Documentation

1. randomNum
2. insertionSort
3. heapsort
4. mergeSort
5. quickSort –

    a) first or last element as a pivot

    b) random number as a pivot

    c) median of three partitioning as a pivot
6. 시간 측정
7. comparison

1. randNum

```
//random으로 값 생성해주는 함수
int randomNum(int adr) {
    static int count = 0;
    int random = 8253729 * adr + 2396403 + count;
    count++;
    if (random < 0) random  = -random;
    return random % SIZE;
}
```

```
int arr[SIZE]; //애초에 배열에 주소값이 랜덤으로 배정
for (int i = 0; i < SIZE; i++) {
    arr[i] = randomNum((int)&arr[i]);
}
```

(in main)

주어진 조건에 내장함수 rand를 사용하지 않고, 임의로 random함수를 만들라고 하였다. 그래서 이 함수는 random으로 숫자를 반환하는 함수를 만들었다. 맨 처음 <time.h> 라이브러리를 사용해서도 만들어보았고 여러가지 방법을 시도해 보았다. 그리고 택한 방법은 main 함수에서 배열을 생성할 때 컴파일러가 배열의 주솟값을 매번 다른 값으로 할당하는 것에서 idea를 얻어 그 주솟값을 randNum 함수에 넘겨주었다. 그리고 그 주솟값에 큰 수를 곱하고 더하여서 정수형의 범위를 넘어서게 하여서 예측할 수 없는 값, 즉 난수가 나온다. 그리고 여기서 음수의 값이 나올 수도 있어서 if문으로 처리를 해주었고, 총 SIZE=1024로 지정하여서 0부터 1023까지의 난수가 생성되게 하였다. 하지만 이렇게만 하면, 난수가 나오지만 몇 번 반복되는 구문이 생겼다. 그래서 static int 형으로 count를 선언하여서 count의 값이 계속 늘어나면서 반복되는 구문이 생기지 않도록 구성하였더니 난수 생성이 성공적으로 되었다.

2. insertionSort

```c
// Insertion sort
void insertionSort(int* arr) {
    for (i=1; i < SIZE ; i++) { // 1부터 마지막 인덱스까지
        int key=arr[i]; // key는 현재 i의 키값
        j = i - 1;// i-1까지는 정렬된 배열 -> arr[i]의 값을 앞의 정렬된 배열에 삽입

        while (j >= 0 && arr[j] > key) { // j는 0보다 작으면 배열이탈하므로 && j번째 원소가 arr[i]보다 크면 자리바꿈
            arr[j + 1] = arr[j];
            j--;
            insertion_count++;
        }
        if (j != -1) insertion_count++;
        //비교 후 count증가
        arr[j + 1] = key;
    }
}
```

1부터 마지막 index까지 index의 값을 늘려서 정렬된 배열에 하나의 원소의 값을 그 정렬된 배열의 원소와 비교하며 삽입하는 insertion sort 알고리즘을 구성하였다.

