



ICE3020 알고리즘 HW #1

제 목

정렬 알고리즘을 구현하고 성능 비교 분석

보고서 작성 서약서

1. 나는 타학생의 보고서를 베끼거나 여러 보고서의 내용을 짜집기하지 않겠습니다.
2. 나는 보고서의 주요 내용을 인터넷사이트 등을 통해 얻지 않겠습니다.
3. 나는 보고서의 내용을 조작하지 않겠습니다.
4. 나는 보고서 작성에 참고한 문헌의 출처를 밝히겠습니다.
5. 나는 나의 보고서를 제출 전에 타학생에게 보여주지 않겠습니다.

나는 보고서 작성시 윤리에 어긋난 행동을 하지 않고 정보통신공학인으로서 나의 명예를 지킬 것을 맹세합니다.

2020년 5월 3일

학부 정보통신공학과

학년 3학년

성명 김소원

학번 12181748

1. 개요

* 아래 네 가지 정렬 알고리즘을 구현하고 성능을 비교 분석하라.

- (1) Selection sort
- (2) Median-of-three Quick sort
- (3) Shell Sort
- (4) Bitonic sort
- (5) Odd-Even Merge sort

2. 상세 설계내용

데이터 셋 크기를 바꿔주어야 하므로 결과값을 볼 때마다 `#define MAX_SIZE` 뒤에 오는 숫자의 크기를 5000, 10000, 50000, 100000....등과 같이 늘려보았다. 그리고 실행 시간을 측정하는데 필요한 함수와 각각의 sort 함수 다섯가지를 작성하였다.

(1) Selection sort

Selection sort는 버블 정렬을 일부 개선한 알고리즘으로 정렬 순서가 맞지 않으면 무조건 자리를 바꿔주었던 버블정렬과 달리, 한번 반복할 때마다 최소값 또는 최댓값을 찾고 요소 위치를 바꾸는 정렬이다.

나는 이때 코드를 최소값을 찾는 코드로 작성하였는데 n개의 데이터가 있다고 가정 할 때 가장 작은 값을 찾으면 그 값이 최소값이 되어 원래 배열의 자리와 교환하는 방법으로 코드를 작성하였다.

(2) Median-of-three Quick sort

Quick sort는 무작위로 pivot을 하나 선택해서 pivot보다 작은쪽/큰 쪽으로 나누어 분할하는 정렬이다. 그러나 Median-of-three Quick sort는 그냥 Quick sort보다 더 자세하게 중간값을 찾아 이것을 pivot으로 결정하는 것이다.

먼저 Median of three quick sort 코드를 짜는데 필요한 교환함수 `sp`를 만들었다. `medianQuickSort` 함수에서는 첫번째 값, 가운데 값, 마지막 값을 우선 정렬한 후 `pivot=arr[mid];` 로 pivot을 배열의 중간값으로 설정하였다. 배열 중간값이 첫번째 값보다 작을 때 중간값과 첫번째 교환해주었으며(즉 중간값이 더 작으니까 앞쪽으로 간다는 말이다), 배열 마지막 값이 중간값보다 작을 때에는 마지막값과 중간값 교환, 즉 이때는 마지막 값이 더 작으므로 앞쪽 배열로 간다는 뜻이다. 마지막으로 배열 중간값이 처음값보다 작을 때 역시 중간값과 처음값을 교환하여 처음값이 더 앞쪽 배열에 위치하도록 하였다.

(3) Shell Sort

Shell sort는 삽입 정렬을 보완한 알고리즘으로 'Donald L.Shell'이라는 사람이 만든 정렬이다. 셸 정렬은 간격(gap)을 만들어 그 gap을 줄여나가는 정렬인데 예를 들어 배열이 8자리 배열이라면 처음 초기 간격은 $8 \times 1/2 = 4$ 가 되어 4자리씩 비교를 하여 정렬한다. 그 다음 간격은 $4 \times 1/2 = 2$ 가 되어 2자리씩 비교하고 그 다음 간격은 $2 \times 1/2 = 1$ 이 되는 식으로 정렬하는 것이다. 이렇게 간격이 1이 될때까지 계속 반복해주는 식으로 코드를 만들었다.

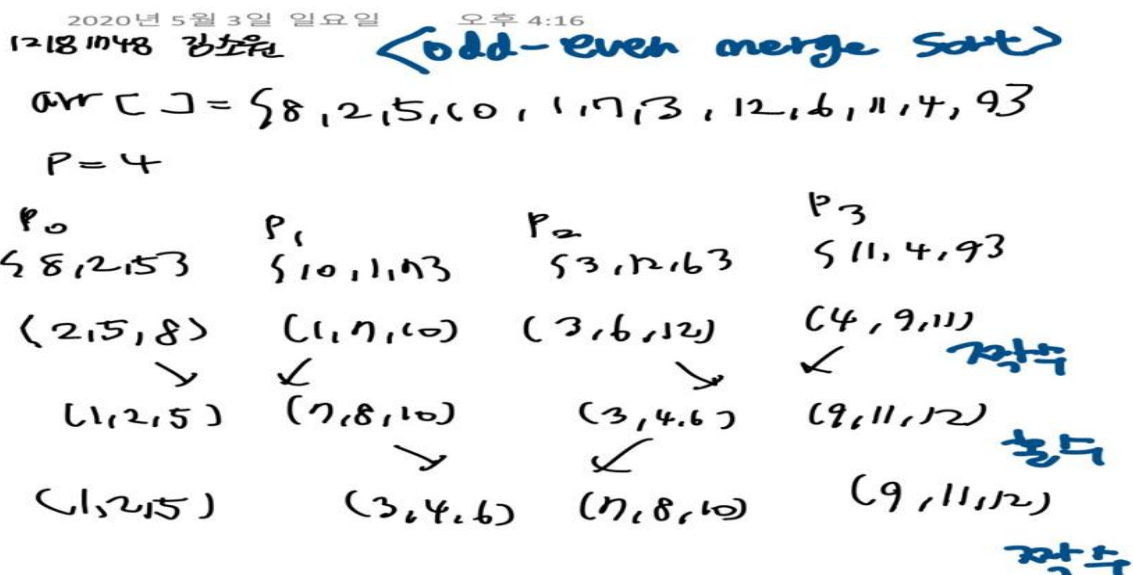
그 다음 Shell sort의 부분 집합으로 사용할 insertion 함수를 만들었는데, 이 함수 속에서 i가 for문을 돌 때 first+gap부터 시작하는 이유는 배열의 제일 처음 원소는 정렬할 필요가 없이 그 다음 값부터 비교하기 때문이다. Shell Sort에서는 위에서 말한거와 같이 gap을 선언해주고 그 gap을 반씩 줄여나갔으며 $gap \% 2 \neq 0$ 일 때 gap의 값을 1 증가시킨 이유로는 gap이 홀수일 때가 더 효율이 좋기 때문이다.

