Object-Oriented Programming Lab #09

Department: 응용물리학과 Student ID: 2017103038 Name: 권인회

A. Code explanation & output analysis (Write the source code and results)

A-1. Listing 12.5

< 코드 >

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <ctime>
/*
* binary_search(a, seek)
* Returns the index of element seek in vector a;
* returns -1 if seek is not an element of a
* a is the vector to search; a's contents must be
* sorted in ascending order.
* seek is the element to find.
*/
int binary_search(const std::vector<int>& a, int seek) {
        int n = a.size();
        int first = 0, // Initially the first element in vector
                last = n - 1, // Initially the last element in vector
                mid; // The middle of the vector
        while (first <= last) {
                mid = first + (last - first + 1) / 2;
                if (a[mid] == seek)
                        return mid; // Found it
                else if (a[mid] > seek)
                        last = mid - 1; // continue with 1st half
                else // a[mid] < seek
                        first = mid + 1; // continue with 2nd half
        }
        return -1; // Not there
}
/*
* linear_search(a, seek)
* Returns the index of element seek in vector a.
* Returns -1 if seek is not an element of a.
* a is the vector to search.
* seek is the element to find.
* This version requires vector a to be sorted in
* ascending order.
*/
int linear_search(const std::vector<int>& a, int seek) {
        int n = a.size();
        for (int i = 0; i < n && a[i] <= seek; i++)
                if (a[i] == seek)
                        return i; // Return position immediately
        return -1; // Element not found
```