3. heapSort

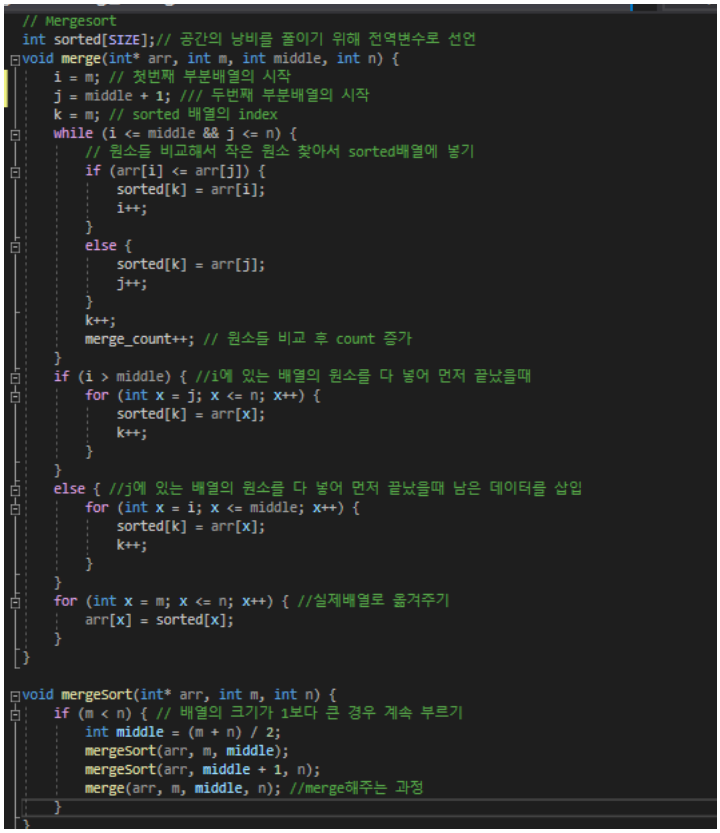
```c
// Heapsort
void heapSort(int* arr) {
    for (i = 1; i < SIZE; i++) {
        int child = i;
        do {
            int root = (child - 1) / 2; // 자식을 토대로 부모를 생성
            if (arr[root] < arr[child]) { // 자식의 값이 더 크면 값을 서로 바꿔줌
                swap(&arr[root], &arr[child]);
            }
            heap_count++; // 비교후 count 증가
            child = root; // 자식의 값이 부모가 되어 root에 있는 값이 최대가 되게 반복적으로 수행
        } while (child != 0); // 최대 heap구조만듬
    }
    for (i = SIZE - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]); // 마지막 원소와 첫번째 원소의 자리를 바꿈 -> 마지막 자리에 계속 들어가게 되어 오름차순 배열
        int root = 0;
        int child = 1;
        do {
            child = root * 2 + 1;
            if (child < i - 1 && arr[child] < arr[child + 1]) { // 자식중에 더 큰값을 찾음
                child++;
            }
            heap_count++; // 자식 원소간의 비교 후 count 증가
            if (child < i && arr[child] > arr[root]) { // 자식의 값이 더 크다면 부모와 자식을 바꿈
                swap(&arr[child], &arr[root]);
            }
            heap_count++; // 자식원소와 부모원소 비교 후 count 증가
            root = child; // 자식의 값을 부모로 넣어줌
        } while (child < i); // root의 값은 자식보다 작을 수 있음 -> 다시 heap구조를 만들어줌
    }
}
```

자식 노드를 먼저 처음 데이터에 넣어주고, 그리고 그 child로 root node를 형성한다. 이때 자식의 값이 더 크면 root 와 child의 값을 바꿔주고, 그 root의 값이 자식이 되고, root에 있는 값이 최대가 될 때까지 반복적으로 수행하는 max heap을 먼저 만들어준다.

그리고 나서 sort를 하는데 최대 heap이므로 root의 값은 가장 큰 값이다. 그래서 가장 마지막 값과 바꿔주고, 마지막 값이 root가 되어 작은 값이 들어갔다. 그래서 이때는 root의 값을 기준으

로 child를 만들었고, child와 root를 비교하면서 다시 max heap을 만든다. 이렇게 배열을 sort하는 heap sort 알고리즘을 구성하였다.

4. mergeSort



```
// Mergesort
int sorted[SIZE];// 공간의 낭비를 줄이기 위해 전역변수로 선언
void merge(int* arr, int m, int middle, int n) {
    i = m; // 첫번째 부분배열의 시작
    j = middle + 1; /// 두번째 부분배열의 시작
    k = m; // sorted 배열의 index
    while (i <= middle && j <= n) {
        // 원소들 비교해서 작은 원소 찾아서 sorted배열에 넣기
        if (arr[i] <= arr[j]) {
            sorted[k] = arr[i];
            i++;
        }
        else {
            sorted[k] = arr[j];
            j++;
        }
        k++;
        merge_count++; // 원소들 비교 후 count 증가
    }
    if (i > middle) { //i에 있는 배열의 원소를 다 넣어 먼저 끝났을때
        for (int x = j; x <= n; x++) {
            sorted[k] = arr[x];
            k++;
        }
    }
    else { //j에 있는 배열의 원소를 다 넣어 먼저 끝났을때 남은 데이터를 삽입
        for (int x = i; x <= middle; x++) {
            sorted[k] = arr[x];
            k++;
        }
    }
    for (int x = m; x <= n; x++) { //실제배열로 옮겨주기
        arr[x] = sorted[x];
    }
}

void mergeSort(int* arr, int m, int n) {
    if (m < n) { // 배열의 크기가 1보다 큰 경우 계속 부르기
        int middle = (m + n) / 2;
        mergeSort(arr, m, middle);
        mergeSort(arr, middle + 1, n);
        merge(arr, m, middle, n); //merge해주는 과정
    }
}
```

우선 mergeSort를 main함수에서 호출하면, 배열의 크기가 1이 될 때까지 divide를 한다. 그리고 나서 merge를 수행하는데, 이때 merge함수에서 두개의 부분배열을 비교하여서 sort를 한다. 그리고 merge해주는 과정을 반복하여 정렬된 배열을 만든다. 이렇게 merge sort알고리즘을 구성하였다.