(4) Bitonic sort

바이토닉 정렬은 batcher가 만든 정렬로 배열이 오름차순이었다 내림차순이 반복되는 정렬로 예를 들어 8자리 배열이라고 가정하면 배열을 앞에서부터 2개씩 비교하였을 때 1,2번째 자리에선 오름차순이어야 하므로 작은 값이 더 왼쪽으로(첫번째 자리로), 그 다음 3,4번째 자리에선 내림차순이어야 하므로 큰 값이 더 왼쪽으로(3번째 자리로), 5,6번째 자리에선 오름차순이어야 하므로 작은 값이 더 왼쪽(5번째 자리로), 마지막으로 7,8번째 자리에선 내림차순이어야 하므로 더 큰 값이 왼쪽(7번째 자리로) 가야한다.

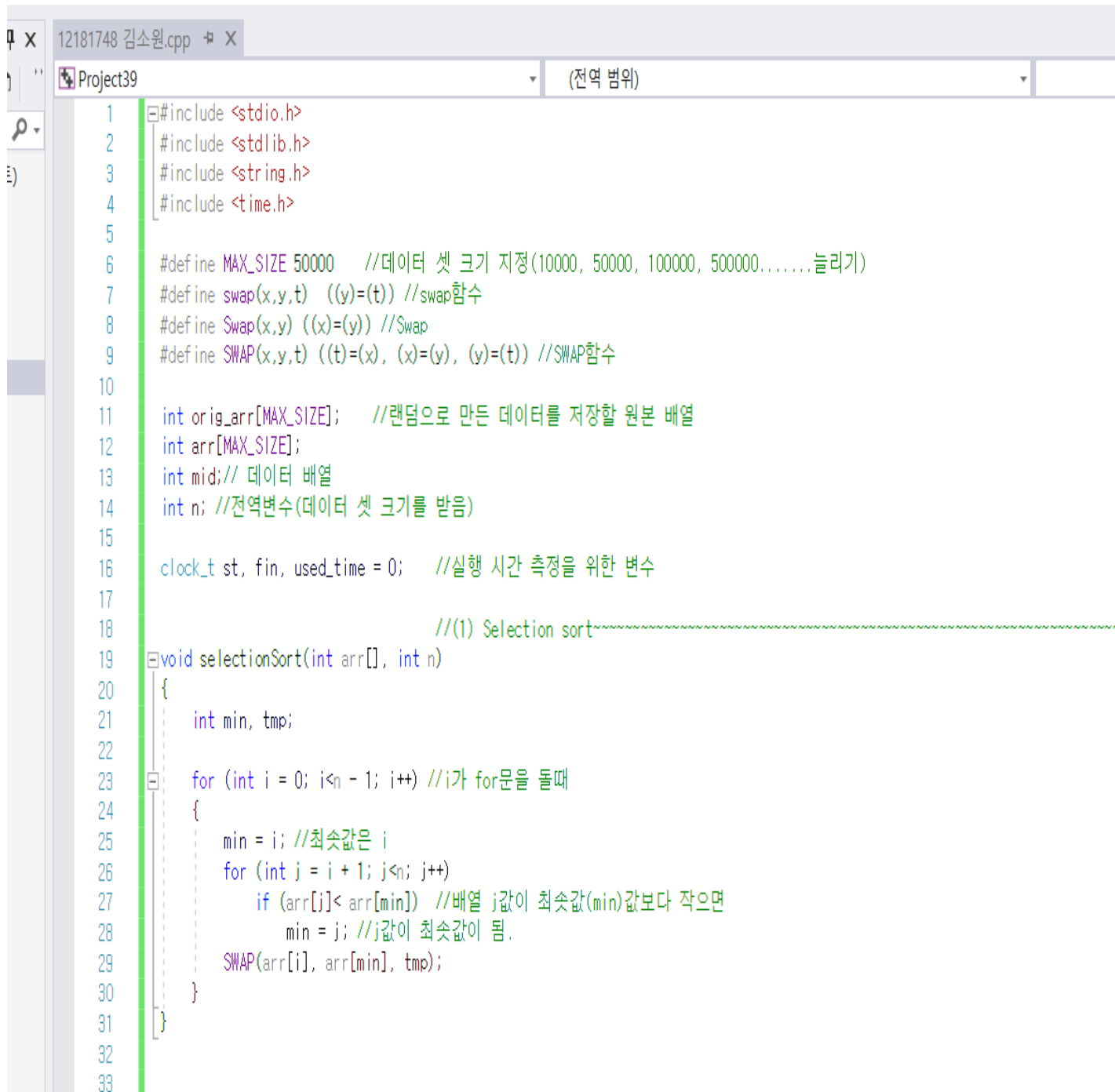
Bitonic sort 함수에서 down은 작은 값을 말하며 위에서 말한것처럼 오름차순일 경우는 작은값이 더 왼쪽으로, 내림차순일 경우는 큰 값이 더 왼쪽으로 가기 위한 Swap함수를 작성하였다.

(5) Odd-Even Merge sort



마지막으로 odd even merge sort는 글로만 표현하는데 어려워 그림으로 먼저 나타내보았다. 이 정렬은 짝-홀수 병합 정렬로 짝수반복과 홀수 반복과정을 거치는 것이다. 예를 들어 배열이 12자리로 되어있다 가정하면 먼저 4등분하여 앞에서부터 각각 3개의 숫자들을 갖는다. 그 안에서 작은 수부터 차례대로 정렬한 후 위에 그림에 나와있는 거와 같이 p0,p1배열끼리 합병하여 또 정렬, p2,p3끼리 합병하여 정렬하여 3개의 숫자씩 나타낸 후 가운데 두개의 배열을 합병하면 작은 수부터 차례대로 정렬이 된다. 이 과정에서 짝수번, 홀수번이 반복되는 것이며 oddEvenMergeSort함수에서 dis는 비교할 거리를 나타낸것이다.

