**03. 정렬과 검색(Sort and Search)**

**03-1. 기초적인 정렬과 검색 알고리즘**

**# 정렬(Sort)**

* **순서 없이 배열되어 있는 자료들을 재배열 하는 것**

정렬의 대상 : 레코드

정렬의 기준 : 정렬 키(sort key)필드

**# 알고리즘 성능 비교**

* **점근적 분석 :** 항상 입력의 크기가 충분히 크다고 가정하고 수행시간을 분석한다.

****

* **경험적 분석 :** 알고리즘을 프로그래밍 언어로 구현 후에 실행 시간을 비교하는 것

**# 수 정렬하기—sort**

#include <iostream>

#include <vector> //vector

#include <algorithm> //sort

using namespace std;

int main(void) {

int N;

int temp;

vector<int> arr;

// 입력 : N개의 수

cin >> N;

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

cin >> temp;

arr.push\_back(temp);

}

cout << endl;

// std::sort : 퀵 정렬 기반으로 구현, 시간 복잡도 : O(nlogn)

sort(arr.begin(), arr.end());

//출력

for (int i = 0; i < N; i++ )

cout << arr[i] << endl;

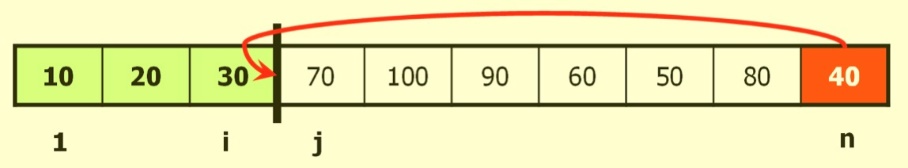
return 0;

}

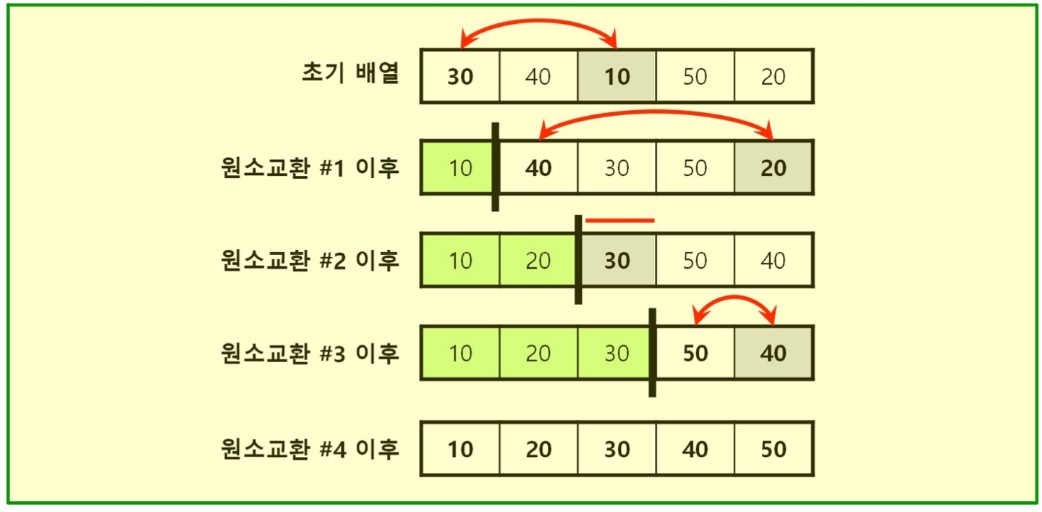
**03-2. 선택 정렬(Selection Sort)**

**# 배열 원소에 대한 선택 정렬 과정**

1. 먼저 정렬되지 않은 리스트에서 가장 작은 원소의 위치 탐색
2. 정렬되지 않은 리스트의 시작 위치에 있는 원소와 교환
3. 각각의 비교 및 교환 후에, 리스트의 경계를 한 개의 원소만큼 이동



**# 동작과정**

****

* 수행시간 : (n-1)+(n-2)+…+2+1=O(n^2) : Workst case, Average Case

**# 알고리즘**

// 선택 정렬: 오름차순

void selectionSort(int\* pArr, int num) {

int\* pSm;

for (int i = 0; i < num; i++) {**🡨 for 루프는 n-1번 반복**

pSm = pArr + i;

for (int j = i; j < num; j++)

// 가장 작은 값을 가진 원소의 위치(주소) 탐색

if (\*pSm > \*(pArr + j)) **🡨 가장 작은 수를 찾기 위한 비교 횟수 : 1,…,n-1**

pSm = pArr + j;

// 가장 작은 값을 선택: 현재 값과 교환

SWAP(pSm, pArr + i); **🡨 교환은 상수 시작 작업**

}

}

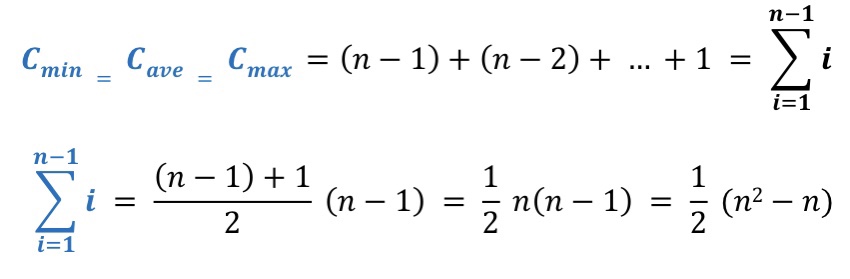
**# 알고리즘 분석**

* **메모리 사용공간 :** n개의 원소에 대하여 n개의 메모리 사용
* **원소 비교 횟수**

1. 첫번째 원소를 기준으로 n개의 원소 비교
2. 두번째 원소를 기준으로 마지막 원소까지 n-1개의 원소 비교
3. 세번째 원소를 기준으로 마지막 원소까지 n-2개의 원소 비교

(생략)

1. i번째 원소를 기준으로 n-i개의 원소 비교



* **어떤 경우에서나 원소 비교 횟수가 같기 때문에, 시간 복잡도 : O(n^2)**

**# 코드**

/\*

정렬: 알고리즘 구현

선택정렬

\*/

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void selectionSort(int\* pArr, int num); // 선택 정렬

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

/\*

//C 스타일 난수 생성(srand와 rand 함수)

srand(static\_cast<unsigned int>(time(nullptr)));

static\_cast<int>(rand() % (99 + 1UL - 0)) + 0;

\*/

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

selectionSort(arr, arrMAXSIZE);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

// 선택 정렬: 오름차순

void selectionSort(int\* pArr, int num) {

int\* pSm;

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

pSm = pArr + i;

for (int j = i; j < num; j++)

// 가장 작은 값을 가진 원소의 위치(주소) 탐색

if (\*pSm > \*(pArr + j))

pSm = pArr + j;

// 가장 작은 값을 선택: 현재 값과 교환

SWAP(pSm, pArr + i);

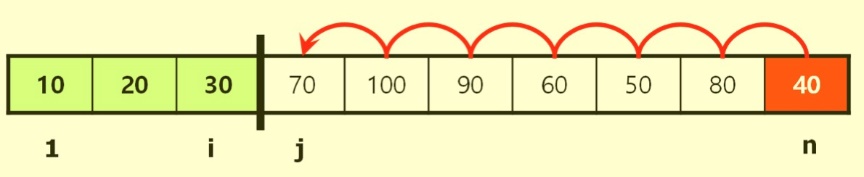
}

}

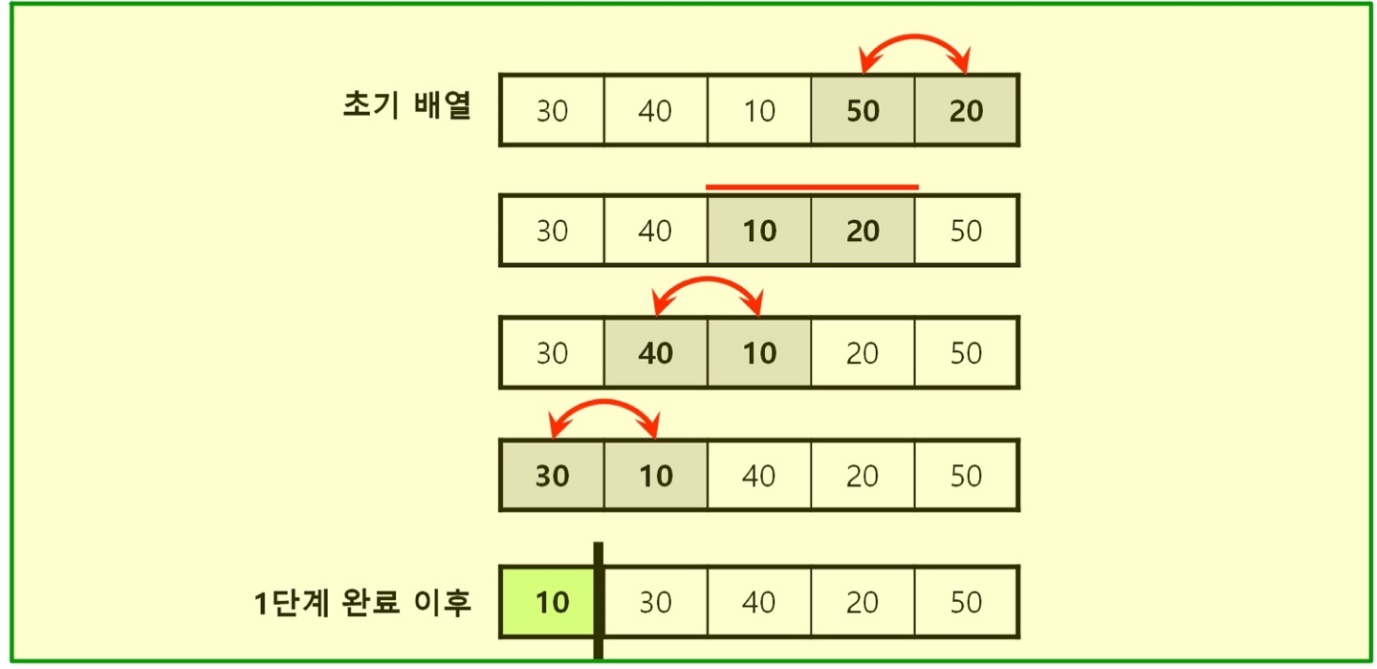
**03-3. 버블 정렬(Bubble Sort)**

**# 배열 원소에 대한 버블 정렬 과정**

1. 정렬되지 않은 리스트의 가장 작은 원소가 정렬된 서브 리스트로 이동한다.
2. 각각의 비교 및 교환 후에 리스트의 경계를 한 개의 원소만큼 이동한다.

