**REPORT**

**[Prac 2]**



**과 목 : 알고리즘06**

**담당교수 : 주종화 교수님**

**학 과 : 컴퓨터공학과**

**학 번 : 2021111971**

**이 름 : 이재혁**

텍스트, 클립아트이(가) 표시된 사진

자동 생성된 설명

**문제 1. 버블정렬**

**의사코드**

**텍스트, 친필, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 친필, 폰트, 잉크이(가) 표시된 사진

자동 생성된 설명**

**프로그램 구현 환경**

프로그래밍언어 : C++

텍스트 에디터 : Visual Studio Code

운영체제 : MAC OS

**프로그램 설계**

BubbleSort 클래스를 정의해 순환적, 비순환적 버블 정렬을 메소드로 정의해, 객체로 배열을 관리한다.

**멤버 변수**

**arr** : 입력받은 크기의 배열을 동적할당 받아 배열의 값을 저장하는 포인터

**len** : 배열의 크기를 저장하는 변수

**멤버 메소드**

**setArr()** : 배열을 설정하는 메소드, 배열의 크기와 그 개수 만큼의 값을 입력받는다.

**printArr()** : 현재 배열을 출력하는 메소드

**it\_BurrbleSort()** : 비순환적 버블 정렬,

index 0부터 끝까지 다음 원소와 값을 비교해 큰 값을 제일 오른쪽 index에 위치하게 한다.

한번 정렬을 시행하면 가장 큰 값이 배열의 가장 오른쪽에 위치한다.

2중 반복문을 사용해 정렬 범위를 1씩 줄여가며 남은 배열 중 가장 큰 값을 오른쪽으로 옮기는 작업을 반복한다.

**re\_BubbleSort(n)** : 순환적 버블 정렬,

이전 까지의 정렬 횟수를 매개변수로 갖는 메소드

index 0부터 끝까지 다음 원소와 값을 비교해 큰 값을 제일 오른쪽 index에 위치하게 한다.

n번의 정렬을 시행했다면, 마지막 n개의 원소는 정렬 되어있기 때문에, 배열의 길이 – n 개의 원소만 다시 정렬

* n이 배열의 길이와 같다면 정렬 된 것을 알 수 있음

1번의 정렬을 더 실행 했기 때문에 n+1의 값을 매개변수로 메소드를 재귀 호출

**소멸자** : 동적할당 받은 메모리를 반환

**소스코드**

// 2021111971 이재혁

// 문제 1번

#include <iostream>

using namespace std;

class BubbleSort {

public:

~BubbleSort() {

delete[] arr;

} // 객체가 소멸할 때 동적할당 했던 배열 메모리 반환

void setArr() { // 정렬 할 배열을 입력받을 메소드

cout << "배열의 길이 입력 : ";

cin >> len; // 길이를 입력받아 멤버 변수에 저장

arr = new int[len]; // 길이 만큼 멤버 포인터 변수에 배열할당

cout << "배열 입력 : ";

for (int i = 0; i < len; i++) {

cin >> arr[i];

} // 길이개수 만큼 배열 입력

}

void printArr() { // 배열을 출력하는 메소드

for (int i = 0; i < len; i++) {

cout << arr[i] << " ";

}

cout << endl;

}

void it\_BubbleSort() { // 비순환적 버블정렬

cout << "-------- 비순환적 버블정렬 --------\n원래 배열" << endl;

this->printArr();

// 0번째 index부터 마지막 1개전 index까지

for (int i = 0; i < len - 1; i++) { // 모든 원소에 대해서 정렬

int isSorted = 1; // 정렬여부를 확인할 변수

for (int j = 0; j < len - i - 1; j++) {

// 현재 원소부터 정렬횟수 -1 까지 비교

if (arr[j] >

arr[j + 1]) { // 현재원소가 다음원소보다 크다면 위치변경

int tmp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = tmp;

isSorted =

0; // 원소의 교환이 발생했으면 정렬이 되지 않았던 배열

}

}

if (isSorted) { // 원소의 교환이 없었다면 이미 정렬된 배열

cout << "정렬 완료" << endl;

this->printArr();

break;

}

// 정렬상황출력

cout << i + 1 << "번째 정렬" << endl;

this->printArr();

