

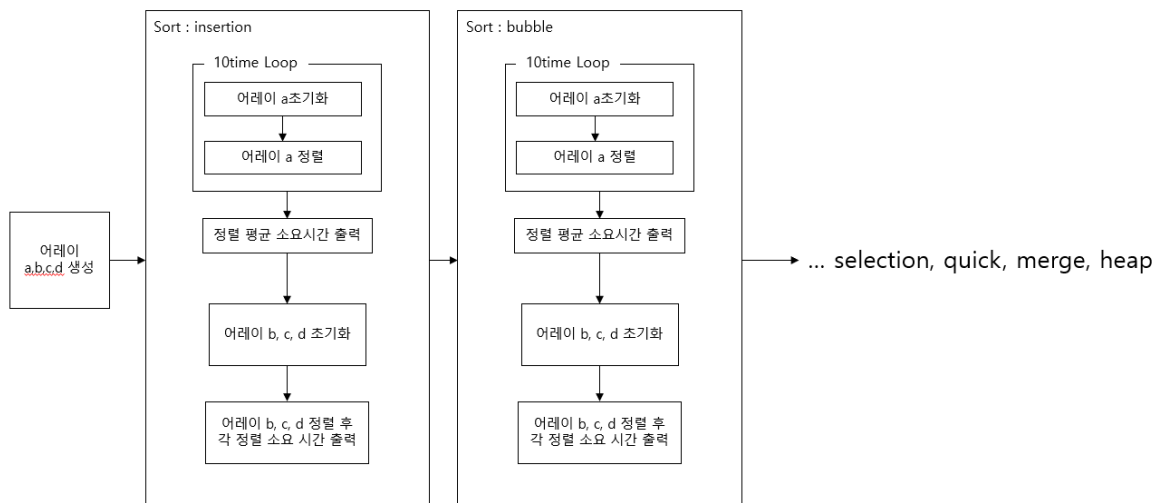
자료구조 Assignment 6 보고서

소프트웨어학과
202220757 김재학

목차

1. 프로그램 실행 과정
2. 가산점 항목 및 중요 코드 설명
3. 정렬시간 실험 결과 및 고찰

1. 프로그램 실행 과정



위의 그림은 프로그램의 실행과정을 간단하게 나타낸 그림이다. 이 프로그램은 시작 시 int형 어레이 a, b, c, d를 10005만큼의 크기만큼 생성한다. 그 후 여러가지 방법으로 정렬을 반복하게 되는데, 정렬 전에 어레이들은 random함수를 이용해서 초기화 된다. a는 완전 무작위의 값을 갖는 어레이로 초기화되고, b는 LOO가 약 10%인 부분 정렬된 어레이, c는 완전 정렬된 어레이, 그리고 d는 완전 역순 정렬 어레이로 초기화 된다. 이 어레이들은 각 정렬 전에 실행되게 된다. 이중 a는 각 정렬을 10번 반복하기 때문에 초기화 또한 10회 시행된다.

정렬이 끝날 때마다 각 정렬을 실행할 때 소요된 시간을 출력한다. a는 10회 정렬되는 동안 수행된 시간의 평균을 출력하며, 나머지 어레이는 각 정렬의 소요시간을 출력한다.

2. 가산점 항목 및 중요 코드 설명

2.1. 가산점 항목 설명

과제에서 주어진 대로 Quick sort의 기능을 개선하였다. 우선 작성된 Quick sort 코드는 다음과 같다.

```
void quicksort(int a[], int L, int R) {
    int left = L, right = R;
    if (left > right) return;
    int pivot = a[(L + R) / 2];    // pivot을 median으로 설정
    int temp;
    do {
        while (a[left] < pivot)
            left++;
        while (a[right] > pivot)
            right--;
        if (left <= right) {
            temp = a[left];
            a[left] = a[right];
            a[right] = temp;

            left++;
            right--;
        }
    } while (left <= right);

    // 분할된 배열중 길이가 짧은 것부터 실행
    if ((right - L) < (R - left)) {
        quicksort(a, L, right);
        quicksort(a, left, R);
    }
    else {
        quicksort(a, left, R);
        quicksort(a, L, right);
    }
}
```