****

**# 동작과정**

****

* **수행시간 : 1+2+…+(n-1)+n=O(n^2) : Worst case, Average case**

**# 알고리즘**

// 버블 정렬: 오름차순

void bubbleSort(int\* pArr, int num) {

for (int i = 0; i < num; i++) **{🡨 for 루프는 n-1번 반복**

for (int j = num - 1; j > 0; j--) **🡨 가장 큰수를 찾기 위한 비교 횟수 : n-1,…, 2, 1**

if (\*(pArr + j) < \*(pArr + j - 1)) **🡨 교환은 상수시간 작업**

SWAP(pArr + j, pArr + j - 1);

}

}

**# 변형된 알고리즘**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**# 알고리즘 분석**

* **메모리 사용 공간 :** n개의 원소에 대하여 n개의 메모리 사용
* **연산 시간**
  + **최선의 경우 :** 자료가 이미 정렬되어 있는경우

- 원소 비교 횟수 : i번째 원소를 (n-i)번 비교하기 때문에 n(n-1)/2

- 원소 교환 횟수 : 자리교환이 발생하지 않는다.

* + **최악의 경우 :** 자료가 역순으로 정렬되어 있는 경우

- 원소 비교 횟수 : i번째 원소를 (n-i)번 비교하기 때문에 n(n-1)/2

- 원소 교환 횟수 : i번째 원소를 (n-i)번 비교하기 때문에 n(n-1)/2

**🡪 평균 시간 복잡도 : O(n^2)**

**# 코드**

**1. 버블정렬 : 오름차순**

/\*

정렬: 알고리즘 구현

버블정렬

\*/

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void bubbleSort(int\* pArr, int num); // 버블 정렬

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

/\*

//C 스타일 난수 생성(srand와 rand 함수)

srand(static\_cast<unsigned int>(time(nullptr)));

static\_cast<int>(rand() % (99 + 1UL - 0)) + 0;

\*/

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

bubbleSort(arr, arrMAXSIZE);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

// 버블 정렬: 오름차순

void bubbleSort(int\* pArr, int num) {

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

for (int j = num - 1; j > 0; j--)

if (\*(pArr + j) < \*(pArr + j - 1))

SWAP(pArr + j, pArr + j - 1);

}

}

**2. 버블정렬 : 변형(향상)된 알고리즘**

void bubbleSortUp(int\* pArr, int num) {

for (int i = 0; i < num; i++)

{

bool state = true;

for (int j = num - 1; j > 0; j--)

{

if (\*(pArr + j) < \*(pArr + j - 1))

SWAP(pArr + j, pArr + j - 1);

state = false;

}

if (state = true)

return;

}

}

**03-4. 삽입 정렬(Insertion Sort)**

**: 배열이 거의 정렬되어 있는 상태일 때 가장 매력적인 알고리즘**

**# 배열 원소에 대한 삽입 정렬 과정**

1. 각 단계에서 정렬되지 않은 리스트의 첫번째 원소를 선택한다.
2. 선택된 원소를 정렬된 리스트의 적절한 위치로 삽입한다.

**텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명**

**# 동작과정**

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

자동 생성된 설명

* 수행시간 : O(n^2)
* Worst Case : 1+2+…+(n-2)+(n-1)=O(n^2)
* Average Case : 1/2(1+2+…+(n-2)+(n-1))=O(n^2)

**# 알고리즘**

// 삽입 정렬: 오름차순

void insertionSort(int\* pArr, int num) { **🡨 pArr[1,…, num]을 정렬**

int i, j, temp;

for (i = 1; i < num; i++) { **🡨 for루프는 num-1번 반복**

temp = \*(pArr + i);

// temp 가 들어갈 공간 확보

for (j = i - 1; j >= 0 && \*(pArr + j) > temp; j--)

\*(pArr + j + 1) = \*(pArr + j);

\*(pArr + j + 1) = temp;

**🡨 삽입은 최악의 경우 i-1회 비교**

}

}

**# 알고리즘 분석**

* **메모리 사용 공간 :** n개의 원소에 대하여 n개의 메모리 사용
* **연산 시간**

1. **최선의 경우 : 원소들이 이미 정렬 되어 있을 때 원소 비교횟수가 최소**
   * 이미 정렬되어 있는 경우에는 바로 앞자리 원소와 한번만 비교
   * 전체 원소 비교횟수 = n-1
   * 시간 복잡도 : O(n)
2. **최악의 경우 : 모든 원소가 역순으로 되어 있을 경우 원소 비교 횟수가 최대**
   * 전체 원소 비교 횟수 = 1+2+3+…+(n-1) = n(n-1)/2
   * 시간 복잡도 : O(n^2)
3. **삽입 정렬의 평균 원소 비교 횟수 = n(n-1)/4**

* **평균 시간 복잡도 : O(n^2)**

**# 코드**

/\*

정렬: 알고리즘 구현

삽입정렬

\*/

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void insertionSort(int\* pArr, int num); // 삽입 정렬

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

/\*

//C 스타일 난수 생성(srand와 rand 함수)

srand(static\_cast<unsigned int>(time(nullptr)));

static\_cast<int>(rand() % (99 + 1UL - 0)) + 0;

\*/

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

insertionSort(arr, arrMAXSIZE);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

// 삽입 정렬: 오름차순

void insertionSort(int\* pArr, int num) {

int i, j, temp;

for (i = 2; i < num; i++) {

temp = \*(pArr + i);

j = i - 1;

// temp 가 들어갈 공간 확보

while (j >= 0 && \*(pArr + j) > temp) {

\*(pArr + j + 1) = \*(pArr + j);

j--;

}

\*(pArr + j + 1) = temp;

}

}

**03-5. 순차검색(Sequential Search)**

**# 검색(Search)**

**: 레코드의 집합에서 주어진 키를 지닌 레코드를 찾는 작업 탐색**

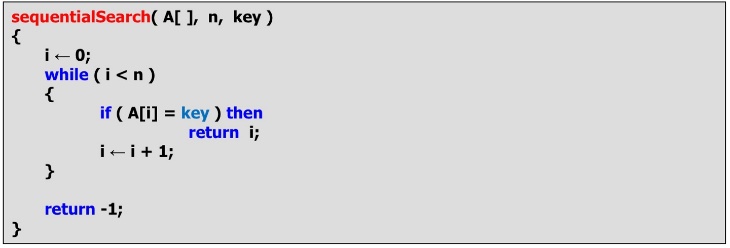
* 주어진 키 값 : 목표 키(target key) 또는 검색 키(search key)

**선형 검색(Linear Search)**

**# 알고리즘**

: 목표치를 찾기 위해 리스트의 처음부터 탐색을 시작해서, 목표치를 찾거나 리스트에 목표치가 없다는 것이 밝혀질 때까지 검색을 계속한다.

* 순차 검색은 순서가 없는 리스트일 때 사용
* 순차 검색은 리스트가 작거나, 가끔 한번씩 검색할 경우에만 사용



**# 동작과정**

**검색성공(목표 데이터 : 73)**

****

**검색실패(목표 데이터 : 90)**

****

**03-6. 이진검색(Binary Search)**

: 이진 검색은 배열이 정렬되어 있을 때 효율적인 알고리즘(순차 검색은 매우 느리다.)

**# 알고리즘**

* **검색할 데이터들은 정렬된 상태이다.**
* **주어진 데이터들은 유일한 키 값을 가지고 있다.**

// 이진탐색: 재귀적용법

int binarySearch(int\* pArr, int first, int last, int key) {

if (first > last) // 재귀함수의 탈출 조건

return -1;

// 검색 범위의 중간 원소의 위치 값 계산

int index, mid = (first + last) / 2;

if (\*(pArr + mid) == key) return mid;

else if (\*(pArr + mid) > key) index = binarySearch(pArr, first, mid - 1, key);

else if (\*(pArr + mid) < key) index = binarySearch(pArr, mid + 1, last, key);

return index;

}

**# 동작과정**

**1. 검색성공(검색 데이터 : 21)**

텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명

**2. 검색실패(검색 데이터 : 11)**

**텍스트, 스크린샷, 평행, 라인이(가) 표시된 사진

자동 생성된 설명**

**# 코드**

**1. 이진검색 : 재귀적 용법**

#include <iostream>

#define arrMAXSIZE 10

using namespace std;

// 이진 탐색 -- 알고리즘 구현 / 재귀적 용법

int binarySearch(int\* pArr, int first, int last, int key);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 5, 9, 13, 17, 21, 28, 37, 46, 55, 88 };

cout << "원시 데이터: ";

PRINT(arr, arrMAXSIZE);

int key;

while (true) {

cout << "검색 데이터 입력(검색 종료: 0) : ";

cin >> key;

if (key == 0)

break;

int index = binarySearch(arr, 0, arrMAXSIZE - 1, key);

if (index == EOF) cout << "없다고!!!" << endl;

else cout << "검색 데이터: " << arr[index] << "번째 위치 " << index + 1 << endl;

}

return 0;

}

// 이진탐색: 재귀적용법

int binarySearch(int\* pArr, int first, int last, int key) {

// 재귀함수의 탈출 조건

if (first > last)

return EOF;

// 중간 원소의 위치(첨자) 계산: first ~ last 범위에서...

int index, mid = (first + last) / 2;

if (\*(pArr + mid) == key) return mid;

else if (\*(pArr + mid) < key) index = binarySearch(pArr, mid + 1, last, key);

else if (\*(pArr + mid) > key) index = binarySearch(pArr, first, mid - 1, key);

return index;

}

**2. 이진검색 : 비재귀적 용법**

#include <iostream>

#define arrMAXSIZE 10

using namespace std;

// 이진 탐색 -- 알고리즘 구현 / 비재귀적 용법

int\* binarySearch(int\* pArr, int\* pFirst, int\* pLast, int key);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 5, 9, 13, 17, 21, 28, 37, 46, 55, 88 };

cout << "원시 데이터: ";