}

}

void re\_BubbleSort(int n) {

// 정렬횟수를 매개변수로 전달받아 (길이 - 정렬횟수 - 1) index까지

// 버블정렬 시행 가장 큰 원소를 오른쪽으로 이동시키기 때문에

// 정렬횟수만큼 뺀 길이를 다시 정렬

if (n == 0) { // 처음 정렬을 시작하면 원래 배열 출력

cout << "--------- 순환적 버블정렬 --------\n원래 배열" << endl;

this->printArr();

} else if (n == len) { // n-1번의 비교로 모든 원소를 비교하기 때문에

// n번째엔 종료

return;

}

int isSorted = 1; // 정렬여부 확인할 변수

for (int i = 0; i < len - n - 1; i++) {

if (arr[i] >

arr[i + 1]) { // 현재원소가 다음원소보다 크다면 원소교환

int tmp = arr[i];

arr[i] = arr[i + 1];

arr[i + 1] = tmp;

isSorted = 0; // 원소 교환이 일어났다면 정렬되지 않았던 배열

}

}

if (isSorted) { // 원소교환이 없어다면 정렬되었던 변수

cout << "정렬 완료" << endl;

this->printArr();

return;

} else {

// 정렬상황출력

cout << n + 1 << "번째 정렬" << endl;

this->printArr();

// 재귀적으로 다음 정렬 호출

re\_BubbleSort(n + 1);

}

}

private:

int \*arr; // 정렬할 배열의 위치를 알고있는 포인터

int len; // 정렬할 배열의 길이정보를 알고있는 포인터

};

int main() {

BubbleSort b;

b.setArr();

b.it\_BubbleSort();

b.setArr();

b.re\_BubbleSort(0);

}

// 배열의 길이 8

// 입력의 원소 30 20 40 10 5 10 30 15

// 입력의 원소 10 9 8 7 6 5 4 3

**결과 분석**

비순환적 버블 정렬

**텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

원래 배열에서 1번째 정렬과정

30 과 20 변경

30 과 40 변경 X

이후로 40이 계속 변경되어 배열의 가장 오른쪽에 위치

2번째 정렬은 다음으로 큰 30이 40 앞으로 밀린 것을 확인할 수 있다.

순환적 버블 정렬

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

비순환적 버블 정렬과 똑같이 정렬되는 것을 확인할 수 있다.

배열의 길이가 8이지만 5번째 정렬이후 정렬이 완료되는 것을 확인할 수 있다.

isSorted변수를 사용해 값 자리 교환이 일어났는지 확인하고 자리교환이 일어나지 않았다면 정렬된 배열이기 때문에 정렬을 종료한다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

내림차순의 배열이 들어오면 7번의 정렬이 시행되는 것을 알 수 있다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

isSorted변수를 처리 하지 않았을 경우 재귀함수가 초반에 정렬 여부를 확인하기 때문에, 시행횟수가 1회 더 많다.

**문제 2 : 퀵 정렬텍스트, 친필, 폰트, 잉크이(가) 표시된 사진

자동 생성된 설명**

**퀵 정렬의 의사코드**

**텍스트, 친필, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명 텍스트, 친필, 폰트, 서예이(가) 표시된 사진

자동 생성된 설명**

**프로그램 설계**

QuickSort 클래스를 정의해 난수 배열 설정 메소드, 순환적 퀵 정렬, 비순환적 퀵 정렬 메소드를 정의해 객체로 배열을 관리한다.

비순환적 퀵 정렬을 구현할 때 재귀호출을 사용하지 않는다.

* 퀵 정렬로 발생하는 분할 배열의 정보 저장필요
* 정렬해야 하는 index의 범위를 stack에 저장해 index 값을 꺼내 오면서 퀵 정렬을 시행한다.

난수 생성

100001크기의 int 배열을 전역변수로 정의

각 index의 값은 0또는 1로 난수로 생성됐으면 1, 아니면 0의 값을 가지고 있다.

큰 크기의 메모리를 소모하지만 현재 배열 상태를 확인하지 않아도 돼서 보다 빠른 시간안에 배열을 생성할 수 있다.

**멤버 변수**

**arr** : 길이가 10인 배열, 10개의 난수가 저장된다.

**stack** : 비순환적 퀵 정렬에서 사용하기 위한 스택

index를 2개씩 저장하고 4개의 index를 추가한 뒤 다음 정렬을 위해 2개를 빼낸다.