```
}
/*
* Tests the execution speed of a given search function on a
* vector.
* search - the search function to test
* v - the vector to search
* trials - the number of trial runs to average
* Returns the elapsed time in seconds
* The C++ chrono library defines the types
* system_clock::time_point and microseconds.
*/
double time_execution(int (*search)(const std::vector<int>&, int),
        std::vector<int>& v, int trials) {
        int n = v.size();
        // Average the time over a specified number of trials
        double elapsed = 0.0;
        for (int iters = 0; iters < trials; iters++) {
                clock_t start_time = clock(); // Start the timer
                for (int i = 0; i < n; i++) // Search for all elements
                        search(v, i);
                clock_t end_time = clock(); // Stop the timer
                elapsed += static_cast<double>(end_time - start_time) / CLOCKS_PER_SEC;
        }
        return elapsed / trials; // Mean elapsed time per run
}
int main() {
        std::cout << "----------------------------------------₩n";
        std::cout << " Vector Linear Binary₩n";
        std::cout << " Size Search Search₩n";
        std::cout << "----------------------------------------₩n";
        // Test the sorts on vectors with 1,000 elements up to
        // 10,000 elements.
        for (int size = 0; size <= 50000; size += 5000) {
                std::vector<int> list(size); // Make a vector of the appropiate size
                // Ensure the elements are ordered low to high
                for (int i = 0; i < size; i++)
                        list[i] = i;
                std::cout << std::setw(7) << size;
                // Search for all the elements in list using linear search
                // Compute running time averaged over five runs
                std::cout << std::fixed << std::setprecision(3) << std::setw(12)
                        << time_execution(linear_search, list, 5)
                        << " sec";
                // Search for all the elements in list binary search
                // Compute running time averaged over 25 runs
                std::cout << std::fixed << std::setprecision(3) << std::setw(12)
                        << time_execution(binary_search, list, 25)
                        << " sec₩n";
        }
}
```

< 실행결과 >

1. Debug 모드에서 실행

Vector Linear Binary

Size Search Search

0	0.000 sec	0.000 sec
5000	0.777 sec	0.004 sec
10000	2.923 sec	0.008 sec
15000	6.415 sec	0.012 sec
20000	11.389 sec	0.017 sec
25000	17.785 sec	0.021 sec
30000	25.648 sec	0.026 sec
35000	35.121 sec	0.030 sec
40000	45.670 sec	0.036 sec
45000	57.988 sec	0.040 sec
50000	72.195 sec	0.052 sec

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 91684개)이(가

) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

2. Release 모드에서 Optimization disabled 한 이후 실행

Vector Linear Binary

Size Search Search

0	0.000 sec	0.000 sec

5000	0.085 sec	0.001 sec
10000	0.305 sec	0.001 sec
15000	0.689 sec	0.002 sec
20000	1.192 sec	0.003 sec
25000	1.912 sec	0.004 sec
30000	2.746 sec	0.004 sec
35000	3.707 sec	0.005 sec
40000	4.804 sec	0.006 sec
45000	6.095 sec	0.007 sec
50000	7.725 sec	0.008 sec

C:₩Users₩Administrator₩source₩repos₩Ex002₩Release₩Ex002.exe(프로세스 90672개)이(

가) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

Release 모드에서 시간 차이를 확인하려면 Property Pages 에서 Optimization 에서 disable 하면 됨 (debug보다 상대적으로 당연히 빠름), Maximize Speed(O2) 기본값 상태에서는 모두 0초가 나옴

< 설명 >

이 코드는 기본적으로 수업시간에 배웠던 검색 방식인 Linear search와 binary search의 실행시간을 비교하는 목적을 갖고 있다. 크게 4부류로 나누어 설명할 것이다.

1. return 값을 갖는 binary_search 함수

이 함수는 int형 벡터와 int형 변수를 parameter로 받아들인다. 함수 내에서 초기화된 변수 n은 parameter로 받아들인 벡터의 사이즈가 할당된다. 그리고, first를 0, last를 n-1로 초기화하고 mid를 선언해준다. 이 세 변수는 index 번호로 사용하기 위함이다. First는 첫 인덱스, last는 벡터 맨 끝 요소의 인덱스를 의미한다. 그래서 first가 last보다 작거나 같은 경우에 while문을 실행하게 되는데, 여기서 mid는 first와 last 사이의 중간값을 말한다. 따라서, 두 변수의 합을 2로 나누어주는

데, 소수가 나올 가능성을 방지하기 위해 괄호처럼 따로 분리하여 그러지 않게끔 설정하였다. 이후, 벡터에서 정한 중간 위치의 값이 2번째 parameter로 받은 찾는 값 변수와 동일하면 바로 그 값이 있는 index (mid)를 return 한다. 만약, index mid에 위치한 값이 찾는 값보다 크다면 last에 이 mid에서 1뺀 값을 할당해주고, 작다면 first에 mid에서 1더한 값을 할당해준다. 마치 up and down 게임과 방식이 동일하다. 예를 들어, 찾는 값이 1~50 사이에서 30이면, 범위의 중간값인 25가 찾는 값보다 작으므로 26~50으로 범위가 새로 지정되는 것이다. 만약, 이 while문의 결과를 통해서 아무 결과가 return 되지 않는다면 이 벡터에 찾는 값이 존재하지 않는 것이다. 따라서, -1를 return 해준다. 물론 이 방법은 벡터 내 요소들이 오름차순 된 경우에만 사용 가능하다.

2. return 값을 갖는 linear_search 함수

이 함수는 int형 벡터와 int형 변수를 parameter로 받아들인다. 함수 내에서 초기화된 변수 n은 parameter로 받아들인 벡터의 사이즈가 할당된다. for문을 이용해서, i는 0부터 1씩 증가시켜가면서 벡터의 사이즈보다 작고, 벡터의 i번째에 있는 요소가 찾는 값보다 작거나 같은 동안 for문을 돌린다. 여기서 만약 벡터의 i번째에 있는 요소가 찾는 값과 같다면 그 index인 i를 return 한다. 이 과정을 통해서도 return 되지 않는다면 -1이 return 된다. 말 그대로 이것은 처음부터 찾는값이 나올때까지 찾는 것이다. 이것도 오름차순이 된 경우에만 사용 가능한 코드이다. (a[i] <= seek가 포함되었기때문, 원래 linear search는 오름차순이 필요 없는 경우도 있다.)

3. return 값을 갖는 time_execution 함수

이 함수는 int형 벡터와 int형 변수를 parameter로 받는 함수의 포인터, int형 벡터, int형 변수 이렇게 총 3개의 parameter를 받아들인다. 함수 내에서 초기화된 변수 n은 parameter로 받아들인 벡터의 사이즈가 할당된다. 시간 측정을 위해 double형으로 변수를 0.0 초기화해준다. for문에 들어가면서 iters 변수를 0부터 1씩 늘려가면서 3번째 parameter로 받은 변수의 값이 되기 전까지 실행한다. 이 첫번째 for문 안에서는 clock_t 타입으로 선언된 변수 start_time에 현재 시각(clock())을 할당해준다. 이후, 두번째 for문을 통해 parameter로 받은 함수를 0부터 벡터의 사이즈가 되기 전까지 계속 1씩 증가시키면서 실행시킨다. 벡터 안에 있는 요소들에서 0부터 벡터의 사이즈-1 값까지 차례대로 1씩 증가시키면서 찾는 것이다. (이를 통하기 위해서는 벡터의 요소들이 0부터 1씩 증가되는 형태로 들어가있어야한다.) 이후 두번째 for문을 다 돌고 나면 clock_t 타입으로 선언된 변수 end_time에 현재시각을 할당해준다. 그리고 end_time에서 start_time을 빼주면서 걸린 시간을 구하고 double형으로 캐스팅 해준 다음에 이 시간이 초 단위가 아니기 때문에 이미 라이브러리에서 정의되어있는 CLOCKS_PER_SEC을 이용해서 초 단위로 바꿔준다. 이 값을 elaspsed 변수에 더해준다. 더해주는 이유는 위와 같은 과정을 실행할때마다 걸린 시간이 바뀔 수 있으므로 표본을 늘리고 평균을 구하기 위해 3번째 parameter로 받은 trial이 시도 횟수이다. 이 시도횟수만큼 반복해서 걸린 시간의 평균값을 초 단위로 return해주는 것이다.