PRINT(arr, arrMAXSIZE);

int key;

while (true) {

cout << "검색 데이터 입력(검색 종료: 0) : ";

cin >> key;

if (key == 0)

break;

int\* pIndex = binarySearch(arr, arr, arr + arrMAXSIZE - 1, key);

if (pIndex == NULL) cout << "없다고!!!" << endl;

else cout << "검색 데이터: " << \*pIndex << "번째 위치 " << pIndex - arr + 1 << endl;

}

return 0;

}

// 이진탐색: 비재귀적용법

int\* binarySearch(int\* pArr, int\* pFirst, int\* pLast, int key) {

while (pFirst <= pLast) {

// 중간 원소의 위치(주소) 계산: pFirst ~ pLast 범위에서...

int\* pMid = pFirst + (pLast - pFirst) / 2;

if (\*pMid == key) return pMid;

else if (\*pMid < key) pFirst = pMid + 1;

else if (\*pMid > key) pLast = pMid - 1;

}

return NULL;

}

**03-7. 쉘 정렬(Shell Sort)**

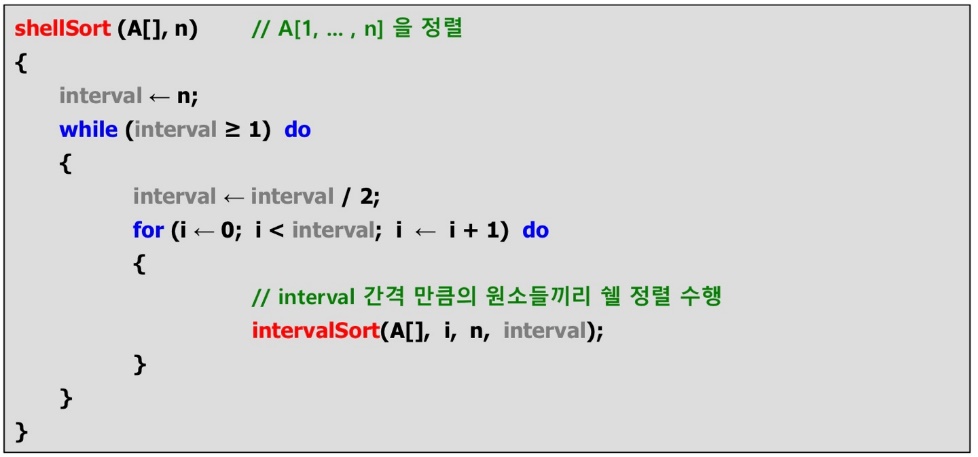
: 일정한 간격으로 데이터들끼리 부분집합을 구성하고, 각 부분집합에 있는 원소들에 대해서 **삽입 정렬**을 수행한다.

* 전체 원소에 대해서 삽입 정렬을 수행하는 것보다 부분집합으로 나누어 정렬하면 **비교와 교환 연산을 감소**시킬 수 있다.
* 쉘 정렬에서는 7-정렬, 4-정렬 등의 용어를 주로 사용함.

**ex) 4- 정렬의 예**

****

**# 알고리즘**

****

**# 알고리즘 분석**

* **메모리 사용 공간 :** n개의 원소에 대하여 n개의 메모리와 매개변수 h에 대한 저장공간 사용
* **연산시간**

1. 원소 비교 횟수 : 처음 원소의 상태에 상관없이 매개변수 h에 의해 결정
2. 일반적인 시간 복잡도 : O(n^1.25)
3. 쉘 정렬은 삽입 정렬의 시간 복잡도O(n^2)보다 개선된 정렬 방법

* **삽입 정렬 + interval = 쉘 정렬**

**# 코드**

/\*

정렬: 알고리즘 구현

쉘정렬

\*/

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void shellSort(int\* pArr, int num); // 쉘 정렬

void intervalSort(int\* pArr, int num, int start, int interval);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

/\*

//C 스타일 난수 생성(srand와 rand 함수)

srand(static\_cast<unsigned int>(time(nullptr)));

static\_cast<int>(rand() % (99 + 1UL - 0)) + 0;

\*/

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

shellSort(arr, arrMAXSIZE);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

// 쉘 정렬: 오름차순

void shellSort(int\* pArr, int num) {

int interval = num;

while (interval >= 1) {

interval /= 2;

for (int i = 0; i < interval; i++)

intervalSort(pArr, num, i, interval);

printf("interval: %d, ", interval);

}

}

// 쉘 정렬에 쓰이는 삽입 정렬: interval 간격만큼 삽입 정렬

void intervalSort(int\* pArr, int num, int start, int interval) {

int i, j, temp;

for (i = start; i < num; i += interval) {

temp = \*(pArr + i);

for (j = i - interval; j >= 0 && \*(pArr + j) > temp; j -= interval)

\*(pArr + j + interval) = \*(pArr + j);

\*(pArr + j + interval) = temp;

}

}

**03-8. 퀵 정렬(Quick Sort)**

* **정렬할 전체 원소에 대해서 정렬을 수행하지 않고, 기준 값을 중심으로 왼쪽 부분 집합과 오른쪽 부분 집합으로 분할하여 정렬**

**기준 값 : Pivot**

왼쪽 부분 집합 : 기준 값보다 작은 원소들을 이동

오른쪽 부분 집합 : 기준 값보다 큰 원소들을 이동

* **퀵 정렬은 다음의 두 가지 기본작업을 반복 수행**
  + **분할(Divide)**

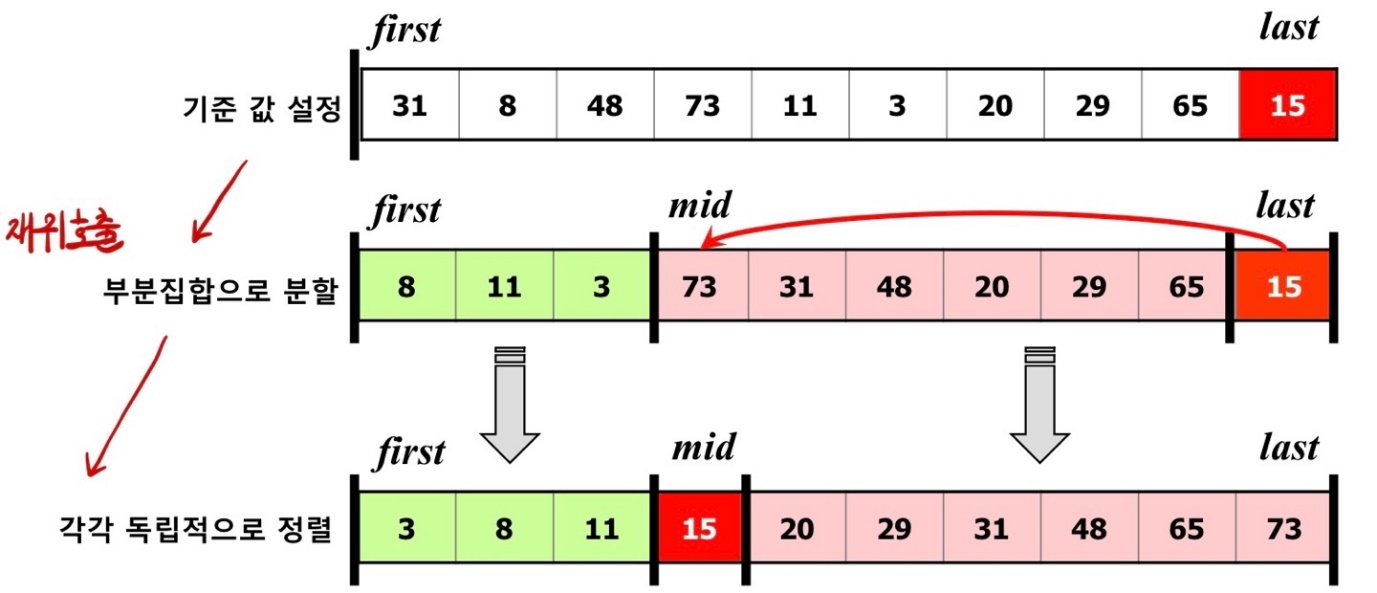
**-** 정렬할 자료들을 기준 값을 중심으로 두 개의 부분 집합으로 분할

* + **정복(Conquer)**

**-** 부분 집합의 원소들 중에서 기준 값보다 작은 원소들은 왼쪽 부분 집합으로, 기준 값보다 큰 원소들은 오른쪽 부분집합으로 정렬

**- 부분 집합의 크기가 1이하로 충분히 작지 않으면, 순환호출을 이용하여 다시 분할**

**# 동작과정**

****

* **평균 수행시간 : O(nlong)**
* **최악의 경우 수행시간 : O(n^2)**

**# 알고리즘**

//퀵 정렬 : 오름차순

void quickSort(int\* pArr, int\* pfirst, int\* plast) {

int\* pmid = NULL;

if (pfirst - pArr < plast - pArr) {

pmid = Partition(pArr, pfirst, plast);//분할 후 기준 값의 위치 값을 반환

quickSort(pArr, pfirst, pmid - 1);//왼쪽 부분 정렬

quickSort(pArr, pmid + 1, plast);//오른쪽 부분 정렬

}

}

int\* Partition(int\* pArr, int\* pfirst, int\* plast) {

int pivot = \*plast; //마지막 원소를 기준 값으로 선택

int i = (pfirst - pArr) - 1;

for (int j = (pfirst - pArr); j < (plast - pArr); j++) {

if (pArr[j] <= pivot)

SWAP(pArr + (++i), pArr + j);//기준 값을 가운데로 위치 시킨다.