* 즉 크기가 최대 4인 스택

**top** : stack의 마지막 원소가 저장된 index 값을 추가할 때 마다 새로운 메모리를 할당하는 것이 아닌 top의 값을 수정

스택의 크기가 4를 넘지 않기 때문에 큰 메모리를 소모하지 않는다 따라서 top의 값을 수정해 효율적으로 stack을 관리할 수 있다.

-1의 값을 가지면 스택이 비어있음

**멤버 메소드**

**생성자** : 배열에 10개의 난수를 추가한다.

스택의 top을 -1 로 설정한다. (스택을 비운다)

**setArr()** : 배열에 새로운 난수 10개를 추가한다.

전역변수 배열을 0으로 초기화 하고 다시 10개의 수를

생성해 추가한다.

**printArr()** : 현재 배열 출력

**re\_QuickSort(int first, int last)** : 순환적 퀵 정렬 배열의

가장 왼쪽에 있는 값을 기준으로 정렬한다.

first, last는 정렬할 범위의 index이다.

first 다음부터 index를 늘려간다.

last부터 index를 줄여간다.

Index값이 교차하는 지점에 기준 값이 위치한다.

Index값을 수정하면서 기준 값의 위치를 수정했을 때 왼쪽에는 큰 값 오른쪽에는 작은 값이 위치하도록 교차점을 찾으면서 배열의 원소도 적절히 교환한다.

기준점을 기준으로 왼쪽 배열과 오른쪽 배열의 범위의 인덱스로 메소드를 재귀 호출한다

**It\_QuickSort()** : 비 순환적 퀵 정렬, 배열 수정 방식은 순환적

퀵 정렬과 동일하다.

스택에 다음 정렬해야 할 index를 계속 추가하며, stack이

빌 때 까지 while로 반복한다.

**s\_push(int n)** : stack에 값을 push 한다.

**s\_pop()** : 스택의 가장 큰 값을 빼낸다.

**s\_empty()** : 스택이 비어있는지 확인한다.

**소스코드**

// 2021111971 이재혁

// 문제 2번

#include <cstdlib>

#include <ctime>

#include <iostream>

using namespace std;

int num[100001] = {0};

// 난수의 발생여부를 저장할 배열

// index에 해당하는 난수가 발생했다면 1, 발생하지 않았다면 0의 값 저장

class QuickSort {

public:

QuickSort() { // 객체가 생성될 때, 난수배열을 생성하고, 스택의 마지막

// index를 -1로 설정한다

int i = 0;

while (i < 10) {

int n = rand() % 100000 + 1;

// 1 ~ 100000 사이의 난수 생성

if (num[n] == 0) { // 처음 발생한 난수라면

arr[i] = n; // 배열에 추가

num[n] = 1; // 생성된 난수로 표시

i++; // 다음에 추가할 위치로 이동

}

}

top = -1; // 스택의 최고 index를 -1로 설정 -> 스택이 비어있음

}

void setArr() { // 새로운 난수 배열을 생성하는 메소드

for (int i = 0; i < 10; i++) {

num[arr[i]] = 0;

}

// 기존 배열에 존재하는 값을 다시 생성되지 않은 값으로 설정

// arr[i]는 객체가 생성될 때 한번 생성되기 때문에,

// setArr() 메소드에서 다시 초기화 후 생성해야한다.

int i = 0;

while (i < 10) {

int n = rand() % 100000 + 1;

if (num[n] == 0) {

arr[i] = n;

num[n] = 1;

i++;

}

} // 난수 배열 다시 생성

}

void printArr() {

for (int i = 0; i < 10; i++) {

cout << arr[i] << "\t";

}

cout << endl;

} // 난수 배열 값 출력

// 순환적 퀵 정렬 : 정렬한 시작, 끝 index를 매개변수로 받아

// 정렬 후 새로운 범위로 재귀호출

void re\_QuickSort(int first, int last) {

// 전달받은 index들이 같거나 이미 교차된 상태라면

// 더 이상 분할 할 수 없는 상태

if (first >= last) {

return;

}

// 정렬 전 배열 출력

cout << "index : " << first << " ~ " << last << " 정렬전" << endl;

this->printArr();