4. main 함수

Size 변수를 0부터 5000 단위로 50000까지 올리면서 for문을 실행할 것이다. Int형 벡터 list를 size 변수 크기만큼 선언해주고, 이 벡터 안에 for문을 이용해서 요소를 할당해준다. 요소의 값은 index 값과 동일하다. (i는 0부터 size-1까지 증가하면서 요소마다 할당) 그리고 도표형태를 갖추기 위해 setw로 조절해주고 size를 먼저 알려준다. (첫 열을 표시해줌) 그리고 소수점 세자리까지 표현되도록 하고 setw로 일정 간격 이후에 앞서 정의한 time_execution함수에 parameter로 linear_search를 먼저 넣고 trial 횟수는 5로 설정하고 실행한다. 그렇게 linear_search 함수를 5번 실행하면서 걸린 시간의 평균 값이 초 단위로 출력될 것이다. (함수 내에서는 list 벡터 안에 있는 요소를 하나하나 차례대로 찾는 과정을 반복, 단 방법은 다름) 이후에 binary_search 함수를 같은 방법으로 실행하되, trial 횟수를 25로 설정한다. 아마 상대적으로 더 조금 걸리기 때문에 더 높은 정확도를 위해 많은 횟수를 시도한 것 같다. 이런식으로 실행되어 위와 같은 실행결과를 불러온다.

A-2. Listing 12.9

< 코드 >

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
/*
* print
* Prints the contents of an int vector
* a is the vector to print; a is not modified
* n is the number of elements in the vector
*/
void print(const std::vector<int>& a) {
        int n = a.size();
        std::cout << "{";
        if (n > 0) {
                std::cout << a[0]; // Print the first element
                for (int i = 1; i < n; i++)
                        std::cout << ',' << a[i]; // Print the rest
        }
        std::cout << "}";
}
/*
* Returns a pseudorandom number in the range begin...end - 1,
* inclusive. Returns 0 if begin >= end.
*/
int random(int begin, int end) {
        if (begin >= end)
                return 0;
        else {
                int range = end - begin;
```

```
                return begin + rand() % range;
        }
}
/*
* Randomly permute a vector of integers.
* a is the vector to permute, and n is its length.
*/
void permute(std::vector<int>& a) {
        int n = a.size();
        for (int i = 0; i < n - 1; i++) {
                // Select a pseudorandom location from the current
                // location to the end of the collection
                std::swap(a[i], a[random(i, n)]);
        }
}
/* Randomly permute a vector? */
void faulty_permute(std::vector<int>& a) {
        int n = a.size();
        for (int i = 0; i < n; i++) {
                // Select a pseudorandom position somewhere in the collection
                std::swap(a[i], a[random(0, n)]);
        }
}
/* Classify a vector as one of the six permutations */
int classify(const std::vector<int>& a) {
        switch (100 * a[0] + 10 * a[1] + a[2]) {
        case 123: return 0;
        case 132: return 1;
        case 213: return 2;
        case 231: return 3;
        case 312: return 4;
        case 321: return 5;
        }
        return -1;
}
/* Report the accumulated statistics */
void report(const std::vector<int>& a) {
        std::cout << "1,2,3: " << a[0] << '₩n';
        std::cout << "1,3,2: " << a[1] << '₩n';
        std::cout << "2,1,3: " << a[2] << '₩n';
        std::cout << "2,3,1: " << a[3] << '₩n';
        std::cout << "3,1,2: " << a[4] << '₩n';
        std::cout << "3,2,1: " << a[5] << '₩n';
}
/*
* Fill the given vector with zeros.
* a is the vector, and n is its length.
*/
void clear(std::vector<int>& a) {
        int n = a.size();
        for (int i = 0; i < n; i++)
                a[i] = 0;
}
int main() {
        // Initialize random generator seed
```

```
        srand(static_cast<int>(time(0)));
        // permutation_tally vector keeps track of each permutation pattern
        // permutation_tally[0] counts {1,2,3}
        // permutation_tally[1] counts {1,3,2}
        // permutation_tally[2] counts {2,1,3}
        // permutation_tally[3] counts {2,3,1}
        // permutation_tally[4] counts {3,1,2}
// permutation_tally[5] counts {3,2,1}
        std::vector<int> permutation_tally{ 0, 0, 0, 0, 0, 0 };
        // original always holds the vector {1,2,3}
        const std::vector<int> original{ 1, 2, 3 };
        // working holds a copy of original that gets permuted and tallied
        std::vector<int> working;
        // Run each permutation one million times
        const int RUNS = 1000000;
        std::cout << "--- Random permute #1 -----₩n";
        clear(permutation_tally);
        for (int i = 0; i < RUNS; i++) { // Run 1,000,000 times
        // Make a copy of the original vector
                working = original;
                // Permute the vector with the first algorithm
                permute(working);
                // Count this permutation
                permutation_tally[classify(working)]++;
        }
        report(permutation_tally); // Report results
        std::cout << "--- Random permute #2 -----₩n";
        clear(permutation_tally);
        for (int i = 0; i < RUNS; i++) { // Run 1,000,000 times
        // Make a copy of the original vector
                working = original;
                // Permute the vector with the second algorithm
                faulty_permute(working);
                // Count this permutation
                permutation_tally[classify(working)]++;
        }
        report(permutation_tally); // Report results
}
```