}

SWAP(pArr + i + 1, plast);

return pArr + i + 1;//기준 값의 위치 값을 반환

}

**# 알고리즘 분석**

* **메모리 사용 공간 :** n개의 원소에 대하여 n개의 메모리 사용
* **연산시간**

1. **최선의 경우 : O(nlogn)**

기준 값에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 n/2개씩 이등분이 되는 경우가 반복되어 수행 단계 수가 최소가 되는 경우

1. **최악의 경우 : O(n^2)**

기준 값에 의헤 원소들을 분할하였을 때 1개와 n-1개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대가 되는 경우

1. **평균 시간 복잡도 : O(nlogn)**

같은 시간 복잡도는 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법

**# 코드**

/\*

정렬: 알고리즘 구현

퀵정렬

\*/

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void quickSort(int\* pArr, int\* pFirst, int\* pLast);// 퀵 정렬

int\* Partition(int\* pArr, int\* pfirst, int\* plast);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

/\*

//C 스타일 난수 생성(srand와 rand 함수)

srand(static\_cast<unsigned int>(time(nullptr)));

static\_cast<int>(rand() % (99 + 1UL - 0)) + 0;

\*/

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

quickSort(arr, arr, arr + arrMAXSIZE - 1);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

//퀵 정렬 : 오름차순

void quickSort(int\* pArr, int\* pfirst, int\* plast) {

int\* pmid = NULL;

if (pfirst - pArr < plast - pArr) {

pmid = Partition(pArr, pfirst, plast);//분할 후 기준 값의 위치 값을 반환

quickSort(pArr, pfirst, pmid - 1);//왼쪽 부분 정렬

quickSort(pArr, pmid + 1, plast);//오른쪽 부분 정렬

}

}

int\* Partition(int\* pArr, int\* pfirst, int\* plast) {

int pivot = \*plast; //마지막 원소를 기준 값으로 선택

int i = (pfirst - pArr) - 1;

for (int j = (pfirst - pArr); j < (plast - pArr); j++) {

if (pArr[j] <= pivot)

SWAP(pArr + (++i), pArr + j);//기준 값을 가운데로 위치 시킨다.

}

SWAP(pArr + i + 1, plast);

return pArr + i + 1;//기준 값의 위치 값을 반환

}

**03-8. 병합 정렬(Merge Sort)**

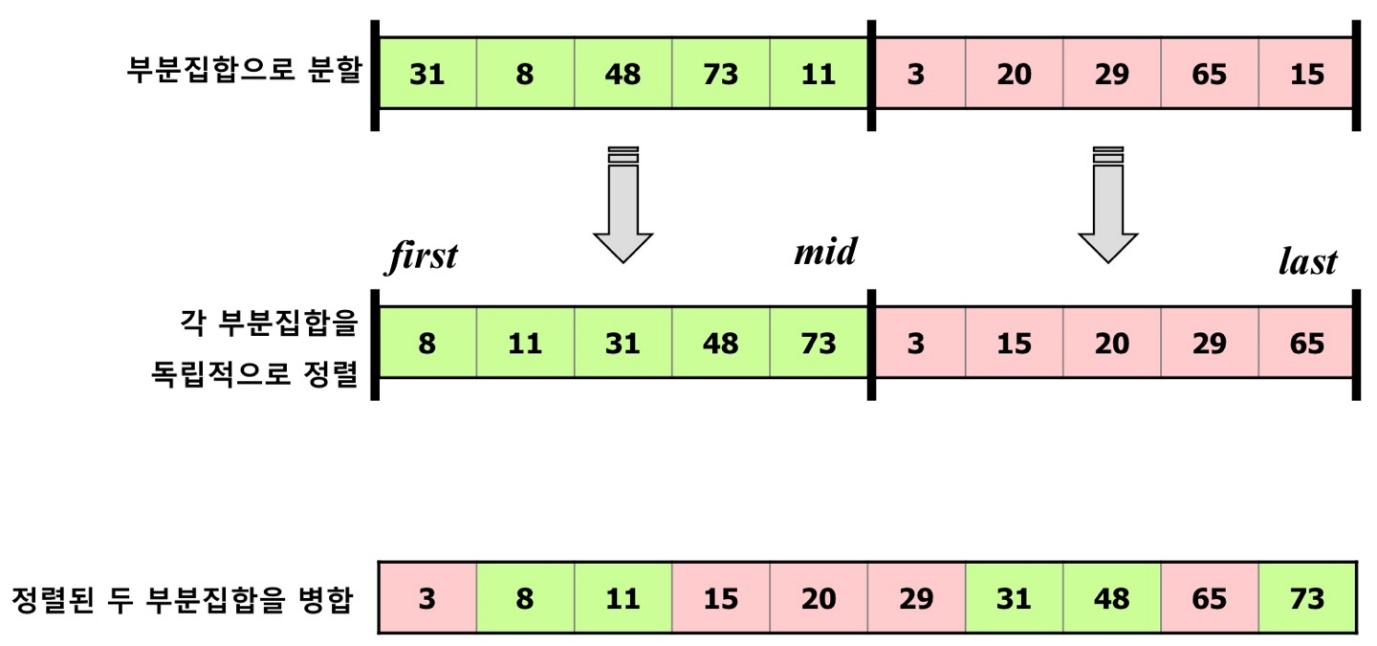
**: 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법**

* **병합 정렬 방법의 종류**
  + **2-way 병합 :** 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 방법
  + **n-way 병합 :** n개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 방법
* **2-way 병합 정렬 : 세 가지 기본 작업을 반복 수행**
  + **분할(Divide) :** 입력 자료를 같은 크기의 부분집합 2개로 분할한다.
  + **정복(Conquer) :** 부분집합의 원소들을 정렬한다.

**\* 만약 부분집합의 크기가 충분히 작지 않으면, 순환호출을 이용하여 다시 분할 정복 기법을 적용한다.**

* + **결합(Combine) :** 정렬된 부분집합을 하나의 집합으로 통합한다.

**# 동작과정**

****

**# 알고리즘**

//병합 정렬 : 오름차순

void mergeSort(int\* pArr, int\* pFirst, int\* pLast) {

int\* mid = NULL;

if ((pFirst - pArr) < (pLast - pArr)) {

mid = pFirst + (pLast - pFirst) / 2;//first와 last 사이의 중간 원소의 위치

mergeSort(pArr, pFirst, mid); //왼쪽 부분집합 정렬

mergeSort(pArr, mid + 1, pLast); //오른쪽 부분집합 정렬

merge(pArr, pFirst, mid, pLast);//정렬된 두 부분집합 병합

}

}

void merge(int\* pArr, int\* pFirst, int\* pMid, int\* pLast) {

int i = pFirst - pArr;

int j = (pMid - pArr) + 1;

int t = i;

int\* tempArr = new int[arrMAXSIZE];

while (i <= (pMid - pArr) && j <= (pLast - pArr)) {

if (pArr[i] <= pArr[j]) {

tempArr[t++] = pArr[i++];

}

else {

tempArr[t++] = pArr[j++];

}

}

while (i <= (pMid - pArr)) tempArr[t++] = pArr[i++];

while (j <= (pLast - pArr)) tempArr[t++] = pArr[j++];

//정렬된 상태로 재구성된 temp 배열을 원본 배열 A에 복사

i = pFirst - pArr;

t = i;

while (i <= (pLast - pArr)) pArr[i++] = tempArr[t++];

delete[] tempArr;

}

**# 알고리즘 분석**

* **메모리 사용 공간**
  + 각 단계에서 새로 병합하여 만든 부분집합을 저장할 공간이 추가로 필요
  + 원소 n개에 대해서 (2\*n)개의 메모리 공간 사용
* **연산 시간**
  + 분할 단계 : n개의 원소를 분할하기 위해서 log2n번의 단계 수행
  + 병합 단계 : 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 n번의 비교연산 수행
* **전체 병합정렬의 시간 복잡도 : O(nlog2n)**

**# 코드**

/\*

정렬: 알고리즘 구현

병합정렬

\*/

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void mergeSort(int\* pArr, int\* pFirst, int\* pLast);// 병합 정렬

void merge(int\* pArr, int\* pFirst, int\* pMid, int\* pLast);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

mergeSort(arr, arr, arr + arrMAXSIZE - 1);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

//병합 정렬 : 오름차순

void mergeSort(int\* pArr, int\* pFirst, int\* pLast) {

int\* mid = NULL;

if ((pFirst - pArr) < (pLast - pArr)) {

mid = pFirst + (pLast - pFirst) / 2;//first와 last 사이의 중간 원소의 위치

mergeSort(pArr, pFirst, mid); //왼쪽 부분집합 정렬

mergeSort(pArr, mid + 1, pLast); //오른쪽 부분집합 정렬

merge(pArr, pFirst, mid, pLast);//정렬된 두 부분집합 병합

}

}

void merge(int\* pArr, int\* pFirst, int\* pMid, int\* pLast) {

int i = pFirst - pArr;

int j = (pMid - pArr) + 1;

int t = i;

int\* tempArr = new int[arrMAXSIZE];

while (i <= (pMid - pArr) && j <= (pLast - pArr)) {

if (pArr[i] <= pArr[j]) {

tempArr[t++] = pArr[i++];

}

else {

tempArr[t++] = pArr[j++];

}

}

while (i <= (pMid - pArr)) tempArr[t++] = pArr[i++];

while (j <= (pLast - pArr)) tempArr[t++] = pArr[j++];

//정렬된 상태로 재구성된 temp 배열을 원본 배열 A에 복사

i = pFirst - pArr;

t = i;

while (i <= (pLast - pArr)) pArr[i++] = tempArr[t++];

delete[] tempArr;

}

**03-9. 특수 정렬 알고리즘**

* **두 원소를 비교하는 정렬의 하한선은 Ω(nlogn)**
* **최악의 경우 정렬 시간이 O(nlong)보다 더 빠를 수는 없는가?**
* **원소들이 특수한 성질을 만족하면 O(n) 정렬도 가능하다.**

# 계수 정렬(Counting Sort) : 원소들의 **크기가 모두 -O(n)~O(n) 범위**에 있을 때

# 기수 정렬(Radix Sort) : 원소들이 **모두 k 이하의 자리수**를 가졌을 때(k: 상수)

# 버킷 정렬(Bucket Sort) : 원소들이 **모두 균등 분포**를 이룰 때

**03-10. 계수 정렬(Counting Sort)**

: 항목들의 순서를 결정하기 위해 집합에 각 항목이 몇 개씩 있는지 세는 작업을 하면서 선형 시간에 정렬하는 효율적인 알고리즘

* 속도가 빠르며 안정적이다.
* 제한 사항

1. 정수나 정수로 표현할 수 있는 자료에 대해서만 동작
2. 카운트들을 위한 충분한 공간을 할당하려면 집합 내의 가장 큰 정수를 알아야 한다.