// first를 index로 갖는 값을 기준으로 퀵정렬 시행

int left = first + 1; // 기준 값 보다 작은 값을 확인할 변수

int right = last; // 기준 값 보다 큰 값을 확인할 변수

while (left <= right) {

// left right가 만났다면 왼쪽에는 기준보다 큰 값

// 오른쪽에는 기준보다 작은 값이 위치해야함 -> 내림차순 정렬

// 등호를 포함하는 이유

// 2개의 값을 정렬할 때 left right가 같아짐

// left, right 값을 조정하지 않으면 수를 비교하지 않고

// 값을 교환하기 때문에 기준 값이 더 작아도 오른쪽에 위치할 수 있음

// 따라서 등호를 추가해 같은 값에서도 left, right값을 조절

while (left <= last && arr[left] > arr[first]) {

left++;

} // 주어진 범위를 넘어가지 않는 선에서 기준값 보다 작은 값을 찾는다

while (first < right && arr[right] < arr[first]) {

right--;

} // 주어진 범위를 넘어가지 않는 선에서 기준값 보다 큰 값을 찾는다

if (left < right) {

// left right가 만나지 않았다면 기준보다

// 작은 값이 교차점의 왼쪽, 큰 값은 교차점의 오른쪽에 위치한다

// 따라서 두 값을 교환한다 -> 내림차순 정렬

int tmp = arr[left];

arr[left] = arr[right];

arr[right] = tmp;

}

}

// left와 right가 교차되었으면 기준 값을 교차점의 값과 교환한다

int tmp = arr[first];

arr[first] = arr[right];

arr[right] = tmp;

// right 왼쪽에는 arr[right] 보다 큰 값

// 오른쪽에는 arr[right] 보다 작은 값이 위치하게 된다

cout << "정렬 후" << endl;

this->printArr();

re\_QuickSort(first, right - 1); // 기준 값 왼쪽 배열을 다시 퀵 정렬

re\_QuickSort(left, last); // 기준 값 오른쪽 배열을 다시 퀵 정렬

}

// 비순환적 퀵 정렬

// 정렬해야하는 범위의 index를 stack에 push하고

// stack에 index가 존재하면 값을 꺼내와 해당하는 범위 만큼 퀵 정렬 시행

void it\_QuickSort() {

this->s\_push(0);

this->s\_push(9);

// 0~9범위의 퀵정렬 부터 시행

while (!this->s\_empty()) {

// stack에 정렬해야하는 index범위가 존재하면 퀵 정렬 시행

int last = this->s\_pop();

int first = this->s\_pop();

// stack은 LIFO 형식이기 때문에 끝 index 먼저 저장

// 퀵 정렬 시행

if (first >= last) {

continue;

}

cout << "index : " << first << " ~ " << last << " 정렬전" << endl;

this->printArr();

int left = first + 1;

int right = last;

while (left <= right) {

while (left <= last && arr[left] > arr[first]) {

left++;

}

while (first < right && arr[right] < arr[first]) {

right--;

}

if (left < right) {

int tmp = arr[left];

arr[left] = arr[right];

arr[right] = tmp;

}

}

int tmp = arr[first];

arr[first] = arr[right];

arr[right] = tmp;

cout << "정렬 후" << endl;

this->printArr();

// 1개의 원소로 분할되지 않았으면 stack에 index 범위를 추가해 다음

// 정렬 시행

if (first < right - 1) {

this->s\_push(first);

this->s\_push(right - 1);

}

if (left < last) {

this->s\_push(left);

this->s\_push(last);

}

}

}

// 기본적인 stack 연산들

// stack의 가장 마지막 값을 저장하는 index인 top을 활영해 연산 수행

// stack 배열 내부에 값은 존재하지만, top index로 값을 덮어쓰고, 반환

// 값을 논리적으로 관리

void s\_push(int n) { stack[++top] = n; }

int s\_pop() { return stack[top--]; }

int s\_empty() {

if (top == -1) {

return 1;

} else {

return 0;

}

}

private:

int arr[10];

int stack[4];

// it\_QucikSort시 사용할 스택 가능한 좌표의 개수는 10개를 넘을 수 없다

int top;

// stack의 마지막 값이 저장된 index -1의 값을 가지면 stack에 값이 없고

// 0이면 가장 마지막 값이 0번 index에 존재한다는 의미

};

int main() {

srand((unsigned int)time(NULL));

QuickSort q;

cout << "---------- 순환적 퀵 정렬 ----------\n원래배열" << endl;

q.printArr();

q.re\_QuickSort(0, 9);

q.setArr();

cout << "---------- 비순환적 퀵 정렬 ----------\n원래배열" << endl;