< 실행결과 >

--- Random permute #1 -----

1,2,3: 166011

1,3,2: 166860

2,1,3: 167141

2,3,1: 166741

3,1,2: 166449

3,2,1: 166798

--- Random permute #2 -----

1,2,3: 148145

1,3,2: 185342

2,1,3: 184582

2,3,1: 185189

3,1,2: 148054

3,2,1: 148688

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 110732개)이(

가) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

< 설명 >

이 코드는 랜덤 순열을 조합할 때 잘못된 방법과 올바른 방법을 비교해준다. 이전과 다르게 main 함수부터 살펴보고자 한다.

Srand함수를 통해 랜덤 시드를 현재 시간으로 맞춰 실행할때마다 랜덤 시드가 바뀌게 한다. 그렇게해야 코드를 실행할때마다 랜덤값이 변경된다. 이후, permutation_tally라는 int형 벡터를 size 6으로 초기화해준다. 이것은 우리가 총 조합할 순열이 6가지이므로 이것이 나온 횟수를 저장하기 위한 벡터이다.

이후 상수로 선언된 original int형 벡터를 {1,2,3}으로 초기화해준다. working이라는 int형 벡터도 선언해주고 for문을 돌릴 횟수 RUNS를 int형 상수로 1000000이라 초기화해준다.

이제부터 랜덤 순열 조합을 시작한다. clear함수를 통해 permutation_tally 벡터의 요소들을 다 0으로 초기화해준다. 이후 첫번째 for문을 들어가는데 i가 0에서 1씩 증가하며 1000000이 될때까지 진행한다. Working 벡터에 original 벡터를 할당해주고 permute함수에 인자로 넣는다. 그렇다면 permute 함수를 살펴보자.

Permute 함수는 인자로 받은 벡터의 사이즈를 int형 변수 n에 할당해주고 for문을 돌린다. I가 0에서 1씩 증가하며 벡터의 사이즈(여기서는 무조건 3이다.) -1이 되기 전까지 돌린다. 이 경우에는 사이즈가 3이므로 for문이 i=0,1일 때 총 2번 실행된다. I=0일 때 표준 라이브러리 함수 swap을

통해 벡터에 있는 0번째 요소와 index 0~2 사이에 있는 요소 중 랜덤으로 swap 된다. 그 이후, i=1일 때, 벡터에 있는 1번째 요소와 index 1~2 사이에 있는 요소 중 랜덤으로 또 swap된다. 따라서, 원래의 벡터가 그대로 나올 수도 있고 꼬일수도 있는거다. 이 과정을 통해 나올수 있는 벡터의 경우의 수가 6가지 인 것이다. {1,2,3}이 들어가서 {1,2,3},{1,3,2},{2,1,3},{2,3,1},{3,1,2},{3,2,1} 중 하나가 거의 같은 확률로 나오는 것이다. 이렇게 해서 working의 벡터가 6가지 중 하나로 바뀌어 나오면 이 working 벡터가 classify 함수에 들어간다.

Classify 함수는 working벡터 안에 들어있는 요소들의 조합에 따라서 index 번호를 지정해주기 위한 것으로, 벡터 안의 순열이 {1,2,3}이면 0,{1,3,2}이면 1,{2,1,3}이면 2,{2,3,1}이면 3,{3,1,2}이면 4,{3,2,1}이면 5를 return하여 permutation_tally의 벡터에 그에 맞는 index번호에 들어가서 1을 증가시키는 것이다. 이 과정이 1000000번 반복되어 각 순열 조합에 대해 랜덤 돌려 나온 횟수가 permutation_tally 벡터에 누적되는 것이다.

따라서, 이후 report 함수를 통해 출력이 되는데, 이 함수를 살펴보면 실행결과와 같은 형식으로 permutation_tally 벡터에 있는 요소의 값들을 출력해줄 수 있도록 한다. 이렇게 되면 첫번째 Random permute #1 과정이 끝난 것이다.