3. 실행 화면



```

12181748 김소원.cpp
Project39 (전역 범위)
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5
6  #define MAX_SIZE 50000 //데이터 셋 크기 지정(10000, 50000, 100000, 500000.....늘리기)
7  #define swap(x,y,t) ((y)=(t)) //swap함수
8  #define Swap(x,y) ((x)=(y)) //Swap
9  #define SWAP(x,y,t) ((t)=(x), (x)=(y), (y)=(t)) //SWAP함수
10
11 int orig_arr[MAX_SIZE]; //랜덤으로 만든 데이터를 저장할 원본 배열
12 int arr[MAX_SIZE];
13 int mid; // 데이터 배열
14 int n; //전역변수(데이터 셋 크기를 받음)
15
16 clock_t st, fin, used_time = 0; //실행 시간 측정을 위한 변수
17
18 // (1) Selection sort ~~~~~
19 void selectionSort(int arr[], int n)
20 {
21     int min, tmp;
22
23     for (int i = 0; i < n - 1; i++) // i가 for문을 돌때
24     {
25         min = i; //최솟값은 i
26         for (int j = i + 1; j < n; j++)
27             if (arr[j] < arr[min]) //배열 j값이 최솟값(min)값보다 작으면
28                 min = j; //j값이 최솟값이 됨.
29         SWAP(arr[i], arr[min], tmp);
30     }
31 }
32
33

```

```

34 // (2) Median-of-three Quick sort
35 void sp(int arr[], int a, int b) { //교환
36
37     int temp = arr[a];
38     arr[a] = arr[b];
39     arr[b] = temp;
40
41 }
42 void medianQuickSort(int arr[], int first, int mid, int last) {
43
44     if (arr[mid] < arr[first])
45         sp(arr, mid, first); //배열 중간값이 첫번째 값보다 작을 때 중간값과 첫번째 교환
46
47     if (arr[last] < arr[mid])
48         sp(arr, last, mid); //배열 마지막 값이 중간값보다 작을 때 마지막값과 중간값 교환
49
50     if (arr[mid] < arr[first])
51         sp(arr, mid, first); //배열 중간값이 처음값보다 작을 때 중간값과 처음값 교환
52
53 }
54
55 void medianQuickSort(int arr[], int first, int last) {
56
57     int i, j;
58     int pivot;
59     mid = (first + last) / 2; //중간값
60
61     medianQuickSort(arr, first, mid, last); //처음, 중간, 마지막 값 우선정렬
62
63     if (last - first + 1 > 3) { //median of quick sort는 젤 처음 3개의 값으로 정렬을 시작하기 때문에 3보다 커야한다.
64         pivot = arr[mid]; //피벗을 배열의 중간값으로 설정
65         sp(arr, mid, last - 1);
66         i = first;
67         j = last - 1; //제일 마지막은 이미 정렬되어있으므로 마지막 바로 앞에 값으로 설정
68
69         while (true) {
70             while (arr[++i] < pivot && i < last); //배열의 i가 증가할때 피벗보다 작으면서 last보다 작을동안
71             while (arr[--j] > pivot && first < j); //배열의 j가 감소할때 피벗보다 크면서 first보다 클동안
72             if (i >= j)
73                 break;
74             sp(arr, i, j); //
75         }
76         sp(arr, i, last - 1);
77         medianQuickSort(arr, first, i - 1); //앞부분 배열(처음부터 i-1까지)
78         medianQuickSort(arr, i + 1, last); //뒷부분 배열(i+1부터 마지막값까지)
79     }
80 }
81
82
83 // (3) Shell Sort
84 void insertion(int arr[], int first, int last, int gap) //shellSort의 부분집합에 사용되는 삽입 함수
85 {
86     int i, j, k;
87
88     for (i = first + gap; i <= last; i = i + gap) // 첫 번째 원소는 정렬 할 필요가 없고 그 다음 배열 값을 k 값으로 설정. 따라 초기값은 first + gap
89     {
90         k = arr[i]; // k값은 배열 i가 된다
91
92         for (j = i - gap; j >= first && k < arr[j]; j = j - gap) //gap 감소
93             arr[j] = arr[j + gap]; //배열 j값이 j+gap 값이 됨.
94         arr[j + gap] = k; //k값이 j+gap값이 됨.
95     }
96 }
97 void shellSort(int arr[], int n)
98 {
99     int i, gap;
100

```

```
100
101     for (gap = n / 2; gap > 0; gap = gap / 2) //gap값을 반씩 줄여나가야 함
102     {
103         if ((gap % 2) == 0) //gap이 짝수일 경우 홀수로 만들어야함
104             gap++;
105
106         for (i = 0; i < gap; i++) //부분리스크 삽입 정렬
107             insertion(arr, i, n - 1, gap);
108     }
109 }
110
111 // (4) Bitonic sort ~~~~~~
112 void bitonicSwap(int arr[], int i, int j, int seq)
113 {
114     if (seq == (arr[i] > arr[j])) //오름차순 정렬
115         Swap(arr[i], arr[j]); //배열 i, j 교환
116 }
117
118 void bitonic(int arr[], int down, int count, int seq)
119 {
120     if (count > 1) //정렬 갯수
121     {
122         int t = count / 2;
123         for (int i = down; i < down + t; i++) //i가 작은값(down)부터 down+t까지 for문을 돌때
124             bitonicSwap(arr, i, i + t, seq); //i와 i+t 교환
125
126         bitonic(arr, down, t, seq); //작은 값->큰값(오름차순)
127         bitonic(arr, down + t, t, seq); //큰 값에서->작은값(내림차순)
128     }
129 }
130
131
132 void bitonicSort(int arr[], int down, int count, int seq)
133 {
134     if (count > 1)
135     {
```

```

133     {
134         if (count>1)
135         {
136             int t = count / 2;
137
138             bitonicSort(arr, down, t, 1); //오름차순 정렬
139
140             bitonicSort(arr, down + t, t, 0); //내림차순 정렬
141
142             bitonic(arr, down, count, seq); //오름차순정렬
143         }
144     }
145
146 void sort(int arr[], int n, int up)
147 {
148     bitonicSort(arr, 0, n, up); //전체 배열 정렬
149 }
150
151
152
153 // (5) Odd-Even Merge sort ~~~~~~
154 void oddEvenMerge1(int i, int j)
155 {
156     if (arr[i] > arr[j]) //배열 i가 j보다 클때
157         Swap(i, j); //i,j 교환
158 }
159 void oddEvenMerge(int arr, int n, int dis) { //dis=비교할 거리
160     {
161         int t = dis * 2;
162         if (t < n)
163         {
164             oddEvenMerge(arr, n, t); // 짝수 반복
165             oddEvenMerge(arr + dis, n, t); // 홀수 반복
166
167             for (int i = arr + dis; i + dis < arr + n; i += t)
168                 oddEvenMerge1(i, i + dis); //배열 i가 i+dis보다 크면 교환
169         }