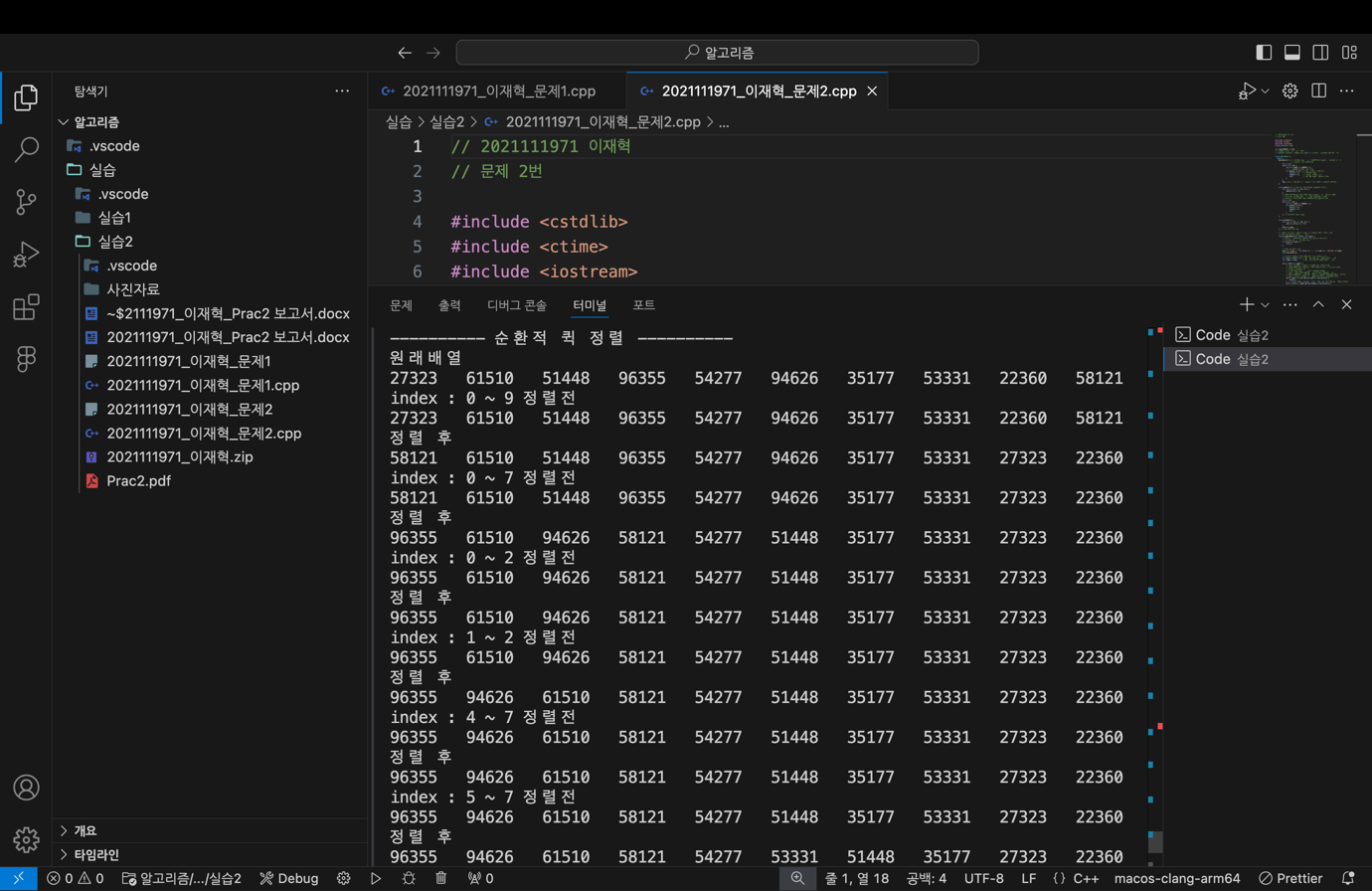
q.printArr();

q.it\_QuickSort();

}

**결과 분석**

순환적 퀵 정렬



0 ~ 9 정렬과정을 확인해보면 기준 값은 가장 왼쪽 값인 27323이다.