Random permute #2 과정도 똑같은데 permute대신 faulty_permute 함수를 사용한다. 이 함수를 살펴보자. 이 함수도 int형 변수 n은 파라미터로 받은 벡터의 사이즈이므로 이 경우 무조건 n=3 이다. 따라서, for문에 들어가게되면 i=0,1,2일 때의 총 3번 실행하는 것이다.

swap안에서 벡터의 0번째 요소와 0~2사이에서의 요소를 랜덤하여 바꾸고, 벡터의 1번째 요소와 0~2사이에서의 요소를 랜덤하여 바꾸고, 벡터의 2번째 요소와 0~2사이에서의 요소를 랜덤하여 바꾼다. 이렇게 되면 상대적으로 순열 조합마다 확률이 동일하지가 않다. 자세한건 확률과 통계 항목인 순열 경우의수를 따져보면 알 수 있다. 당장 {1,2,3}이 나올 경우의 수만 따져도 {1,3,2}보다 적은 것을 알 수 있다. (수학 항목의 계산에 따르면 4가지, 6가지 차이인 것으로 알고 있다.) 따라서 동일한 확률로 나오지 않기 때문에 1000000번 실행해도 모든 순열 조합이 나오는 횟수가 비슷하지 않고 차이가 심하게 나는 것이다.

설명을 안 한 부분이 있는데, swap할 때 사용하는 random 함수는 편의를 위해 사용자 정의로 만들어둔 함수로, rand()함수를 범위를 지정하여 이용할 수 있게 해둔 함수이다. 2개의 int형 인수를 받아 시작점, 끝점으로 쓰는데 시작점이 끝점과 같거나 큰 경우에는 0이 return 된다. 예를 들어, 파라미터로 (0,3)을 넣으면 0~2까지의 random 값이 나오도록 하는 것이다.

그렇기 때문에 랜덤 순열 조합을 이끌어내기 위해서는 faulty_permute 함수가 아니라 permute 함수와 같은 과정을 택해야한다.

B. Exercises (Write the questions down on your answer sheet)

(pp. 381-382), Exercises 1-9

(write output analysis for all exercises)

1. Complete the following function that reorders the contents of a vector so they are reversed from their original order. For example, a vector containing the elements 2, 6, 2, 5, 0, 1, 2, 3 would be transformed into 3, 2, 1, 0, 5, 2, 6, 2. Note that your function must physically rearrange the elements within the vector, not just print the elements in reverse order.

```
void reverse(std::vector<int>& v) {
        // Add your code...
}
```

< 코드 >

```
void reverse(std::vector<int>& v) {
        int n = v.size();
        for (int i = 0; i < (n + 1) / 2; i++)
                std::swap(v[i], v[n - i - 1]);
}
```

< 전체 코드 >

```
#include <iostream>
#include <vector>
#include <cstdlib>

void reverse(std::vector<int>& v) {
        int n = v.size();
        for (int i = 0; i < (n + 1) / 2; i++)
                std::swap(v[i], v[n - i - 1]);
}
void print(const std::vector<int>& a) {
        int n = a.size();
        std::cout << "{";
        if (n > 0) {
                std::cout << a[0];
                for (int i = 1; i < n; i++)
                        std::cout << ',' << a[i];
}
        std::cout << "}";
}

int main() {
        std::vector<int> test{ 2,6,2,5,0,1,2,3 };
        reverse(test);
        print(test);
}
```

< 실행결과 >

{3,2,1,0,5,2,6,2}

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 113564개)이(

가) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

< 설명 >

Reverse 함수에 parameter로 벡터를 레퍼런스로 받아서 int형 변수 n에 벡터의 사이즈를 넣고 사이즈의 반절을 기준으로 앞뒤를 swap 하였다. 따라서, 이 벡터 기준으로 n=8이므로 벡터의 0번째 요소와 7번째요소가 바뀌고, 1번재와 6번째, 2번째와 5번째, 3번째와 4번째가 바뀌어 원하는 벡터가 된 것이다. I=0,1,2,3일 때 실행되므로 4번 실행되니까 맞다.

2. Complete the following function that reorders the contents of a vector so that all the even numbers appear before any odd number. The even values are sorted in ascending order with respect to themselves, and the odd numbers that follow are also sorted in ascending order with respect to themselves. For example, a vector containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 would be transformed into 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 Note that your function must physically rearrange the elements within the vector, not just print the elements in reverse order.

```
void special_sort(std::vector<int>& v) {
        // Add your code...
}
```

