 장기적으로는 성능 문제와 무관한 과제가 나오면 더 좋을 것 같다.