```

```
169     }
170     else
171         oddEvenMerge1(arr, arr + dis); //t가 n보다 작으면 arr, arr+dis 교환
172
173
174     oddEvenMerge1(arr, arr + dis); //arr가 arr+dis보다 크면 arr, arr + dis 서로 교환
175 }
176 }
177 void oddEvenMergeSort(int arr[], int n)
178 {
179     if (n>1)
180     {
181         int t = n / 2;
182         oddEvenMergeSort(arr, t);
183         oddEvenMergeSort(arr + t, t);
184         oddEvenMerge(*arr, n, 1);
185     }
186 }
187
188 //원본 배열을 복사하는 함수
189 void CopyArr(void)
190 {
191     int i;
192     for (i = 0; i<n; i++) //i가 for문을 돌때
193         arr[i] = orig_arr[i]; //원본배열을 배열 i로
194 }
195
196 //실행 시간 측정 및 출력 함수
197 void CalcTime(void)
198 {
199     used_time = fin - st;
200     printf("\n실행 시간 : %f sec\n", (float)used_time / CLOCKS_PER_SEC);
201 }
202
203
```



```
202
203
204 void main() //메인함수
205 {
206     int i;
207
208     n = MAX_SIZE;
209     for (i = 0; i < n; i++)
210         orig_arr[i] = rand(); //랜덤으로 설정
211
212     printf("데이터 셋 크기 : %d\n\n", n);
213
214     CopyArr();
215     st = clock();
216     printf("(1) Selection sort"); //선택정렬 시간재기
217     selectionSort(arr, n);
218     fin = clock();
219     CalcTime();
220
221     CopyArr();
222     st = clock();
223     printf("(2) Median-of-three Quick sort"); //중간값 정렬 시간 재기
224     medianQuickSort(arr, 0, n);
225     fin = clock();
226     CalcTime();
227
228     CopyArr();
229     st = clock();
230     printf("(3) Shell Sort"); //셸 정렬 시간 재기
231     shellSort(arr, n);
232     fin = clock();
233     CalcTime();
234
235     CopyArr();
236     st = clock();
237     printf("(4) Bitonic sort"); //바이오틱 정렬 시간 재기
238     bitonicSort(arr, 0, n, 0);
239     fin = clock();
240     CalcTime();
```

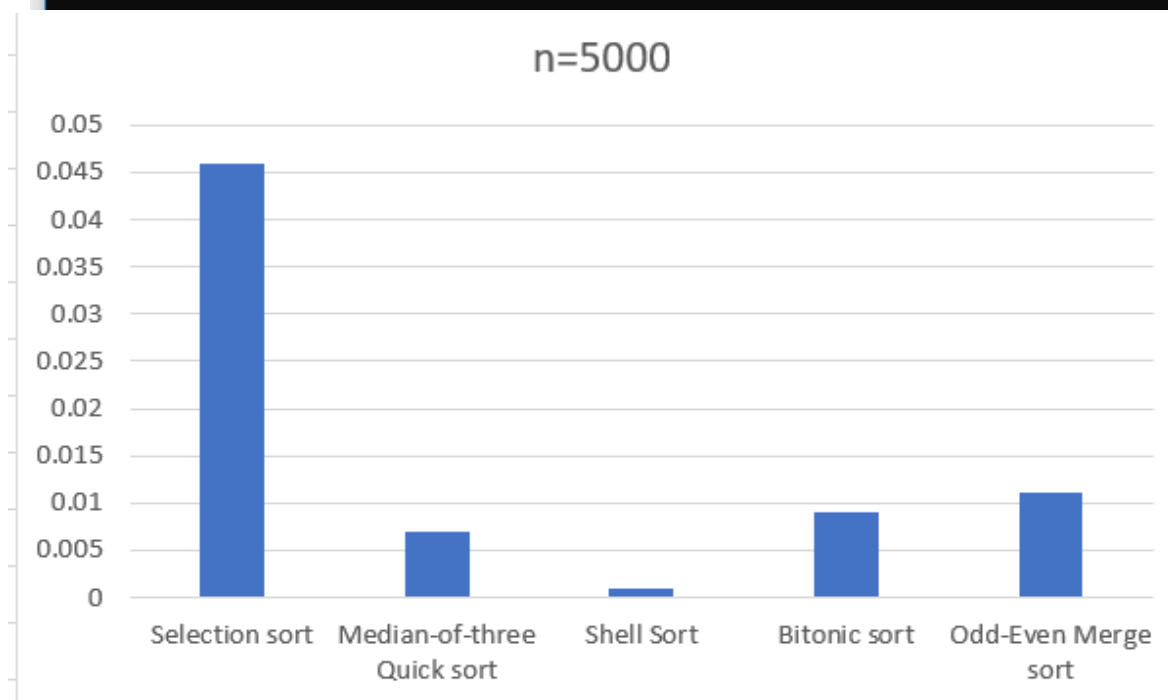
```
240     CalcTime();
241
242     CopyArr();
243     st = clock();
244     printf("(5) Odd-Even Merge sort"); //짝홀 합병 정렬 시간 재기
245     oddEvenMergeSort(arr, n);
246     fin = clock();
247     CalcTime();
248 }
```