배열과 배열의 시작주소, 배열의 마지막 주소를 인자로 전달받으면 QuickSort를 실행하게 된다. 정렬에 필요한 pivot은 배열의 중앙값으로 설정이 되게 된다. 분할이 끝나면 스택 공간을 효율적으로 실행하기 위해 분할된 길이가 짧은 부분부터 이 함수를 재귀 실행하게 된다.

분할된 길이가 짧은 부분부터 함수를 재귀하는 이유는 최대한으로 시스템 스택의 깊이가 깊어지는 것을 막기 위해서이다. 이 함수는 분할한 만큼 함수를 다시 call 하게 되는데, 분할된 결과 중 길이가 짧을수록 다시 분할되는 수가 적을 것이고, 이는 스택을 최소한으로 쌓을 수 있다는 것을 의미한다.

2.2. 중요 코드 설명

Quick sort를 제외한 나머지 정렬 알고리즘들은 강의노트의 코드를 참고하였기 때문에 거의 유사한 형태를 가지고 있다. 특이한 점이 없기 때문에 이 보고서에서는 설명을 생략한다.

생성된 각 어레이를 초기화 시키는 함수는 `void initArray()`와 `void initArray_a();` 이다. 우선 `initArray` 함수는 어레이 `a`를 제외한 나머지 어레이를 각 조건에 맞춰서 초기화 시킨다. `c`와 `d`어레이는 단순 반복문을 이용해 초기화 하였고, `b`는 다음과 같은 코드를 이용해 L00가 10%가 되도록 초기화 하였다.

```
b[0] = rand();
for (int i = 1; i < SIZE / 10; i++) {    //b init
    b[i] = rand() - b[0];
}
for (int i = SIZE / 10; i < SIZE; i++) {
    b[i] = rand();
}
```

`b`의 첫번째 원소를 `rand`함수를 이용해 초기화 하고 두번째 원소부터 전체 사이즈의 약 10%가 되는 지점까지 `rand`함수로 생성된 숫자에서 첫번째 원소의 값을 빼서 첫번째 원소보다 작은 값을 갖도록 만들었다. 그리고 나머지 원소는 `rand`함수로 초기화 하였다. 이렇게 L00가 약 10%가 될 수 있도록 초기화 하였다.

`initArray_a` 함수는 `a`어레이만 초기화 시키는 함수이다. 프로그램 실행 과정 중 `a`의 소요시간 측정은 4번 반복한 결과의 평균값을 요구하기 때문에 `a`어레이만 초기화 시키는 함수를 따로 만들었다. 이 함수는 단순 반복으로 `a`의 각 원소들을 `rand`함수를 이용해 초기화한다.





각 정렬의 시간측정은 다음과 같은 코드로 측정하였다.

```
clock_t start = clock();
insertionSort(b);
clock_t end = clock();
```

위의 코드는 `b`어레이에 대하여 `insertion sort`를 진행하는 코드이다. `clock()`함수를 이용해 현재시간을 측정하여서 ‘측정의 끝 시간 - 측정 시작 시간’으로 정렬 소요시간을 구하였다. 구해진 시간은 `ms`단위로 기록된다.

3. 정렬 시간 실험 결과 및 고찰

프로그램 시작 시 각 어레이의 초기화 된 결과를 디버깅 툴을 통해 확인한 결과는 다음과 같다.

Name	Value	Type
▶  a	0x00007ff796357e00 {4189, 4912, 16374, 9945, 17346, 27840, 21834, 2315, 7639, 9911, 28672, ...}	int[10005]
▶  b	0x00007ff796361a60 {8204, 1070, 23951, 10332, -5869, 4551, 11415, 23826, 18893, 17989, 443...	int[10005]
▶  c	0x00007ff79636b6c0 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, ...}	int[10005]
▶  d	0x00007ff796375320 {10005, 10004, 10003, 10002, 10001, 10000, 9999, 9998, 9997, 9996, 9995, ...}	int[10005]

이를 각 정렬 알고리즘을 이용해 정렬하게 되면 다음 같은 값을 가지게 된다.