**# 알고리즘**

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

**# 동작과정**

**텍스트, 스크린샷, 번호, 라인이(가) 표시된 사진

자동 생성된 설명**

**# 코드**

/\*

정렬: 알고리즘 구현

계수정렬

\*/

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void contingSort(int\* pArr, int num); // 계수정렬

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

/\*

//C 스타일 난수 생성(srand와 rand 함수)

srand(static\_cast<unsigned int>(time(nullptr)));

static\_cast<int>(rand() % (99 + 1UL - 0)) + 0;

\*/

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

contingSort(arr, arrMAXSIZE);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

//계수 정렬

void contingSort(int\* pArr, int num) {

int\* countArr, \* sortArr;

int maxNum = 0;

for (int i = 0; i < num; i++)

if (maxNum < pArr[i])

maxNum = pArr[i];

// countArr : 원본 데이터에서 각 항목들의 발생 횟수를 저장한다.

// 단, 원본 데이터의 값 자체가 countArr의 첨자가 된다.

countArr = new (nothrow) int[maxNum + 1] { 0 };

if (countArr == nullptr) {

cout << "동적 메모리 공간 할당 실패!!!" << endl;

return;

}

for (int i = 0; i < num; i++) countArr[pArr[i]]++;

for (int i = 1; i <= maxNum; i++) countArr[i] += countArr[i - 1];

// sortArr : countArr의 값을 첨자로 하여 원본 데이터를 정렬된 데이터로 구성한다.

sortArr = new (nothrow) int[num] {0};

if (countArr == nullptr) {

cout << "동적 메모리 공간 할당 실패!!!" << endl;

exit(100);

}

for (int i = num - 1; i >= 0; i--) {

sortArr[countArr[pArr[i]] - 1] = pArr[i];

countArr[pArr[i]]--;

// PRINT(sort, num);

}

// 정렬된 데이터(sortArr)로 원본 데이터(pArr)를 재구성한다.

for (int i = 0; i < num; i++)

pArr[i] = sortArr[i];

delete[] countArr;

delete[] sortArr;

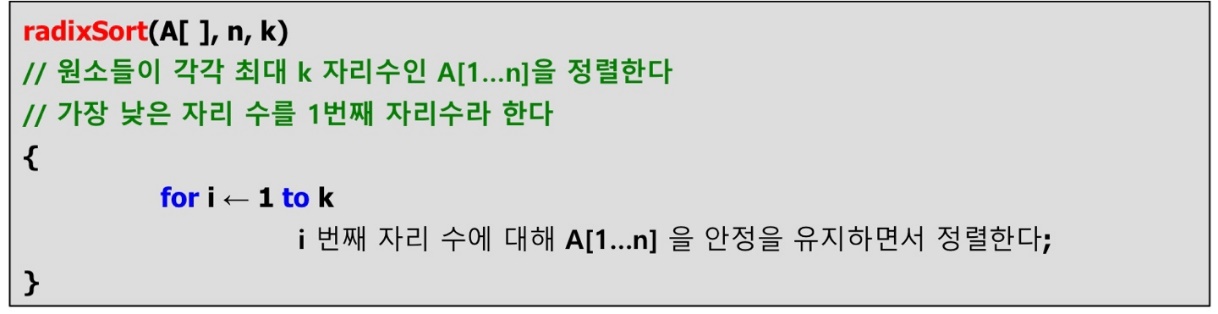
}

**03-11. 기수 정렬(Radix Sort)**

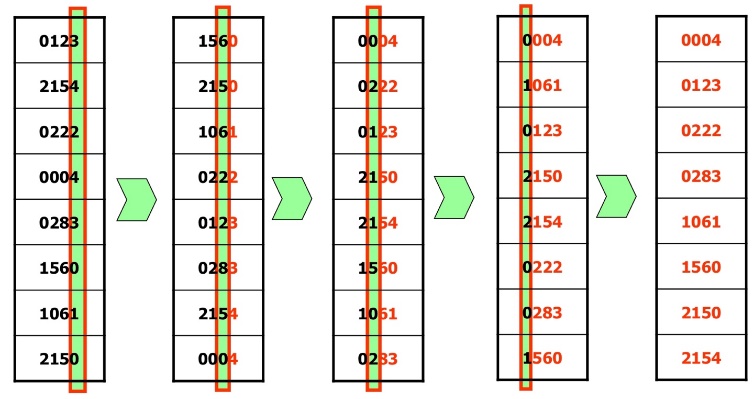
: 입력들이 모두 k이하의 자리수를 가진 특수한 경우에(자연수가 아닌 제한된 종류를 가진 알파벳 등도 해당) 사용할 수 있는 방법

* O(n) 시간이 소요되는 알고리즘
* 안정성 정렬(Stable Sort) : 같은 값을 가진 원소들은 정렬 후에도 원래의 순서가 유지되는 성질을 가진 정렬을 일컫는다.

**# 알고리즘**

****

**# 동작과정**

****

**# 알고리즘 분석**

* **메모리 사용 공간 :** 원소 n개에 대해서 n개의 메모리 공간 사용(기수 r에 따라서 버킷 공간이 추가로 필요)
* **연산 시간**

1. 연산시간은 정렬할 원소의 수(n)와 키 값의 자릿수(d)와 버킷의 수를 결정하는 기수(r)에 따라서 달라진다.
2. **수행할 전체 작업 :** d(n+r)

🡨 정렬할 원소 n개를 r개의 버킷에 분배하는 작업 : (n+r)

🡨 이 작업을 자릿수 d만큼 반복

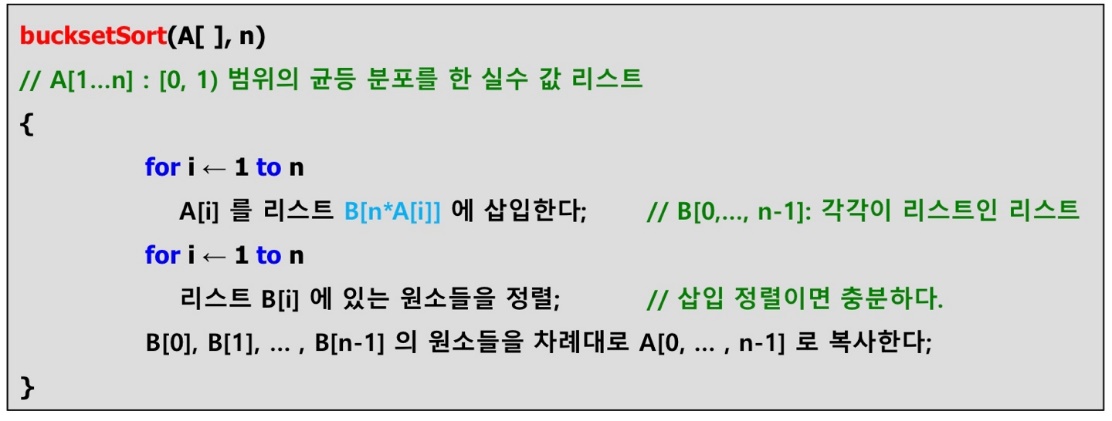
1. **시간 복잡도 :** O(d(n+r))

**03-12. 버킷 정렬(Bucket Sort)**

: 원소들이 균등 분포를 하는 [0, 1) 범위의 실수인 경우

* [0, 1] 범위는 아니어도 쉽게 [0, 1] 범위로 변환할 수 있다.

**# 알고리즘**

****

**# 동작과정**

**텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 도표, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명**

**03-13. 실습보고서2\_연습문제 #02\_삽입정렬 : 알고리즘**

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void insertionSort(int\* pArr, int num); // 삽입 정렬

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

//배열 생성

int arr[arrMAXSIZE] = { 0 };

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

cout << "20215229 이채훈" << endl;

cout << endl;

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

insertionSort(arr, arrMAXSIZE);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

// 삽입 정렬: 오름차순

void insertionSort(int\* pArr, int num) {

int i, j, temp;

for (i = 1; i < num; i++) {

temp = \*(pArr + i);

// temp 가 들어갈 공간 확보

for (j = i - 1; j >= 0 && \*(pArr + j) > temp; j--)

\*(pArr + j + 1) = \*(pArr + j);

\*(pArr + j + 1) = temp;

PRINT(pArr, num);

}

}

**# 연구조사**

**삽입정렬**

: 정렬되지 않은 원소를 정렬된 영역의 적당한 위치에 넣어주는 정렬

**삽입 정렬 수행 과정**

1) 각 단계에서 비 정렬 구역의 첫 번째 원소를 선택한다.

2) 첫 번쨰 원소를 정렬 구역의 원소와 비교하여 적당한 자리에 삽입한다.

**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리 사용

**연산 시간**

1) Best Case : 자료가 이미 정렬되어 있는 경우

2) Worst Case : 자료가 역순으로 정렬되어 있는 경우

**03-14. 실습보고서2\_연습문제 #03-1\_이진탐색 : 재귀적 용법**

#include <iostream>

#define arrMAXSIZE 10

using namespace std;

// 이진 탐색 : 재귀적 용법

int binarySearch(int\* pArr, int first, int last, int key);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 5, 9, 13, 17, 21, 28, 37, 46, 55, 88 };

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "원시 데이터: "; PRINT(arr, arrMAXSIZE);

int key;

while (true) {

cout << "검색 데이터 입력(검색 종료: 0) : "; cin >> key;

if (key == 0)

break;

int index = binarySearch(arr, 0, arrMAXSIZE - 1, key);

if (index == -1)

cout << "없다고!!!" << endl;

else

cout << "검색 데이터: " << arr[index] << "번째 위치 " << index + 1 << endl;

}

return 0;

}

// 이진탐색: 재귀적용법

int binarySearch(int\* pArr, int first, int last, int key) {

// 재귀함수의 탈출 조건

if (first > last)

return -1;

// 검색범위의 중간 원소의 위치 값 계산

int index, mid = (first + last) / 2;

if (\*(pArr + mid) == key)

return mid;

else if (\*(pArr + mid) > key)

index = binarySearch(pArr, first, mid - 1, key);

else if (\*(pArr + mid) < key)

index = binarySearch(pArr, mid + 1, last, key);

return index;

}

**# 연구조사**

이진 탐색 : 중간값을 이용하여 찾고자하는 값인 key값을 찾아내는 알고리즘

**전제조건**

1) 배열이 정렬 상태여야 한다.