4. 분석 및 결론

N에 대한 성능 비교 그래프는 엑셀로 나타내보았다. 세로축이 실행 시간을 나타낸 것이며 단위는 sec이다

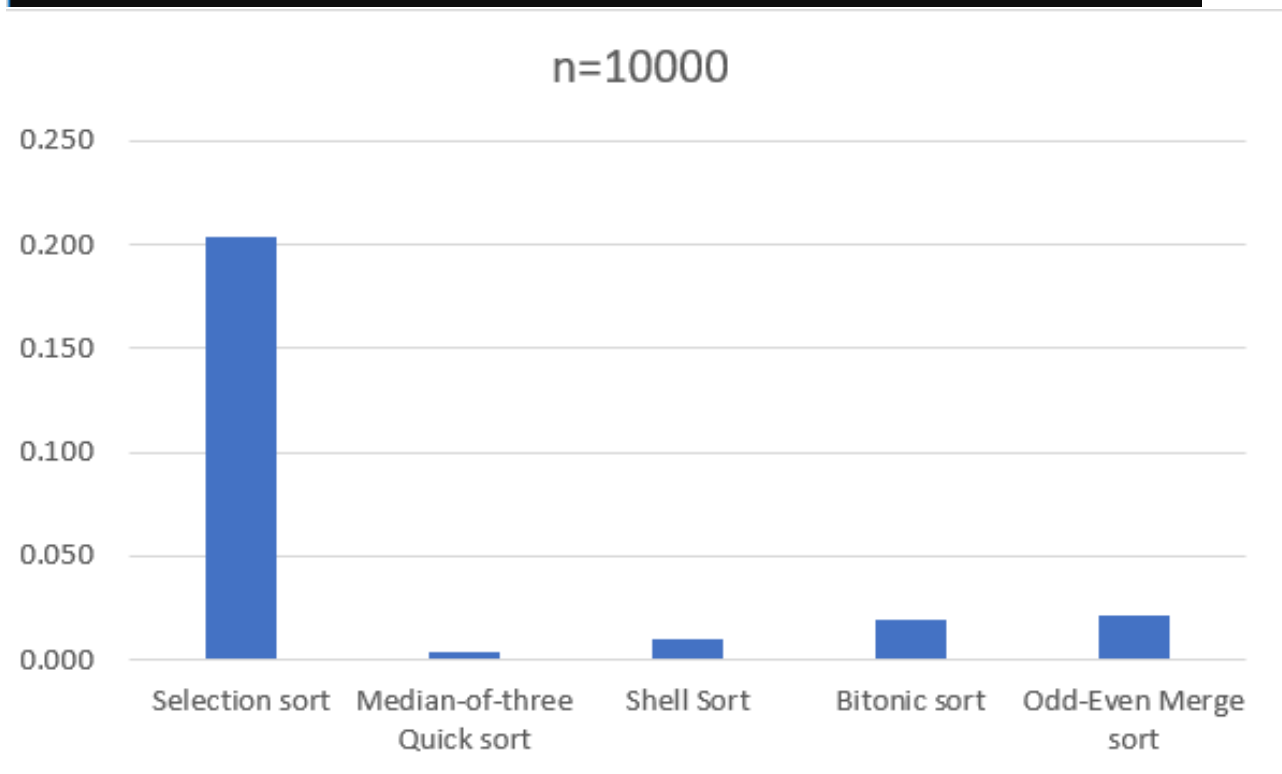
<데이터 셋의 크기가 5000일 때 결과 및 그래프>

```
C:\WINDOWS\system32\cmd.exe
데이터 셋 크기 : 5000
(1) Selection sort
실행 시간 : 0.046000 sec
(2) Median-of-three Quick sort
실행 시간 : 0.007000 sec
(3) Shell Sort
실행 시간 : 0.001000 sec
(4) Bitonic sort
실행 시간 : 0.009000 sec
(5) Odd-Even Merge sort
실행 시간 : 0.011000 sec
계속하려면 아무 키나 누르십시오 . . .
```



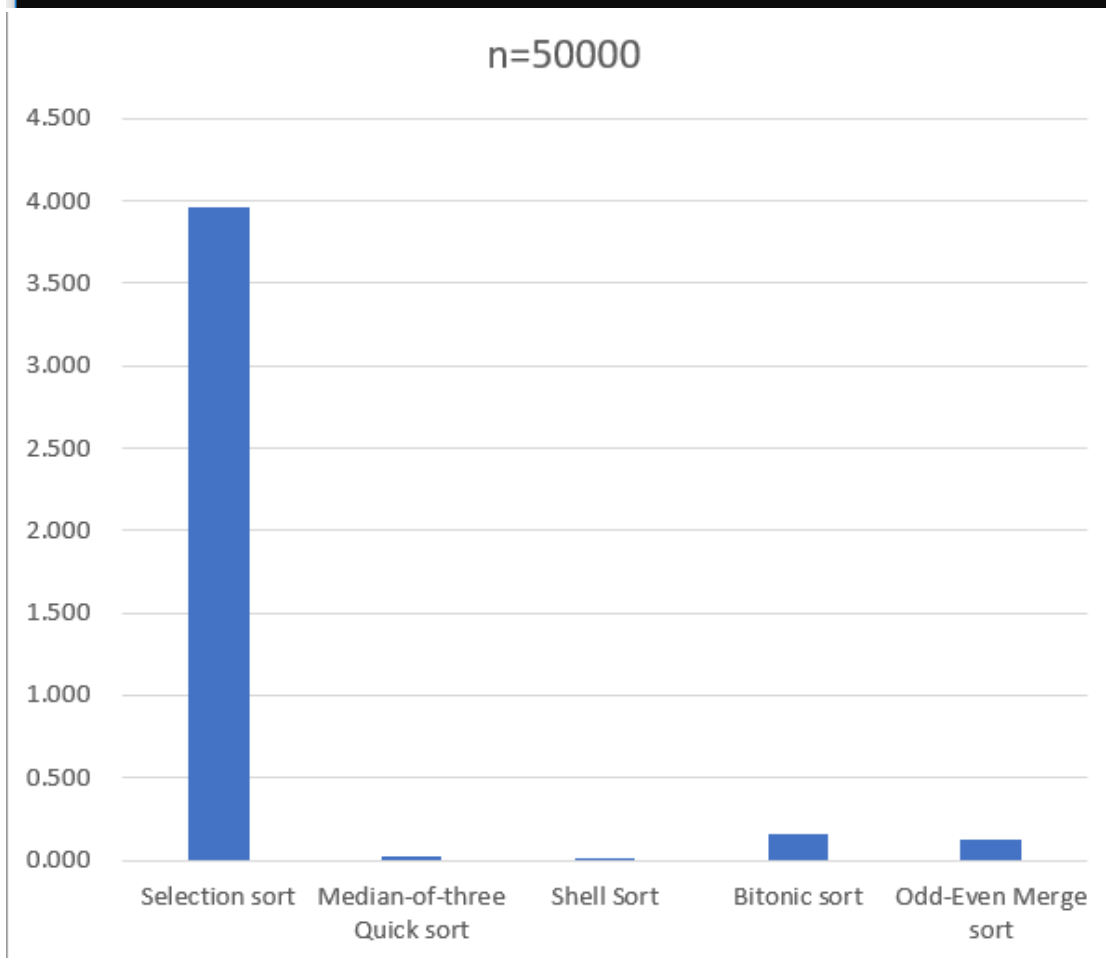
<데이터 셋의 크기가 10000일 때 결과 및 그래프>

```
C:\WINDOWS\system32\cmd.exe
데이터 셋 크기 : 10000
(1) Selection sort
실행 시간 : 0.204000 sec
(2) Median-of-three Quick sort
실행 시간 : 0.004000 sec
(3) Shell Sort
실행 시간 : 0.001000 sec
4) Bitonic sort
실행 시간 : 0.019000 sec
(5) Odd-Even Merge sort
실행 시간 : 0.021000 sec
계속하려면 아무 키나 누르십시오 . . .
```