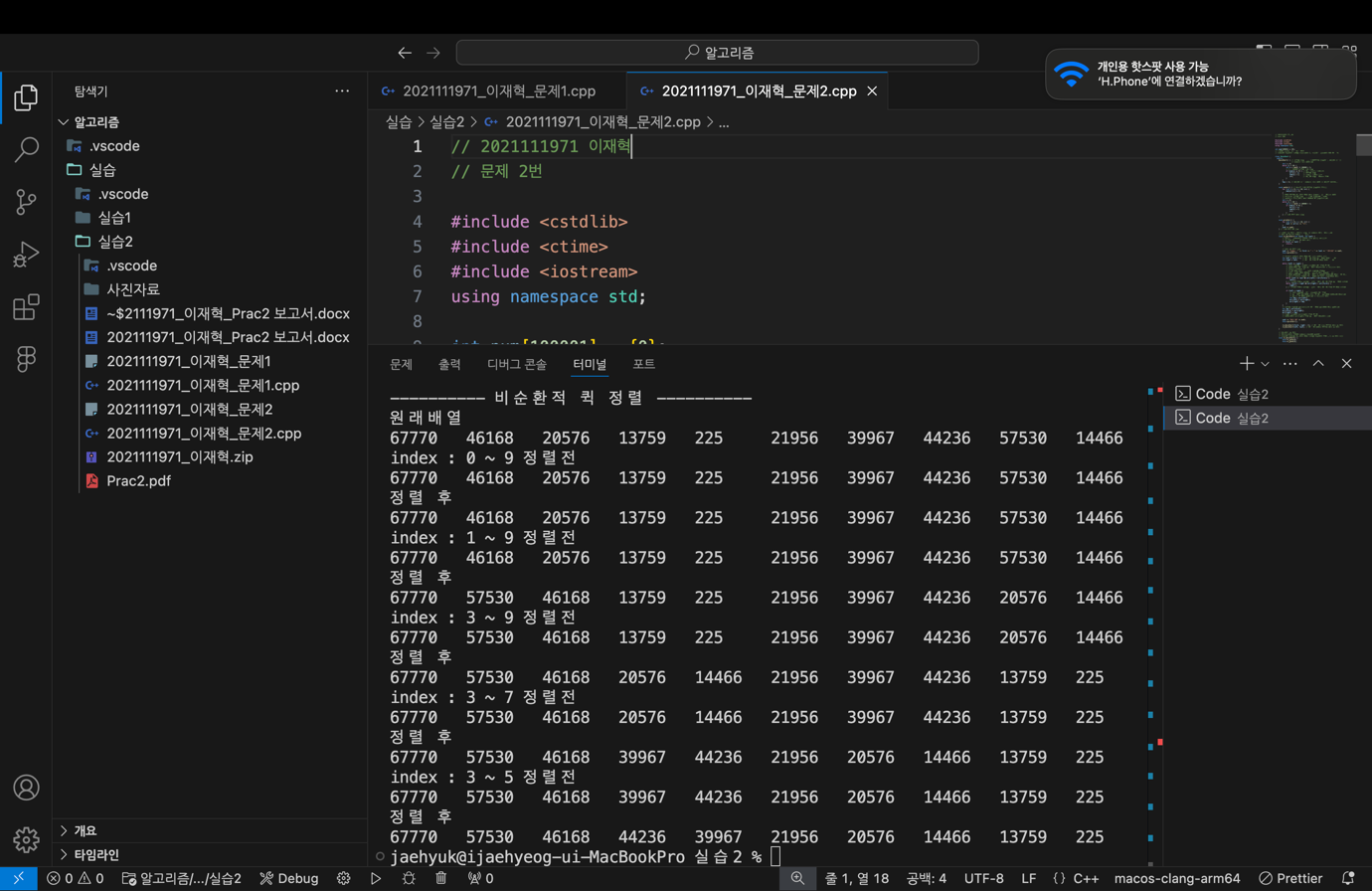
정렬 후에 27323의 위치는 index 8인 것을 확인할 수 있다.

27323의 오른쪽에는 작은 값인 22360

왼쪽에는 큰 값들이 위치한 것을 확인할 수 있다.

Index 8에 기준 값이 위치했기 때문에 index8을 제외한 범위 (0~7) 범위로 다시 퀵 정렬을 시행한다.

비순환적 퀵 정렬



새로운 난수 배열을 생성해 다시 정렬을 시행한다.

Stack은 Last In First Out 방식이기 때문에 나중에 push한

오른쪽 분할 배열(3~9)이 먼저 정렬되는 것을 확인할 수 있다.