< 전체 코드 >

```
#include <iostream>
#include <vector>
#include <cstdlib>

void sort(std::vector<int>& v) {
        int n = v.size();
        for (int i = 0; i < n - 1; i++) {
                int small = i;
                for (int j = i + 1; j < n; j++)
                        if (v[j] < v[small])
                                small = j;
                if (i != small)
                        std::swap(v[i], v[small]);
```

```
        }
}

void print(const std::vector<int>& a) {
        int n = a.size();
        std::cout << "{";
        if (n > 0) {
                std::cout << a[0];
                for (int i = 1; i < n; i++)
                        std::cout << ',' << a[i];
        }
        std::cout << "}";
}

void special_sort(std::vector<int>& v) {
        std::vector<int> odd, even;
        int n = v.size();
        for (int i = 0; i < n; i++)
                if (v[i] % 2 == 0)
                        even.push_back(v[i]);
                else
                        odd.push_back(v[i]);
        sort(even);
        sort(odd);
        int j = 0;
        for (int i = 0; i < even.size(); i++){
                v[i] = even[i];
                j++;
        }
        for (int i = 0; i < odd.size(); i++) {
                v[j + i] = odd[i];
        }
}

int main() {
        std::vector<int> test{ 2,1,10,4,3,6,7,9,8,5 };
        special_sort(test);
        print(test);
}
```

< 실행결과 >

{2,4,6,8,10,1,3,5,7,9}

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 12660개)이(가

) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

< 설명 >

이번에는 코드가 비효율적으로 길어졌지만, 처음에 머릿속에 떠오른 알고리즘대로 실행해보니 정상적으로 되어 그대로 코드를 옮겼다. sort라는 함수를 또 추가해서 이용했는데, 문제에 다른 사용자정의 함수를 사용하지 말라는 소리가 없었기에 넣었다. 만약, 있었어도 함수 안의 내용을 썼으면 된다. 마땅한 알고리즘이 생각나지않아 파라미터로 받은 벡터에서 홀수와 짝수를 분리하여 다른 벡터에 할당하였다. 이 각각의 벡터를 오름차순으로 sort하였고 이 sort된 벡터를 짝수부터 원래의 첫 요소부터 차례대로 할당해주면서 변경을 하였다. 본인도 효율적이라고 생각하지는 않는 코드지만 이 문제만 풀고 있기에 너무 오랜시간이 걸려 제출을 못할 것 같아 다음 문제를 풀고자 한다. 목적은 달성하였다.

3. Complete the following function that shifts all the elements of a vector backward one place. The last element that gets shifted off the back end of the vector is copied into the first (0th) position. For example, if a vector containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 is passed to the function, it would be transformed into 5, 2, 1, 10, 4, 3, 6, 7, 9, 8 Note that your function must physically rearrange the elements within the vector, not just print the elements in the shifted order.

```
void rotate(std::vector<int>& v) {
        // Add your code...
}
```

< 코드 >

```
void rotate(std::vector<int>& v) {
        int n = v.size();
        int save = v[n - 1];
        for (int i = n - 1; i > 0; i--)
                v[i] = v[i - 1];
        v[0] = save;
}
```

< 전체 코드 >

```
#include <iostream>
#include <vector>
#include <cstdlib>

void print(const std::vector<int>& a) {
        int n = a.size();
        std::cout << "{";
        if (n > 0) {
                std::cout << a[0];
                for (int i = 1; i < n; i++)
                        std::cout << ',' << a[i];
        }
        std::cout << "}";
}
```

```
void rotate(std::vector<int>& v) {
	int n = v.size();
	int save = v[n - 1];
	for (int i = n - 1; i > 0; i--)
		v[i] = v[i - 1];
	v[0] = save;
}

int main() {
	std::vector<int> test{ 2,1,10,4,3,6,7,9,8,5 };
	rotate(test);
	print(test);
}
```

< 실행결과 >

{5,2,1,10,4,3,6,7,9,8}

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 123160개)이(

가) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

< 설명 >

파라미터로 받은 벡터 사이즈를 int형 변수 n에 넣어두고 벡터의 맨 끝 요소의 값을 save 변수에 할당해준다. 그리고, for문을 통해서 벡터의 맨 뒤 요소부터 한칸씩 앞으로 가면서 자신의 앞 요소의 값을 할당받는다. 그리고 마지막에 벡터의 index 0번째 요소에 save했던 원본의 마지막 요소 값을 할당하면서 rotate 함수의 목적을 달성하였다.