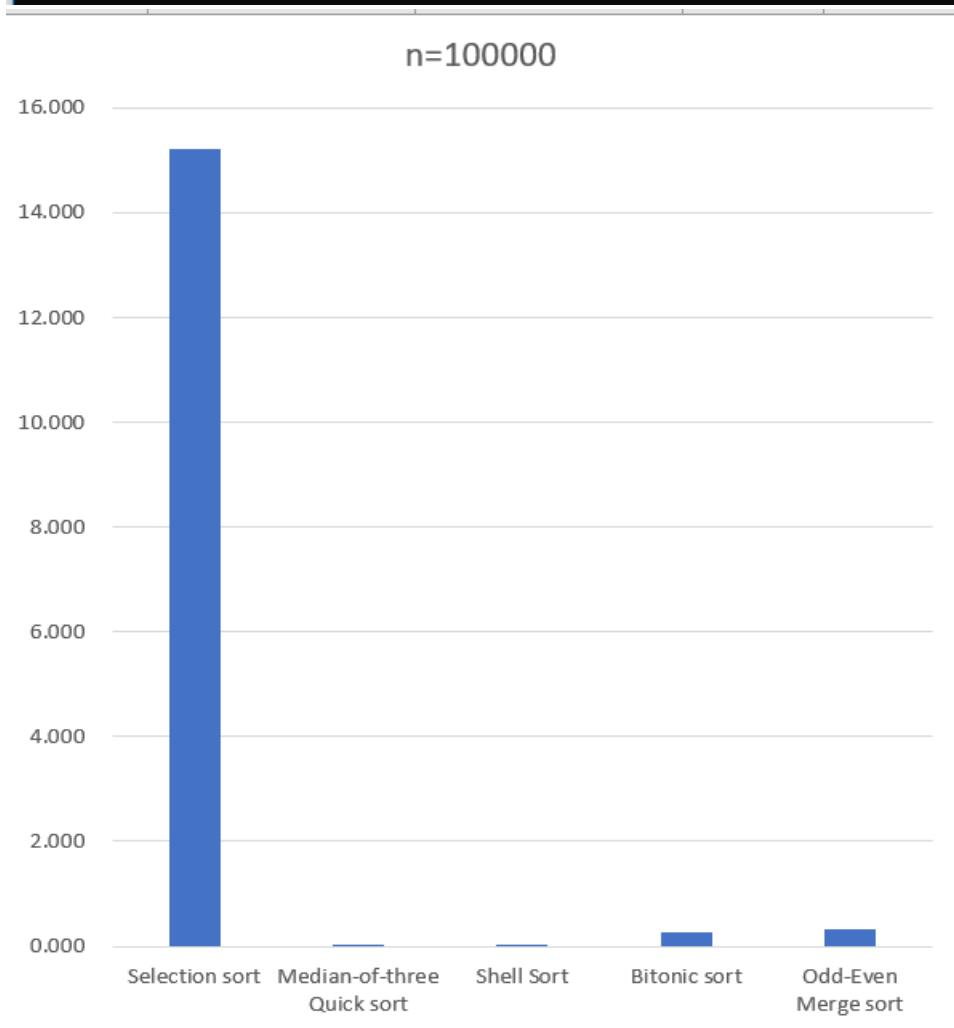
<데이터 셋의 크기가 50000일 때 결과 및 그래프>

```
C:\WINDOWS\system32\cmd.exe
데이터 셋 크기 : 50000
(1) Selection sort
실행 시간 : 3.960000 sec
(2) Median-of-three Quick sort
실행 시간 : 0.019000 sec
(3) Shell Sort
실행 시간 : 0.014000 sec
4) Bitonic sort
실행 시간 : 0.153000 sec
(5) Odd-Even Merge sort
실행 시간 : 0.129000 sec
계속하려면 아무 키나 누르십시오 . . .
```



<데이터 셋의 크기가 100000일 때 결과 및 그래프>

```
C:\WINDOWS\system32\cmd.exe
데이터 셋 크기 : 100000
(1) Selection sort
실행 시간 : 15.205000 sec
(2) Median-of-three Quick sort
실행 시간 : 0.031000 sec
(3) Shell Sort
실행 시간 : 0.020000 sec
4) Bitonic sort
실행 시간 : 0.279000 sec
(5) Odd-Even Merge sort
실행 시간 : 0.319000 sec
계속하려면 아무 키나 누르십시오 . . .
```