2) 각 데이터는 유일한 값을 갖는다.

**과정**

1) 탐색할 범위 경계값의 중간 값을 구한다.

2) 중간값과 key값을 비교한다

3) 중간값 > key값 : 탐색 범위를 첫번째값 ~ 중간값-1 로 바꾼다

중간값 < key값 : 탐색 범위를 중간값+1 ~ 마지막값 으로 바꾼다.

**Q. mid 찾을 때, (pFirst+pLast)/2 를 못하는 이유?**

포인터 연산

포인터끼리의 연산은 뺄셈만 의미를 가진다. 다른 연산은 임의의 위치를 나타낼 뿐이므로 의미를 갖지 않는다.

포인터끼리의 뺄셈은 두 포인터 간의 상대적 길이이다. 이 때 길이는 포인터의 자료형에 따라 다르게 나온다.

ex) int\* 의 경우, 포인터 한칸 당 4byte 를 의미

ex) 만약 int\* a와 b의 차 b-a 의 실제 값이 12이더라도, 이는 12/4바이트인 3으로 나타난다.

주소값을 16진수로 나타낸 수, 둘을 더하면 이를 표현할 수 있는 방법이 없음 + 의미도 없음

pFirst+(pLast-pFirst)/2 -> 주소값의 빼기 연산은 주소값의 차이를 바이트 수로 나눈 것을 반환한다.

**03-15. 실습보고서2\_연습문제 #03-2\_이진탐색 : 비재귀적 용법**

#include <iostream>

#define arrMAXSIZE 10

using namespace std;

// 이진 탐색 : 비재귀적 용법

int\* binarySearch(int\* pArr, int\* pFirst, int\* pLast, int key);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 5, 9, 13, 17, 21, 28, 37, 46, 55, 88 };

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "원시 데이터: "; PRINT(arr, arrMAXSIZE);

int key;

while (true) {

cout << "검색 데이터 입력(검색 종료: 0) : ";

cin >> key;

if (key == 0)

break;

int\* pindex = binarySearch(arr, arr, arr + arrMAXSIZE - 1, key);

if (pindex == NULL)

cout << "없다고!!!" << endl;

else cout << "검색 데이터: " << \*pindex << "번째 위치 " << pindex - arr + 1 << endl;

}

return 0;

}

// 이진탐색: 비재귀적용법

int\* binarySearch(int\* pArr, int\* pFirst, int\* pLast, int key) {

while (pFirst <= pLast) { // 재귀함수의 탈출 조건

// 검색범위의 중간 원소의 위치 값 계산

int\* pMid = pFirst + (pLast - pFirst) / 2;

if (\*pMid == key)

return pMid;

else if (\*pMid < key)

pFirst = pMid + 1;

else if (\*pMid > key)

pLast = pMid - 1;

}

return NULL;

}

**03-16. 실습보고서2\_연습문제 #04\_퀵정렬 : 알고리즘**

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void quickSort(int\* pArr, int\* pFirst, int\* pLast);// 퀵 정렬

int\* Partition(int\* pArr, int\* pfirst, int\* plast);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

quickSort(arr, arr, arr + arrMAXSIZE - 1);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

//퀵 정렬 : 오름차순

void quickSort(int\* pArr, int\* pfirst, int\* plast) {

int\* pmid = NULL;

if (pfirst - pArr < plast - pArr) {

pmid = Partition(pArr, pfirst, plast);//분할 후 기준 값의 위치 값을 반환

PRINT(pfirst, plast-pfirst+1);

quickSort(pArr, pfirst, pmid - 1);//왼쪽 부분 정렬

quickSort(pArr, pmid + 1, plast);//오른쪽 부분 정렬

}

}

int\* Partition(int\* pArr, int\* pfirst, int\* plast) {

int pivot = \*plast; //마지막 원소를 기준 값으로 선택

int i = (pfirst - pArr) - 1;

for (int j = (pfirst - pArr); j < (plast - pArr); j++) {

if (pArr[j] <= pivot)

SWAP(pArr + (++i), pArr + j);//기준 값을 가운데로 위치 시킨다.

}

SWAP(pArr + i + 1, plast);

return pArr + i + 1;//기준 값의 위치 값을 반환

}

**교수님 코드**

//퀵 정렬: 오름차순

void quickSort(int\* pArr, int\* pFirst, int\* pLast) {

//재귀 함수 탈출조건

if (pFirst >= pLast)

return;

//분할: 기준 값의 왼쪽(작은 값)과 오른쪽(큰 값) 부분 집합

int\* mid, \* pi, \* pj;

pi = pFirst - 1;

for (pj = pFirst; pj < pLast; pj++)

if (\*pj <= \*pLast) // 기준 값(pivot): 마지막원소

SWAP(++pi, pj);

SWAP(pi + 1, pLast); // 기준 값을 가운데로

mid = pi + 1; // 기준값의 위치(주소)

//PRINT(pFirst, (int)(pLast-pFirst)+1);

quickSort(pArr, pFirst, mid - 1); //왼쪽 부분 정렬

quickSort(pArr, mid + 1, pLast); //오른쪽 부분 정렬

return;

}

**# 연구조사**

퀵정렬 : 기준 값을 이용하여 부분집합을 두 개로 나누고, 각각을 정렬한다.

기준 값을 중심으로 작은 원소들은 왼쪽 부분집합으로, 큰 원소들은 오른쪽 부분 집합으로 분할하여 정렬하는 방식의 정렬 알고리즘

퀵 정렬 수행 과정

1) Left index를 리스트 가장 왼쪽에, Right index를 리스트 가장 오른쪽에 위치시킨다.

2) Left 에 위치한 원소가 pivot보다 큰 값이 나올 때까지 Left를 리스트의 오른쪽 방향으로 이동시킨다. Right에 위치한 원소가 pivot보다 작거나 같은 값이 나올 때까지 리스트의 왼쪽 방향으로 이동시킨다.

3) Left와 Right가 교차하지 않았다면, 두 인덱스에 위치한 원소를 교환한 후, 2번 과정으로 돌아간다.

4) Left 와 Right가 교차했다면, pivot과 Right에 위치한 원소를 교환한다. 교환한 후의 pivot의 위치를 기준으로 왼쪽과 오른쪽의 부분집합을 구분하고, 두 부분 집합에 대하여 퀵 정렬 알고리즘을 재귀 호출한다.

메모리 사용 공간

n개의 원소에 대하여 n개의 메모리 사용

연산 시간

1) Best Case : 왼쪽 부분집합과 오른쪽 부분집합이 정확히 이등분 되는 경우

2) Worst Case : 부분집합이 1개와 n-1개로 치우쳐 분할되는 경우가 반복되는 경우

**03-17. 실습보고서2\_연습문제 #05\_병합정렬 : 알고리즘**

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void mergeSort(int\* pArr, int\* pFirst, int\* pLast);// 병합 정렬

void merge(int\* pArr, int\* pFirst, int\* pMid, int\* pLast);

void PRINT(int\* pArr, int num) {

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

mergeSort(arr, arr, arr + arrMAXSIZE - 1);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

//병합 정렬 : 오름차순

void mergeSort(int\* pArr, int\* pFirst, int\* pLast) {

int\* mid = NULL;

if ((pFirst - pArr) < (pLast - pArr)) {

mid = pFirst + (pLast - pFirst) / 2; // first와 last 사이의 중간 원소의 위치

int leftSize = mid - pFirst + 1;

int rightSize = pLast - mid;

mergeSort(pArr, pFirst, mid); // 왼쪽 부분집합 정렬

mergeSort(pArr, mid + 1, pLast); // 오른쪽 부분집합 정렬

merge(pArr, pFirst, mid, pLast); // 정렬된 두 부분집합 병합

PRINT(pFirst, leftSize + rightSize);

}

}

void merge(int\* pArr, int\* pFirst, int\* pMid, int\* pLast) {

int i = pFirst - pArr;

int j = (pMid - pArr) + 1;

int t = i;

int\* tempArr = new int[arrMAXSIZE];

while (i <= (pMid - pArr) && j <= (pLast - pArr)) {

if (pArr[i] <= pArr[j]) {

tempArr[t++] = pArr[i++];

}

else {

tempArr[t++] = pArr[j++];

}

}

while (i <= (pMid - pArr))

tempArr[t++] = pArr[i++];

while (j <= (pLast - pArr))

tempArr[t++] = pArr[j++];

//정렬된 상태로 재구성된 temp 배열을 원본 배열 A에 복사

i = pFirst - pArr;

t = i;

while (i <= (pLast - pArr))

pArr[i++] = tempArr[t++];

delete[] tempArr;

}

**# 연구조사**

**병합정렬 :** 정렬된 집합을 합하여 하나의 집합을 만든다.

하나의 배열을 두 개의 균등한 크기로 분할하고 분할된 부분 집합을 정렬한 다음, 두 개의 정렬된 부분 집합를 합하여 전체가 정렬되게 하는 방법이다

**분할 :** 입력 배열을 같은 크기의 2개의 부분 배열로 분할한다.

**정복 :** 부분 배열을 정렬한다. 부분 배열의 크기가 충분히 작지 않으면 순환 호출 을 이용하여 다시 분할 정복 방법을 적용한다.

**결합 :** 정렬된 부분 배열들을 하나의 배열에 합병한다

**병합 정렬 수행 과정**

1) 정렬되지 않은 리스트를 원소 1개의 부분리스트로 분할한다.

2) 부분 리스트가 하나만 남을 때까지 반복 병합하면서 정렬된 부분 리스트를 생성한다.

3) 마지막 하나 남은 부분 리스트가 정렬된 리스트가 된다.

**메모리 사용 공간**

n개의 원소에 대하여 (2\*n)개의 메모리 공간이 사용된다.