5. quickSort

```
//quick sort_version1
int partition_1(int* arr, int start, int end) {
    int pivot = end; // pivot이 end값에 가도록
    i = start - 1;

    for (int j = start; j <= pivot - 1; j++) {
        if (arr[j] <= arr[pivot]) {
            i++;
            swap(&arr[i], &arr[j]);
        }
        quick_count_1++; //pivot이랑 비교 후 count 증가
    }
    swap(&arr[i + 1], &arr[pivot]);

    return i + 1;
}

void quickSort_1(int* arr, int start, int end) {
    if (start < end) {
        int m = partition_1(arr, start, end);
        quickSort_1(arr, start, m - 1);
        quickSort_1(arr, m + 1, end);
    }
}
```

quick sort는 기본적으로 pivot을 정하고, 그 pivot으로 배열들의 원소를 비교하여 pivot값을 기준으로 나눠 sort를 진행하는 재귀함수이다.

a) first or last element as a pivot

```
int partition_1(int* arr, int start, int end) {
    int pivot = end; // pivot이 end값에 가도록
    i = start - 1;
```

pivot을 마지막 값으로 설정해주었다.

b) random number as a pivot

```
int partition_2(int* arr, int start, int end) {
    int num;
    i = start - 1;
    int pivot = randomPivot((int)&num,start,end);
    if (pivot != end) {
        swap(&arr[pivot], &arr[end]);
        pivot = end;
    }
```

```
int randomPivot(int adr,int start, int end) {
    static int count = 0;
    int random = 8253729 * adr + 2396403 + count;
    count++;
    if (random < 0) random = -random;
    random = random % (end - start) + start;
    //cout << random << "\n";
    return random;
}
```

randomPivot 함수를 이용하여 랜덤하게 pivot값을 뽑았고, 그 pivot값을 마지막으로 넘겨 quick sort가 진행되게 했다. randomPivot 함수는 randomNum 함수와 동일한 방식으로 만들었고, 특이한점은 start와 end 값을 넘겨주어 그 배열의 index안에서 나올 수 있도록 구성하였다.

c) median of three partitioning as a pivot

```
int partition_3(int* arr, int start, int end) {
    int mid = (start + end) / 2;
    int pivot = 0;
    if (arr[mid] > arr[start]) {
        if (arr[mid] < arr[end]) {
            pivot = mid;
        }
        else {
            if (arr[start] > arr[end]) {
                pivot = start;
            }
            else {
                pivot = end;
            }
        }
    }
    else {
        if (arr[start] < arr[end]) {
            pivot = start;
        }
        else {
            if (arr[mid] > arr[end]) {
                pivot = mid;
            }
            else {
                pivot = end;
            }
        }
    }

    swap(&arr[pivot], &arr[end]);
    pivot = end;// pivot이 end값에 가도록
    i = start - 1;
```

start, end를 이용하여 그 가운데의 값을 찾고 그 3개의 값 중 중간 값을 찾아 그 값을 end로 보내서 quick sort를 진행하도록 만들었다.

6. 시간측정

```
//시간 측정 및 sort실행
auto start_time1 = chrono::high_resolution_clock::now();
insertionSort(arr1);
auto end_time1 = chrono::high_resolution_clock::now();

//print
//cout << "insertion sort: ";
//arrPrint(arr1);
cout << "time : " << (end_time1 - start_time1).count() << endl;
```

<time.h> 라이브러리를 사용해 구성하려고 했으나 배열의 데이터가 많지 않아서 너무 모든 정렬이 너무 빠르게 정렬이 되었다. 그래서 정확한 값을 볼 수 없었다. 찾아보니 <chrono>라는 라이브러리가 있었고, 이 라이브러리는 나노세컨드까지 시간을 측정하여 더 정밀하게 정렬의 시간복잡도를 확인할 수 있다.