<데이터 셋의 크기가 150000일 때 결과 및 그래프>

```
C:\WINDOWS\system32\cmd.exe
데이터 셋 크기 : 150000

(1) Selection sort
실행 시간 : 37.889999 sec

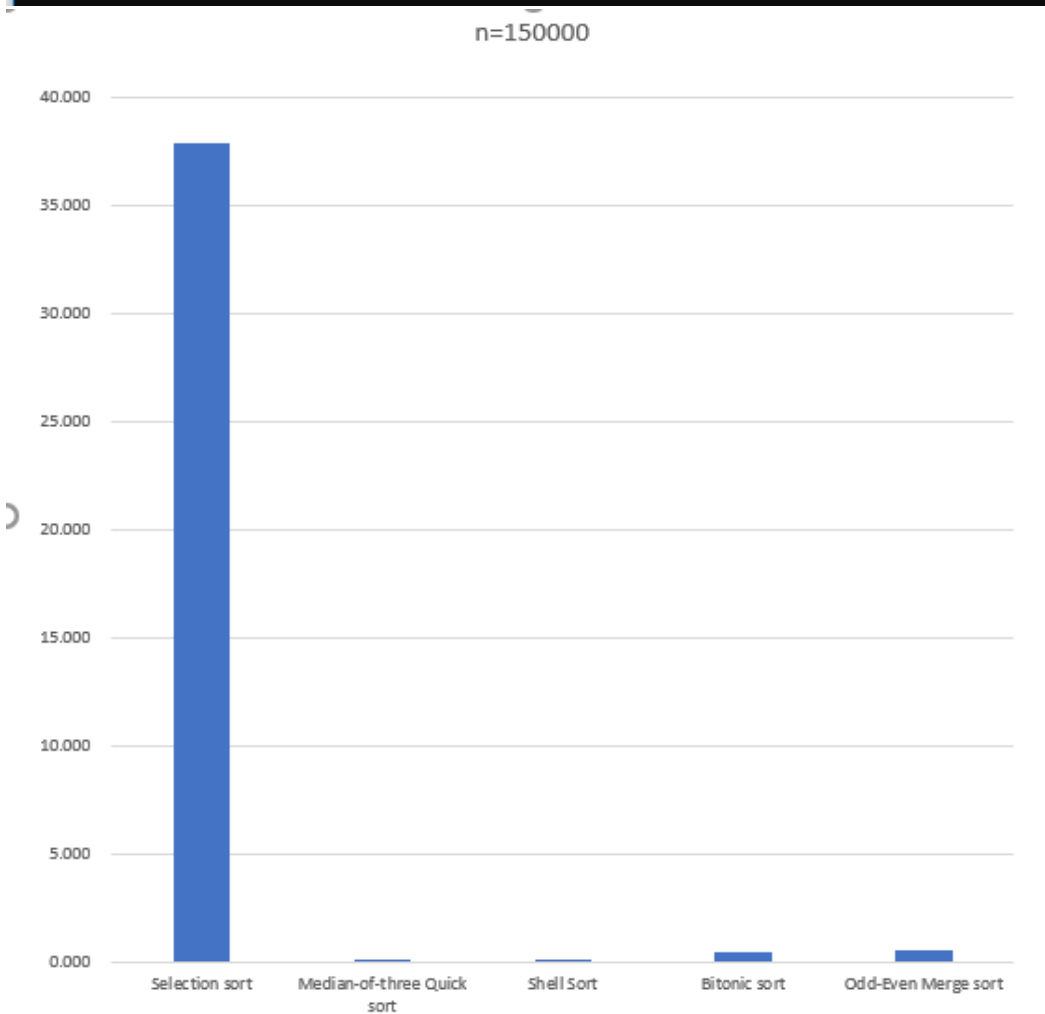
(2) Median-of-three Quick sort
실행 시간 : 0.067000 sec

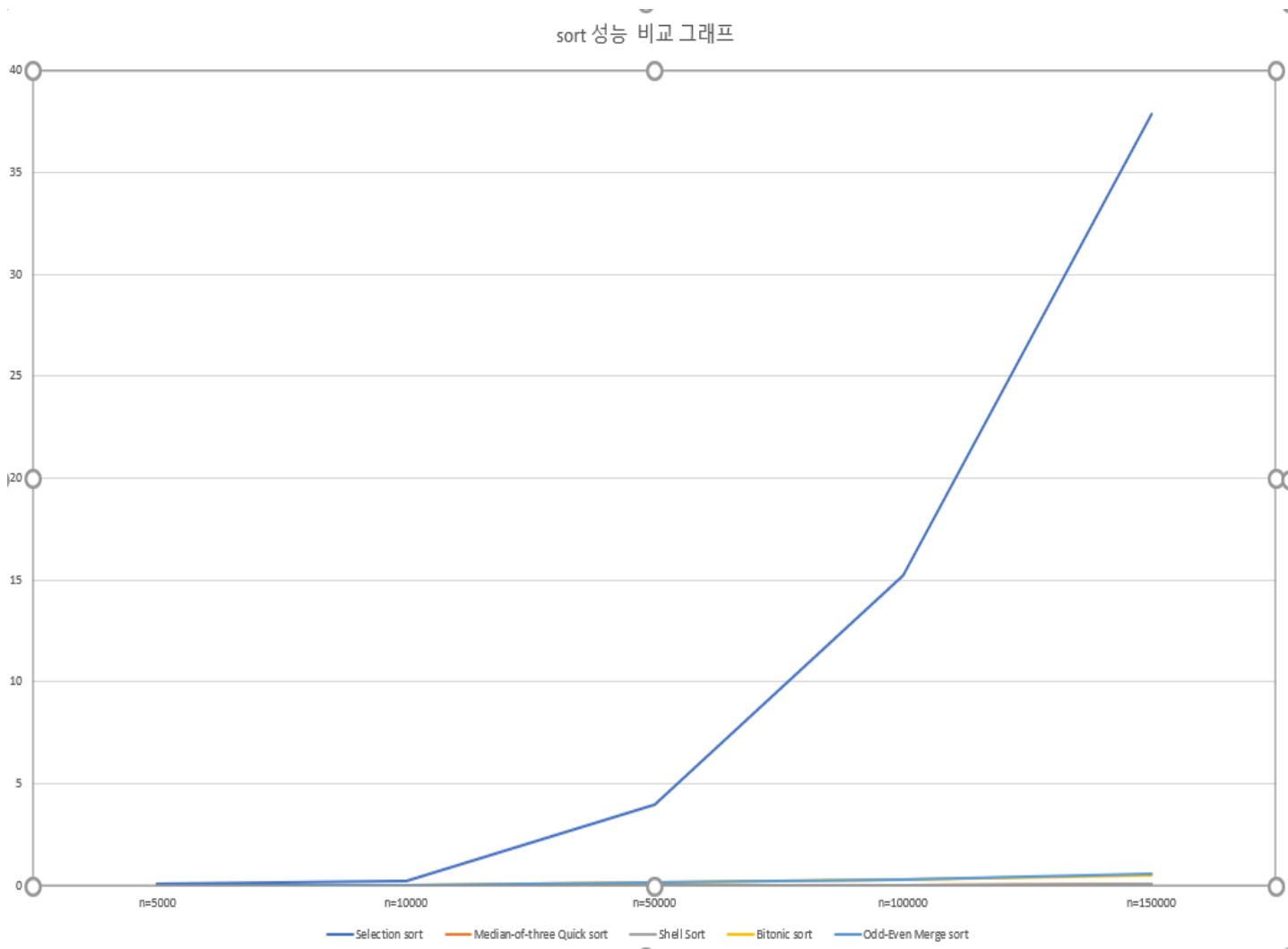
(3) Shell Sort
실행 시간 : 0.049000 sec

(4) Bitonic sort
실행 시간 : 0.469000 sec

(5) Odd-Even Merge sort
실행 시간 : 0.540000 sec

계속하려면 아무 키나 누르십시오 . . .
```