**03-18. 중간고사 족보**

**Test2**

#include <iostream>

#define arrMaxsize 10

using namespace std;

int\* binarySearch(int\* pArr, int\* pFirst, int\* pLast, int key);

void OUTPUT(int\* pArr, int num);

int main() {

int arr[arrMaxsize] = { 5,9,13,17,21,28,37,46,55,88 };

cout << "원시 데이터 : ";

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

cout.width(3);

cout << arr[i];

}

int x;

while (true) {

cout << '\n';

cout << "검색 데이터 입력(검색종료 : 0) ";

cin >> x;

if (x == 0) break;

int\* result =binarySearch(arr, arr, arr + arrMaxsize - 1, x);

if (result == NULL) {

cout << "없다고!!!" << endl;

continue;

}

else {

cout << "검색 데이터 위치 : ";

cout << result - arr + 1 << "번째 위치 " << \*result;

}

}

return 0;

}

/\*

1. 중간값 찾기

2. pmid == key 면 return

3. key < pmid면 왼쪽에서 다시 찾기

4. key > pmid면 오른쪽에서 다시 찾기

\*/

int\* binarySearch(int\* pArr, int\* pFirst, int\* pLast, int key) {

while (pFirst-pArr <= pLast-pArr) {

int\* pmid = pFirst + (pLast - pFirst) / 2;

if (\*pmid == key) return pmid;

else if (key < \*pmid) {

pLast = pmid - 1;

continue;

}

else if (key > \*pmid) {

pFirst = pmid + 1;

continue;

}

}

return NULL;

}

**03-19. 송\_이진탐색알고리즘 : 재귀적 용법**

#include <iostream>

using namespace std;

int\* binarySearch(int\* pArr,int\* pFirst,int\* pLast, int key);

void PRINT(int\* pArr,int num);

int main(){

cout << "20193281 송형준 " << endl;

cout << '\n';

int arr[10]={5,9,13,17,21,28,37,46,55,86};

PRINT(arr,10);

cout << '\n';

while(true){

int num;

cout << "검색 데이터 입력(검색 종료 : 0) : ";

cin >> num;

if(num ==0) break;

int\* answer = binarySearch(arr,arr,arr+9,num);

if(answer==NULL) cout << "없다고 !!!!! " << endl;

else {

cout << "검색 데이터의 위치 : " << (int)(answer-arr)+1 << endl;

}

}

return 0;

}

int\* binarySearch(int\* pArr,int\* pFirst,int\* pLast, int key){ // key : 찾아야 하는 값

if(\*pFirst > \*pLast) {

return NULL; //first가 더 크면 아래과정 할 필요 x

}

int\* mid = pFirst+(pLast-pFirst)/2;

if(key==\*mid) return mid;

else if(key < \*mid) binarySearch(pArr,pFirst,mid-1,key);

else if(key > \*mid) binarySearch(pArr,mid+1,pLast,key);

}

void PRINT(int\* pArr,int num){

cout << " 원시 데이터 : ";

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

cout << pArr[i] << " ";

}

}

**03-20. 송\_계수 정렬**

#include <iostream>

#include <ctime>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void countingSort(int\* pArr, int num);

int main(){

int arr[arrMAXSIZE+1]={0};

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0, 99); //0~99 를 생성하는 dis 객체 선언

for(int i=1;i<arrMAXSIZE+1;i++){

\*(arr+i)=dis(gen); //dis안에 난수 생성 엔진을 넣어 숫자 생성

}

cout << " 정렬 전 : ";

for(int i=1;i<arrMAXSIZE+1;i++){

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

}

cout << "\n";

countingSort(arr,arrMAXSIZE);

}

void countingSort(int\* pArr, int num){

int counting[100]={0};

int ans[arrMAXSIZE+1]={0};

for(int i=1;i<num+1;i++){

counting[pArr[i]]++;

}//갯수 세기

for(int i=1;i<100;i++){ //1부터 시작

counting[i]+=counting[i-1];

}

//누적합 배열

for(int i=num;i>=1;i--){

//counting[pArr[i]의 의미 : (정렬하고자 하는 원소)(pArr)가 들어갈 (정답배열)(ans)의 인덱스

ans[counting[pArr[i]]]=pArr[i];

counting[pArr[i]]--;

}

cout << " 정렬 후 : ";

for(int i=1;i<=num;i++){

cout << ans[i] << " ";

}

}

**연구조사**

**계수 정렬** : 각 숫자의 갯수를 센 수, 작은 수부터 갯수만큼 출력

전제조건

1) 정수로 표현 가능한 배열만 정렬할 수 있다.

2) arr의 data값 중 최댓값을 알아야한다.

과정

1) 각 data의 갯수를 센다

2) 카운트 배열에 갯수를 저장한다.

\* 카운트 배열(count\_arr)이란 : count\_arr의 인덱스 = data값, count\_arr의 데이터 = data의 갯수

3) 누적합 배열 생성

-> count\_arr의 data값을 왼쪽부터 계속 누적하여 자기 자신 자리에 대입

4) for문으로 배열을 정리하여 출력

stable 과 unstable

배열 내에 동일한 값이 두개 이상 존재할 때 그 값의 순서가 보장되는 것을 stable하다고 표현한다.

ex) {1,3,2,1} 을 {1ㄱ,3,2,1ㄴ} 이라 표현할 때, 정렬 후의 모습이 {1ㄱ,1ㄴ,2,3}이면 stable하다

만약 {1ㄴ,1ㄱ,2,3} 일 경우 unstable 하다.

stable은 매번 동일한 값을 출력하고 싶은 경우에 사용한다.

stable : mergeSort, insertSort, countSort, bubbleSort

unstable : quickSort,heapSort, selectionSort, shellSort

**rand함수**

C 스타일 난수 생성(srand와 rand 함수)

srand(static\_cast<unsigned int>(time(nullptr)));

static\_cast<int>(rand() % 100);

**문제점**

1. 수학적 알고리즘에 의한 난수 생성이기에, 실제 난수가 아니다.

2. 시드가 너무 느리게 변하기 때문에, 여러 난수 프로그램을 동시에 돌릴 경우 같은 난수가 생성된다.

3. 0~99까지의 숫자 발생 확률이 균등하지 않다.

따라서 c++ 스타일의 난수 생성을 이용한다

<random> 내의 random\_device 클래스 사용

실제 컴퓨터의 무작위적인 요소를 이용해 진정한 난수를 생성 가능하다.

허나 난수 생성 속도가 느리기 때문에, 시드값 초기화에만 사용하고, 이후의 난수는 난수 엔진을 사용하여 생성한다.

**mt19937 gen(시드번호)**

mt19937은 엔진의 한 종류로, <random> 내에는 다른 엔진들 역시 정의되어있다

ex) 객체의 크기가 클 경우 minstd\_rand

이후엔 난수 생성의 범위를 지정해준다.

std::uniform\_int\_distribution<int> : 균등분포, 범위의 숫자를 동일한 확률로 생성한다.

**03-21. 송\_성능평가 : 6개의 정렬성능 비교**

#include <iostream>

#include <ctime>

#include <random>

#define arrSize 30000

using namespace std;

void RANDOM(int\* pArr,int num);

void SWAP(int\* pa,int\* pb);

void PRINT(int\* pArr,int num);

//기본정렬

void selectSort(int\* pArr,int num);

void bubbleSort(int\* pArr,int num);

void insertSort(int\* pArr,int num);

//고급정렬

void mergeSort(int\* pArr,int\* pFirst,int\* pLast);

void merge(int\* pArr,int\* pFirst,int\* pMid, int\* pLast);

void quickSort(int\* pArr,int\* pfirst,int\* plast);

int\* Partition(int\* pArr, int\* pfirst, int\* plast);

void shellSort(int\* pArr, int num);

void intervalSort(int\* pArr, int start, int num, int interval);

int main(){

int arr[arrSize]={0};

clock\_t start;

clock\_t end;

cout << "20193281 송형준 " << endl;

cout << '\n';

//선택정렬

RANDOM(arr,arrSize);

cout << "정렬 전 : " ;

PRINT(arr,arrSize);

start = clock();

selectSort(arr,arrSize);

end = clock();

cout << "선택정렬 : " << (double)(end-start)/CLOCKS\_PER\_SEC << endl;

cout << "정렬 후 : " ;

PRINT(arr,arrSize);

//버블정렬

cout << '\n';

RANDOM(arr,arrSize);

cout << "정렬 전 : " ;

PRINT(arr,arrSize);

start = clock();

bubbleSort(arr,arrSize);

end = clock();

cout << "버블정렬 : " << (double)(end-start)/CLOCKS\_PER\_SEC << endl;

cout << "정렬 후 : " ;

PRINT(arr,arrSize);

//삽입정렬

cout << '\n';

RANDOM(arr,arrSize);

cout << "정렬 전 : " ;

PRINT(arr,arrSize);

start = clock();

insertSort(arr,arrSize);

end = clock();

cout << "삽입정렬 : " << (double)(end-start)/CLOCKS\_PER\_SEC << endl;

cout << "정렬 후 : " ;

PRINT(arr,arrSize);

//병합정렬

cout << '\n';

RANDOM(arr,arrSize);

cout << "정렬 전 : " ;

PRINT(arr,arrSize);

start = clock();

mergeSort(arr,arr,arr+arrSize-1);

end = clock();

cout << "병합정렬 : " << (double)(end-start)/CLOCKS\_PER\_SEC << endl;

cout << "정렬 후 : " ;

PRINT(arr,arrSize);

//퀵정렬

cout << '\n';

RANDOM(arr,arrSize);

cout << "정렬 전 : " ;

PRINT(arr,arrSize);

start = clock();

quickSort(arr,arr,arr+arrSize-1);

end = clock();

cout << "퀵정렬 : " << (double)(end-start)/CLOCKS\_PER\_SEC << endl;

cout << "정렬 후 : " ;

PRINT(arr,arrSize);

//쉘정렬

cout << '\n';

RANDOM(arr,arrSize);

cout << "정렬 전 : " ;

PRINT(arr,arrSize);

start = clock();

shellSort(arr,arrSize);

end = clock();

cout << "퀵정렬 : " << (double)(end-start)/CLOCKS\_PER\_SEC << endl;

cout << "정렬 후 : " ;

PRINT(arr,arrSize);

return 0;

}

void RANDOM(int\* pArr,int num){

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(0,99);

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

\*(pArr+i)=dis(gen);

}

}

void SWAP(int\* pa,int\* pb){

int temp = \*pa;

\*pa=\*pb;

\*pb=temp;

}

void PRINT(int\* pArr,int num){

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

return;

}

void selectSort(int\* pArr,int num){

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

int smallest = i;

for(int j=i+1;j<num;j++){

if(pArr[j] < pArr[smallest]) SWAP(pArr+j,pArr+smallest);