7. comparison

ex)

```
void insertionSort(int* arr) {
    for (i=1; i < SIZE ; i++) { // 1부터 마지막 인덱스까지
        int key=arr[i]; // key는 현재 i의 키값
        j = i - 1;// i-1까지는 정렬된 배열 -> arr[i]의 값을 앞의 정렬된 배열에 삽입

        while (j >= 0 && arr[j] > key) { // j는 0보다 작으면 배열이탈하므로 && j번째 원소가 arr[i]보다 크면 자리바꿈
            arr[j + 1] = arr[j];
            j--;
            insertion_count++;
        }
        if (j != -1) insertion_count++;
        //비교 후 count증가
        arr[j + 1] = key;
    }
}
```

```
while (i <= middle && j <= n) {
    // 원소들 비교해서 작은 원소 찾아서 sorted배열에 넣기
    if (arr[i] <= arr[j]) {
        sorted[k] = arr[i];
        i++;
    }
    else {
        sorted[k] = arr[j];
        j++;
    }
    k++;
    merge_count++; // 원소들 비교 후 count 증가
```

배열의 원소들을 비교하는 comparison은 배열이 값을 비교할 때 count라는 변수를 써서 값을 증가시켜주는 형태로 구성하였다. 그래서 원소를 서로 비교하고 그 실행이 true이든 false이든 비교는 하였으므로 count를 적절히 증가시켜주는 형태로 comparison을 하였다.

Lessons

코드를 실행시켜 나온 output file을 보면, 대략 insertion sort가 가장 많은 비교가 발생하였고, 따라서 가장 오래 실행시간이 되는 것을 확인해 볼 수 있다. 왜냐하면 시간복잡도가 O(n^2)이기 때문이다. 그래서 insertion, heap, merge, quick에 비해 느리고 비교횟수도 많다는 것을 직접 확인해 볼 수 있다. 그리고 heap sort는 insertion sort에 비해 시간복잡도가 작은 O(nlogn)이라서 훨씬 더 빠른 결과를 내는 것을 알 수 있다. 그리고 merge sort의 시간복잡도는 O(nlogn)으로 heap sort와 같지만 heap sort에 비해 빠른 결과가 나오는 것을 확인할 수 있다. 이것은 많은 data들이 있을 때, 비교하는 연산이 heap sort에 비해 적어서 이런 결과가 생성되는 것을 알 수 있다. 그리고 quick sort를 살펴보면 우선 pivot이 start또는 end값에 있을 때는 merge sort와 비슷한 시간의 결과가 나오거나 좀더 큰 것을 살펴볼 수 있는데, 이는 아마도 pivot을 뒤로 계속 보내는 과정에서 시간이 걸리는 것 같다. 그리고 quick sort의 pivot 값이 잘못 설정되면 시간복잡도는 최악의 경우인 O(n^2)이므로 배열이 어떻게 나열되어 있는지에 따라 다른 결과를 생성하는 것 같다. random으로 pivot을 설정하는 것은 pivot값을 start또는 end에 넣었을 때 즉 앞의 결과와 비슷하거나 더 시간이 오래 걸리는데, 이는 random으로 pivot을 뽑는 과정에서 더 오래 걸리는 것이라고 파악할 수 있다. 그리고 pivot을 median of three partitioning으로 했을 때는 quick sort에서 가장 작은 시간을 갖는다. 이는 pivot을 median of three partitioning으로 뽑는 것이 배열을 부분배열로 나누는데 가장 효율적이라고 파악할 수 있다.

결국, 같은 O(nlogn)인 heap sort, merge sort, quick sort라고 해도 비교적 많은 양의 data를 넣어보면 맨 처음 나열 되어있는 배열의 순서, pivot을 설정하는 과정 등에 따라 각각 comparison 횟수와 실제 측정 시간이 정렬 알고리즘마다 다를 수 있다는 것을 확인할 수 있다. 그래서 정렬하려는 배열의 특징에 따라 알맞은 알고리즘을 선택해서 정렬해야 하고, 정렬하려는 목적에 따라서도 알맞은 알고리즘을 선택해야한다. 예를 들어, 가장 큰 값 또는 가장 작은 값을 효율적으로 찾기 위해서는 heap sort를 사용해야 하고, stable한 알고리즘을 원한다면 merge sort를 사용해야 한다. 이 과제를 통해 배운 점은 단순히 코딩만 잘하는 것이 아니라 각각 정렬 알고리즘이 어떤 특징을 갖고 있는지, 어떻게 적용해야 할지를 깨닫게 되었고 시간과 비교횟수를 통해 배열의 특성에 맞게 정렬 알고리즘을 사용해야 한다는 것을 알게 되었다.