마지막으로 n 이 점점 증가할 때 각각의 sort들의 실행시간을 한번에 비교해서 보여주는 그래프도 만들어보았는데 이 그래프를 통해 Selection sort의 실행시간이 가장 크게 증가한다는 것을 알 수 있었다. (세로축은 시간/ 단위:sec)

Selection sort는 다른 sort에 비하여 구현이 매우 쉬웠다. 또한 selection sort는 실제로 교환하는 횟수가 적기 때문에 많은 교환이 효율적으로 사용될 수 있다는 것을 알게 되었다.

Selection sort의 단점으로는 N 을 랜덤으로 설정하여 숫자를 늘릴 때마다 selection sort의 실행시간이 다른 정렬들에 비해 항상 오래 걸렸는데 실제로 selection sort의 시간 복잡도는 $O(N^2)$ 으로 다른 정렬에 비해 시간이 오래걸리는 정렬이다.

두번째로 Median of Quick Sort는 실행시간이 짧게 걸렸다. Quick sort는 평균적으로 $O(n \log n)$ 의 시간복잡도를 가지고 최악의 경우에는 selection sort와 마찬가지로 $O(N^2)$ 의 시간복잡도를 가지게 되지만 Median of Quick Sort는 pivot을 최악으로 잡아주는 경우는 없기 때문에 $O(N^2)$ 의 시간복잡도를 가질 걱정은 할 필요가 없다. 따라 실행시간이 길게 걸리지 않았다.

Shell sort는 큰 거리가 이동가능함으로 삽입 정렬보다 최종 자리에 더 빨리 찾아갈 수 있어 연산 횟수를 줄일 수 있다는 장점이 있다. 또한 한번에 정렬하는 것이 아니라 부분리스트를 만들어 정렬된상태로 조금씩 만들어나가므로 실행시간이 보다 더 빠르게 수행될 수 있다.

단점이 있다면 간격이 일정하지 않고 잘못 설정한다면 효율이 떨어지고 거리를 많이 이동할 경우 반복되는 비교연산이 많이 일어난다는 것이다.

Shell sort의 최선 시간 복잡도는 $O(n)$ 이고 평균은 $O(n^{1.25})$, 최악일 경우엔 $O(N^2)$ 의 시간복잡도를 갖는 불안정 정렬이다.

Bitonic sort는 시간복잡도가 $O(\log^2 n)$ 으로 빠른 편에 속한다. 실행시간도 굉장히 빠르게 나왔다. 또한 Bitonic sort는 병렬화가 무척 쉽다는 장점이 있으며 GPU, 멀티코어에서도 사용 가능하다. 바이토닉 정렬은 Sorter나 Sorting Network라는 표현으로도 불린다. 그 이유는 Quick Sort처럼 데이터에 따라 pivot이 달라지거나 혹은 어떤 배열에서 몇 번째 데이터끼리 서로 비교할지 바뀌는게 아니라 데이터에 상관없이 항상 비교할 데이터의 위치가 이미 정해져 있기 때문이다.

마지막으로 Odd-even merge sort는 시간복잡도가 $O(\log^2 n)$ 이며 n 에 따른 실행시간을 보았을 때 빠른편에 속했다. Merge sort보다 odd even merge sort가 최적의 평균 실행 시간을 가지며 퀵정렬과 달리 pivot을 따로 설정할 필요없이 무조건 반으로 분할하기 때문에 pivot의 값이 영향을 주는 경우는 없다는 장점이 있다.

단점이 있다면 merge sort는 임시배열에 원본을 계속해서 옮겨주는 정렬이므로 메모리를 할당할 수 없는 경우엔 사용 할 수 없는 정렬이다.

정렬 알고리즘의 성능을 분석하고 비교해보면서 각각의 정렬이 어떠한 방법으로 사용되는지와 각 정렬의 시간 복잡도와 장단점, 걸리는 실행시간을 자세히 알게 되었다. 또한 필요한 상황에 맞게 알맞은 정렬 알고리즘을 사용하여 실행시간을 단축시키는 것이 중요하다 깨달았으며 보다 더 효율적인 결과를 내도록 알고리즘을 구현시킬 것이다.

5. 참고문헌 및 참고자료

-네이버 블로그 https://blog.iwanhae.ga/bitonic-sort_batchers-oddeven-mergesort/

-유튜브 Odd-Even Merge Sort | Parallel Algorithm | Sorting Networks

-유튜브 Quick Sort 퀵정렬 알고리즘 설명

-누구나 자료 구조와 알고리즘: 상식으로 이해하는 자료 구조와 알고리즘! 6장 (공)저: 제이 웬그로우

-네이버블로그

<https://codingdog.tistory.com/entry/%EB%B0%94%EC%9D%B4%ED%86%A0%EB%8B%89-%EC%A0%95%EB%A0%AC-%EC%97%87%EA%B0%88%EB%A0%A4-%EA%B0%80%EB%A9%B4%EC%84%9C-sorting-%ED%95%9C%EB%8B%A4>

- Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching

(공)저: Robert Sedgewick