}

}

}

void bubbleSort(int\* pArr,int num){

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

bool state = true;

for(int j=num;j>i;j--){

if(pArr[j] < pArr[j-1]){

SWAP(pArr+j,pArr+(j-1));

state=false;

}

}

if(state) break;

}

}

void insertSort(int\* pArr,int num){

for(int i=1;i<num;i++){

int j = i-1;

int temp = pArr[i];

while(j>=0 && temp<pArr[j]){

pArr[j+1]=pArr[j];

j--;

}

pArr[j+1]=temp;

}

}

void mergeSort(int\* pArr,int\* pFirst,int\* pLast){

int\* mid = NULL;

if((pFirst-pArr) < (pLast-pArr)){

mid = pFirst+(pLast-pFirst)/2;

mergeSort(pArr,pFirst,mid); //처음~중간 정렬

mergeSort(pArr,mid+1,pLast); //중간~끝 정렬

merge(pArr,pFirst,mid,pLast);

}

}

void merge(int\* pArr,int\* pFirst,int\* pMid, int\* pLast){

int i = pFirst-pArr;

int j = (pMid-pArr)+1;

int k = i;

int\* tempArr = new int[arrSize];

while(i <= (pMid-pArr) && j <= (pLast-pArr)){

if(pArr[i] <= pArr[j] ){

tempArr[k++] = pArr[i++];

}

else{

tempArr[k++] = pArr[j++];

}

}

while(i <= (pMid-pArr)) tempArr[k++] = pArr[i++];

while(j <= (pLast-pArr)) tempArr[k++] = pArr[j++];

i=pFirst-pArr;

k=i;

while(i <= (pLast-pArr)) pArr[i++] = tempArr[k++];

delete[] tempArr;

}

void quickSort(int\* pArr,int\* pfirst,int\* plast){

int\* pmid=NULL;

if(pfirst-pArr < plast-pArr){ //기준값보다 작으면

pmid = Partition(pArr,pfirst,plast);

quickSort(pArr,pfirst,pmid-1);

quickSort(pArr, pmid+1, plast);

}

}

int\* Partition(int\* pArr, int\* pfirst, int\* plast){

int pivot = \*plast; //마지막 원소 기준

int i = (pfirst-pArr)-1; //first-1,

for(int j=(pfirst-pArr);j<(plast-pArr);j++){

if(pArr[j]<=pivot) SWAP(pArr+(++i),pArr+j);

}

SWAP(pArr+i+1,plast);

return pArr+i+1;

}

// 쉘 정렬

void shellSort(int\* pArr, int num) {

int interval = num;

double result;

while (interval >= 1) {

interval = interval / 2;

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

intervalSort(pArr, i, num, interval);

}

}

return;

}

void intervalSort(int\* pArr, int start, int num, int interval) {

int temp, i, j;

for (i = start+interval; i < num; i = i+interval) { // 부분리스트 원소 수만큼 반복 + 첫 원소는 정렬됐다 가정

temp = \*(pArr + i); //정렬 안된 부분리스트의 첫 원소

// temp가 들어갈 공간을 확보

for (j = i-interval; 0 <= j && \*(pArr + j) > temp; j = j-interval)

\*(pArr + j + interval) = \*(pArr + j);

\*(pArr + j + interval) = temp; //원소 삽입

}

return;

}

**연구조사**

**1. 선택정렬 : 가장 작은 원소를 선택하여 정렬**

제자리 정렬의 하나로서, 입력 배열(정렬되지 않은 값들) 이외에 다른 추가 메모리를 요구하지 않는 정렬 방법이다. 해당 순서에 원소를 넣을 위치는 이미 정해져 있고, 어떤 원소를 넣을지 선택하는 알고리즘

첫 번째 순서에는 첫 번째 위치에 가장 최솟값을 넣는다.

두 번째 순서에는 두 번째 위치에 남은 값 중에서의 최솟값을 넣는다.

선택 정렬 수행 과정

1) 비정렬구역 중에서 최솟값을 찾는다.

2) 그 값을 비정렬구역의 맨앞과 교체한다.

3) 하나의 원소만 남을 때까지 위의 1~3 과정을 반복한다

메모리 사용 공간

n개의 원소에 대하여 n개의 메모리 사용

연산 시간

1) Best Case : 자료가 이미 정렬되어 있는 경우, 총 소요시간: n(n-1)/2

2) Worst Case : 정렬된 자료에서 최솟값만 맨 뒤에 위치한 경우, 총 소요시간: n(n-1)/2 + (n-1)

**2. 버블정렬 : 인접한 두 원소의 크기를 비교하고, 작은 원소를 왼쪽으로 옮긴다.**

거품이 수면 위로 올라오는 듯한 모습을 닮아 붙여진 이름, 1회전을 수행하고 나면 가장 큰 자료가 맨 뒤로 이동하므로 2회전에서는 맨 끝에 있는 자료는 정렬에서 제외되고, 2회전을 수행하고 나면 끝에서 두 번째 자료까지는 정렬에서 제외된다. 이렇게 정렬을 1회전 수행할 때마다 비정렬구역에서 제외되는 데이터가 하나씩 늘어난다.

버블 정렬 수행 과정

1) 비 정렬구역의 맨 칸 원소와 한칸 앞의 원소를 비교한다.

2) 맨 뒤 칸의 원소가 더 작을 경우 둘의 위치를 변경한다.

3) 비 정렬구역의 맨 앞 경계를 한 칸 뒤로 미룬다.

4) 위 과정을 반복한다.

메모리 사용 공간

n개의 원소에 대하여 n개의 메모리 사용

연산 시간

1) Best Case : 자료가 이미 정렬되어 있는 경우, 총 소요시간: n(n-1)/2

2) Worst Case : 자료가 역순으로 정렬되어 있는 경우, 총 소요시간: n(n-1)

구현이 매우 간단하나, 구조 자체의 비효율성, 그리고 일반적으로 자료의 SWAP이 자료를 이동시키는 것보다 더 복잡하기 때문에 단순성에도 불구하고 거의 쓰이지 않는다.

**3. 삽입정렬 : 정렬되지 않은 원소를 정렬된 영역의 적당한 위치에 넣어주는 정렬**

삽입 정렬 수행 과정

1) 각 단계에서 비 정렬 구역의 첫 번째 원소를 선택한다.

2) 첫 번쨰 원소를 정렬 구역의 원소와 비교하여 적당한 자리에 삽입한다.

메모리 사용 공간

n개의 원소에 대하여 n개의 메모리 사용

연산 시간

1) Best Case : 자료가 이미 정렬되어 있는 경우

2) Worst Case : 자료가 역순으로 정렬되어 있는 경우

**4. 병합정렬 : 정렬된 집합을 합하여 하나의 집합을 만든다.**

하나의 배열을 두 개의 균등한 크기로 분할하고 분할된 부분 집합을 정렬한 다음, 두 개의 정렬된 부분 집합를 합하여 전체가 정렬되게 하는 방법이다

분할 : 입력 배열을 같은 크기의 2개의 부분 배열로 분할한다.

정복 : 부분 배열을 정렬한다. 부분 배열의 크기가 충분히 작지 않으면 순환 호출 을 이용하여 다시 분할 정복 방법을 적용한다.

결합 : 정렬된 부분 배열들을 하나의 배열에 합병한다

병합 정렬 수행 과정

1) 정렬되지 않은 리스트를 원소 1개의 부분리스트로 분할한다.

2) 부분 리스트가 하나만 남을 때까지 반복 병합하면서 정렬된 부분 리스트를 생성한다.

3) 마지막 하나 남은 부분 리스트가 정렬된 리스트가 된다.

메모리 사용 공간

n개의 원소에 대하여 (2\*n)개의 메모리 공간이 사용된다.

**5. 퀵정렬 : 기준 값을 이용하여 부분집합을 두 개로 나누고, 각각을 정렬한다.**

기준 값을 중심으로 작은 원소들은 왼쪽 부분집합으로, 큰 원소들은 오른쪽 부분 집합으로 분할하여 정렬하는 방식의 정렬 알고리즘

퀵 정렬 수행 과정

1) Left index를 리스트 가장 왼쪽에, Right index를 리스트 가장 오른쪽에 위치시킨다.

2) Left 에 위치한 원소가 pivot보다 큰 값이 나올 때까지 Left를 리스트의 오른쪽 방향으로 이동시킨다. Right에 위치한 원소가 pivot보다 작거나 같은 값이 나올 때까지 리스트의 왼쪽 방향으로 이동시킨다.

3) Left와 Right가 교차하지 않았다면, 두 인덱스에 위치한 원소를 교환한 후, 2번 과정으로 돌아간다.

4) Left 와 Right가 교차했다면, pivot과 Right에 위치한 원소를 교환한다. 교환한 후의 pivot의 위치를 기준으로 왼쪽과 오른쪽의 부분집합을 구분하고, 두 부분 집합에 대하여 퀵 정렬 알고리즘을 재귀 호출한다.

메모리 사용 공간

n개의 원소에 대하여 n개의 메모리 사용

연산 시간

1) Best Case : 왼쪽 부분집합과 오른쪽 부분집합이 정확히 이등분 되는 경우

2) Worst Case : 부분집합이 1개와 n-1개로 치우쳐 분할되는 경우가 반복되는 경우

**6. 쉘 정렬 : 일정한 간격으로 부분집합을 나누고, 각 부분집합의 정렬을 수행한다.**

일정 간격 떨어져 있는 원소들끼리 부분집합을 구성한 후, 각 부분집합의 원소들에 대해서 삽입 정렬을 수행하되, 간격을 줄여가며 삽입 정렬을 반복하여 전체 원소들을 정렬하는 방식의 정렬 알고리즘

쉘 정렬 수행 과정

1) 첫 번째 원소와 첫 번째 원소로부터 interval만큼 간격이 있는 원소들을 부분 집합에 포함

2) 부분 집합에 포함된 원소들을 삽입 정렬 수행

3) 두 번째 원소와 두 번째 원소로부터 h의 배수만큼 간격이 있는 원소들을 부분 집합에 포함

4) 부분 집합에 포함된 원소들을 삽입 정렬 수행

5) 위 과정을 반복

메모리 사용 공간

n개의 원소에 대하여 n개의 메모리와 매개변수 h에 대한 저장공간 사용

연산 시간 : 원소의 상태와 상관없이 간격에 의해 결정