4. Complete the following function that determines if the number of even and odd values in a vector is the same. The function would return true if the vector contains 5, 1, 0, 2 (two evens and two odds), but it would return false for the vector containing 5, 1, 0, 2, 11 (too many odds). The function should return true if the vector is empty, since an empty vector contains the same number of evens and odds (0). The function does not affect the contents of the vector.

```
bool balanced(const std::vector<int>& v) {
	// Add your code...
}
```

< 코드 >

```
bool balanced(const std::vector<int>& v) {
	std::vector<int> odd, even;
	int n = v.size();
```

```
        for (int i = 0; i < n; i++)
                if (v[i] % 2 == 0)
                        even.push_back(v[i]);
                else
                        odd.push_back(v[i]);
        if (even.size() == odd.size())
                return true;
        else
                return false;
}
```

< 전체 코드 >

```
#include <iostream>
#include <vector>
#include <cstdlib>

bool balanced(const std::vector<int>& v) {
        std::vector<int> odd, even;
        int n = v.size();
        for (int i = 0; i < n; i++)
                if (v[i] % 2 == 0)
                        even.push_back(v[i]);
                else
                        odd.push_back(v[i]);
        if (even.size() == odd.size())
                return true;
        else
                return false;
}

int main() {
        std::vector<int> test{ 5,1,0,2,11 };
        std::cout << balanced(test);
}
```

< 실행결과 >

0

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 122560개)이(

가) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

< 설명 >

2번에서 사용했던 것처럼 벡터를 파라미터로 받으면 벡터의 요소를 하나씩 뒤져보면서 짝수인 경우 새로 선언해준 even 벡터에 넣어주고, 홀수인 경우 odd 벡터에 넣어준다. 그 이후 두 벡터의

사이즈가 같으면 true를 return하고 그렇지 않다면 false를 return한다. 간단하다.

5. Complete the following function that returns true if vector a contains duplicate elements; it returns false if all the elements in a are unique. For example, the vector 2, 3, 2, 1, 9 contains duplicates (2 appears more than once), but the vector 2, 1, 0, 3, 8, 4 does not (none of the elements appear more than once). An empty vector has no duplicates. The function does not affect the contents of the vector.

```
bool has_duplicates(const std::vector<int>& v) {
        // Add your code...
}
```

< 코드 >

```
bool has_duplicates(const std::vector<int>& v) {
        int n = v.size();
        int m = 0;
        for (int i = 0; i < n - 1; i++) {
                for (int j = i + 1; j < n; j++) {
                        if (v[i] == v[j])
                                m = 1;
                }
        }
        return m;
}
```

< 전체 코드 >

```
#include <iostream>
#include <vector>
#include <cstdlib>

bool has_duplicates(const std::vector<int>& v) {
        int n = v.size();
        int m = 0;
        for (int i = 0; i < n - 1; i++) {
                for (int j = i + 1; j < n; j++) {
                        if (v[i] == v[j])
                                m = 1;
                }
        }
        return m;
}
int main() {
        std::vector<int> test{ 2, 1, 0, 3, 8, 4 };
        std::cout << has_duplicates(test);
}
```

< 실행결과 >

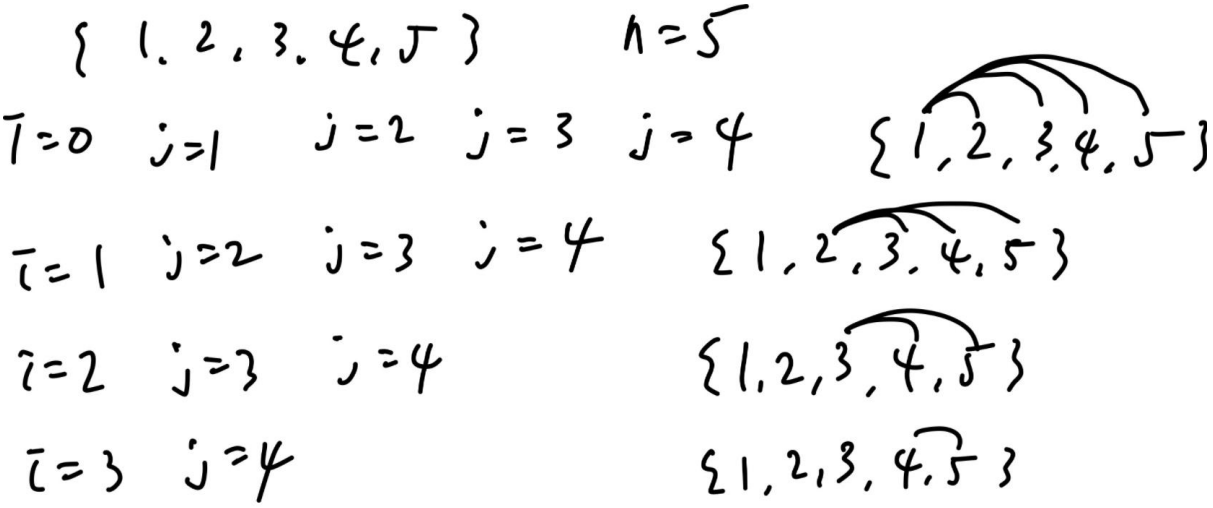
0

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 136176개)이(

가) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

< 설명 >



for문을 사용하여 벡터의 첫번째 요소부터 순서대로 뒤의 값이 동일한지 비교를 해본다. 자세한 절차는 그림을 참고하고, 만약 동일한 요소가 발견된다면 m이라는 변수에 1을 할당해줘서 return 할 때 true가 되도록 한다.