Name	Value	Type
▶ a	0x00007ff796357e00 {2, 5, 6, 13, 14, 15, 23, 23, 24, 30, 34, 35, 36, 38, 39, 39, 40, 42, 49, 53, 60, ...}	int[10005]
▶ b	0x00007ff796361a60 {-15547, -15523, -15464, -15443, -15374, -15334, -15320, -15207, -15161, -15129, ...}	int[10005]
▶ c	0x00007ff79636b6c0 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, ...}	int[10005]
▶ d	0x00007ff796375320 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, ...}	int[10005]

이러한 식으로 프로그램은 각 어레이를 초기화 시키고 정렬하면서 실행되게 된다.

다음 사진은 프로그램의 콘솔 출력화면이다. 3번 실행한 결과 사진을 첨부하였다.

```

선택 Microsoft Visual Studio Debug Console
====InsertionSort====
a_average: 111.400000
b: 88.000000
c: 0.000000
d: 193.000000
====BubbleSort====
a_average: 490.500000
b: 461.000000
c: 192.000000
d: 355.000000
====SelectionSort====
a_average: 170.500000
b: 168.000000
c: 168.000000
d: 182.000000
====QuickSort====
a_average: 2.000000
b: 2.000000
c: 1.000000
d: 1.000000
====MergeSort====
a_average: 2.700000
b: 3.000000
c: 2.000000
d: 1.000000
====HeapSort====
a_average: 2.500000
b: 3.000000
c: 1.000000
d: 2.000000
done!
D:\OneDrive - 아주대학교\아주대학교\아주대 1학년\2학기\자료구조\assignment\assignment 6\Project1\wx64\Debug\Project1.exe
(process 27100) exited with code 0.
Press any key to close this window . . .

```

```

Microsoft Visual Studio Debug Console
====InsertionSort====
a_average: 109.200000
b: 95.000000
c: 0.000000
d: 190.000000
====BubbleSort====
a_average: 490.500000
b: 522.000000
c: 182.000000
d: 280.000000
====SelectionSort====
a_average: 163.000000
b: 165.000000
c: 169.000000
d: 195.000000
====QuickSort====
a_average: 2.400000
b: 2.000000
c: 0.000000
d: 0.000000
====MergeSort====
a_average: 2.800000
b: 3.000000
c: 1.000000
d: 1.000000
====HeapSort====
a_average: 2.900000
b: 4.000000
c: 2.000000
d: 2.000000
done!
D:\OneDrive - 아주대학교\아주대학교\아주대 1학년\2학기\자료구조\assignment\assignment 6\Project1\wx64\Debug\Project1.exe
(process 34092) exited with code 0.
Press any key to close this window . . .

```

```
Microsoft Visual Studio Debug Console

====InsertionSort====
a_average: 116.900000
b: 87.000000
c: 0.000000
d: 295.000000
====BubbleSort====
a_average: 500.300000
b: 481.000000
c: 184.000000
d: 329.000000
====SelectionSort====
a_average: 175.000000
b: 167.000000
c: 165.000000
d: 178.000000
====QuickSort====
a_average: 2.400000
b: 2.000000
c: 0.000000
d: 1.000000
====MergeSort====
a_average: 2.700000
b: 3.000000
c: 2.000000
d: 1.000000
====HeapSort====
a_average: 2.100000
b: 2.000000
c: 1.000000
d: 2.000000
done!
D:\OneDrive - 아주대학교\아주대학교\아주대 1학년\2학기\자료구조\assignment\assignment 6\Project1\wx64\Debug\Project1.exe
(process 12688) exited with code 0.
Press any key to close this window . . .
```