6. Can binary search be used on an unsorted vector? Why or why not?

: 사용할 수 없다. 왜냐하면 binary search 자체가 오름차순이건 내림차순이건 정렬이 되어있다는 가정하에 사용할 수 있는 것인데, 코드를 확인하면 알 수 있다. 오름차순의 경우 그 벡터의 사이즈의 반절에 있는 요소의 값이 찾는 값보다 크면 반절 기준 앞부분으로, 찾는 값보다 작으면 반절 기준 뒷부분으로 가는 것이다. 하지만 이것이 모두 오름차순이 되어있다고 가정되어야 상식적으로 가능한 일이다. 그래야 찾을 수 있기 때문이다. 정렬이 안되어있으면 내가 찾는 값이 가운데 요소의 값보다 작아도 뒷부분에 있는지 알 수가 없다.

7. Can linear search be used on an unsorted vector? Why or why not?

: 특정한 조건이 없는 한 사용 가능하다. Linear search의 경우 내가 찾는 값을 벡터의 맨 처음 요

소부터 찾을 때까지 계속 하나씩 뒤로 가면서 찾는 것이다. 그러다가 찾으면 index를 return 하는 것이고, 못 찾으면 -1을 return 한다. 어차피 정렬이 되어있건 안되어있건 앞에서부터 무작정 찾는 거기 때문에 정렬이 되어있을 필요가 없다는 것이다.

8. Complete the following function is_ascending that returns true if the elements in a vector of integers appear in ascending order (more precisely, non-descending order, if the vector contains duplicates). For example, the following statement std::cout << is_ascending({3, 6, 2, 1, 7}) << '₩n'; would print false, but the statement std::cout << is_ascending({3, 6, 7, 12, 27}) << '₩n'; would print true. The nonexistent elements in an empty vector are considered to be in ascending order because they cannot be out of order.

```
bool is_ascending(std::vector<int>& v) {
        // Add your code...
}
```

< 코드 >

```
bool is_ascending(std::vector<int>& v) {
        int n = v.size();
        int m = 1;
        for (int i = 1; i < n; i++)
                if (v[i-1] > v[i])
                        m = 0;
        return m;
}
```

< 전체 코드 >

```
#include <iostream>
#include <vector>
#include <cstdlib>

bool is_ascending(std::vector<int>& v) {
        int n = v.size();
        int m = 1;
        for (int i = 1; i < n; i++)
                if (v[i-1] > v[i])
                        m = 0;
        return m;
}


int main() {
        std::vector<int> test{3, 6, 7, 12, 27 };
        std::cout << is_ascending(test);
}
```

< 실행결과 >

1

C:₩Users₩Administrator₩source₩repos₩Ex002₩Debug₩Ex002.exe(프로세스 137356개)이(

가) 종료되었습니다(코드: 0개).

이 창을 닫으려면 아무 키나 누르세요...

< 설명 >

우선 빈 벡터의 경우에도 true(1)이 출력된다. 함수 내에서 m을 true(1)로 설정해두고 for문을 통해 벡터의 첫 요소부터 시작하여 그 바로 다음의 요소의 값과 비교하며 벡터의 끝까지 만약 앞의 요소가 더 큰 경우에 m을 false(0)으로 바꿔버린다. 그 이후 m을 return하면 오름차순이 아닌 경우 false, 맞으면 true가 출력된다.

9. Consider a sort function that uses the is_ascending function from the previous problem. It uses a loop to test the permutations of a vector of integers. When it finds a permutation that contains all of the vector's elements in ascending order it exits the loop. Do you think this would be a good sorting algorithm? Why or why not?

: is_ascending 함수는 벡터 안 요소들이 오름차순으로 정렬이 되어있으면 true 아니면 false를 return한다. Permutation loop의 경우 벡터 안 요소들의 순열 조합의 경우의 수를 모두 반환하는 코드이다. 예를 들어, 벡터 {1, 2, 3}이 있을 때, 총 6가지의 순열 조합이 나온다. 이 순열 조합을 모두 반환할 때, 여기서 오름차순으로 정렬된 경우의 순열은 딱 하나이다. (벡터의 요소의 개수에 상관없이 하나일 수 밖에 없다.) 바로 {1, 2, 3}이다. 그래서 is_ascending 함수를 이용해서 여러가지 순열 조합 중에서 오름차순 정렬이 된 순열을 도출해내는 것이 효율적이냐고 묻고 있지만 본인 생각에는 No이다. 왜냐하면, 주어진 벡터에 대해서 모든 순열의 조합을 알아낼 필요가 없이, 주어진 벡터를 오름차순으로 정렬하는 것이 훨씬 효율적이기 때문이다.