실험 결과 전체적으로 소요시간은 bubble > insertion > selection > quick = merge = heap 순서로 나타났다. 이는 각 정렬의 이론적 시간 복잡도의 비교와 거의 유사한 정도로 보인다. 특히 quick, merge, heap 정렬과 bubble, insertion, selection 정렬과 비교하였을 때 시간이 큰 차이가 나는 것을 알 수 있는데, 이는 평균 시간복잡도가 n^2 과 $n \log n$ 으로 사이즈가 10005인 상황에서 큰 차이가 나기 때문이다.

quick, merge, heap의 더 자세한 비교를 위해 어레이의 사이즈를 약 10배의 사이즈인 100000으로 늘려서 이 세개의 정렬만 다시 비교를 해보았다.

```
Microsoft Visual Studio Debug Console

====QuickSort====
a_average: 40.500000
b: 39.000000
c: 12.000000
d: 12.000000
====MergeSort====
a_average: 51.800000
b: 51.000000
c: 26.000000
d: 38.000000
====HeapSort====
a_average: 38.700000
b: 32.000000
c: 28.000000
d: 23.000000
done!
D:\OneDrive - 아주대학교\아주대학교\아주대 1학년\2학기\자료구조\assignment\assignment 6\Project1\wx64\Debug\Project1.exe
(process 34248) exited with code 0.
Press any key to close this window . . .
```

```
Microsoft Visual Studio Debug Console

====QuickSort====
a_average: 37.200000
b: 20.000000
c: 7.000000
d: 7.000000
====MergeSort====
a_average: 32.300000
b: 36.000000
c: 25.000000
d: 27.000000
====HeapSort====
a_average: 40.900000
b: 37.000000
c: 20.000000
d: 18.000000
done!
D:\OneDrive - 아주대학교\아주대학교\아주대 1학년\2학기\자료구조\assignment\assignment 6\Project1\wx64\Debug\Project1.exe
(process 32464) exited with code 0.
Press any key to close this window . . .
```

```
Microsoft Visual Studio Debug Console

====QuickSort====
a_average: 32.400000
b: 26.000000
c: 9.000000
d: 11.000000
====MergeSort====
a_average: 32.400000
b: 30.000000
c: 16.000000
d: 17.000000
====HeapSort====
a_average: 32.700000
b: 29.000000
c: 77.000000
d: 52.000000
done!
D:\OneDrive - 아주대학교\아주대학교\아주대 1학년\2학기\자료구조\assignment\assignment 6\Project1\wx64\Debug\Project1.exe
(process 35288) exited with code 0.
Press any key to close this window . . .
```

반복된 프로그램의 실행을 보았을 때 각 실행마다 같은 정렬 알고리즘이더라도 다른 실행결과를 보여주었다. 이는 정렬할 데이터의 미리 정렬되어 있는 순서에 따라 차이가 나는 것임을 알 수 있는 결과이다. 첫번째 실험결과의 a어레이의 평균시간을 보면, heap정렬이 다른 두개의 정렬에 비해 월등한 성능을 보여주는 것처럼 보인다. 이는 heap정렬이 상황에 따라서 다른 두개의 정렬보다 더 빠르게 동작 할 수 있다는 것을 보여주는 결과이다. 하지만 이미 정렬되어 있는 상태의 어레이 c, d에 대해서는 quick 정렬과 merge정렬이 heap정렬에 비해서 더 좋은 성능을 보여주고 있는 것을 알 수 있다. quick정렬의 경우 이미 정렬되어 있는 상태라면 별다른 연산 없이 지나갈 수 있고, merge 정렬의 경우 비슷한 이유를 가지고 있기 때문이다. 하지만 heap정렬의 경우 이미 정렬되어 있는 결과와 상관없이 매 정렬마다 MAX Heap을 만들어서 정렬을 진행하기 때문에 MAX Heap을 만드는 과정에서 많은 시간이 소요될 것이다. 따라서 이러한 결과가 나온 것이다.