

알고리즘 설계 HW #1

정렬 알고리즘 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 3

성명 심규환

학번 12181793

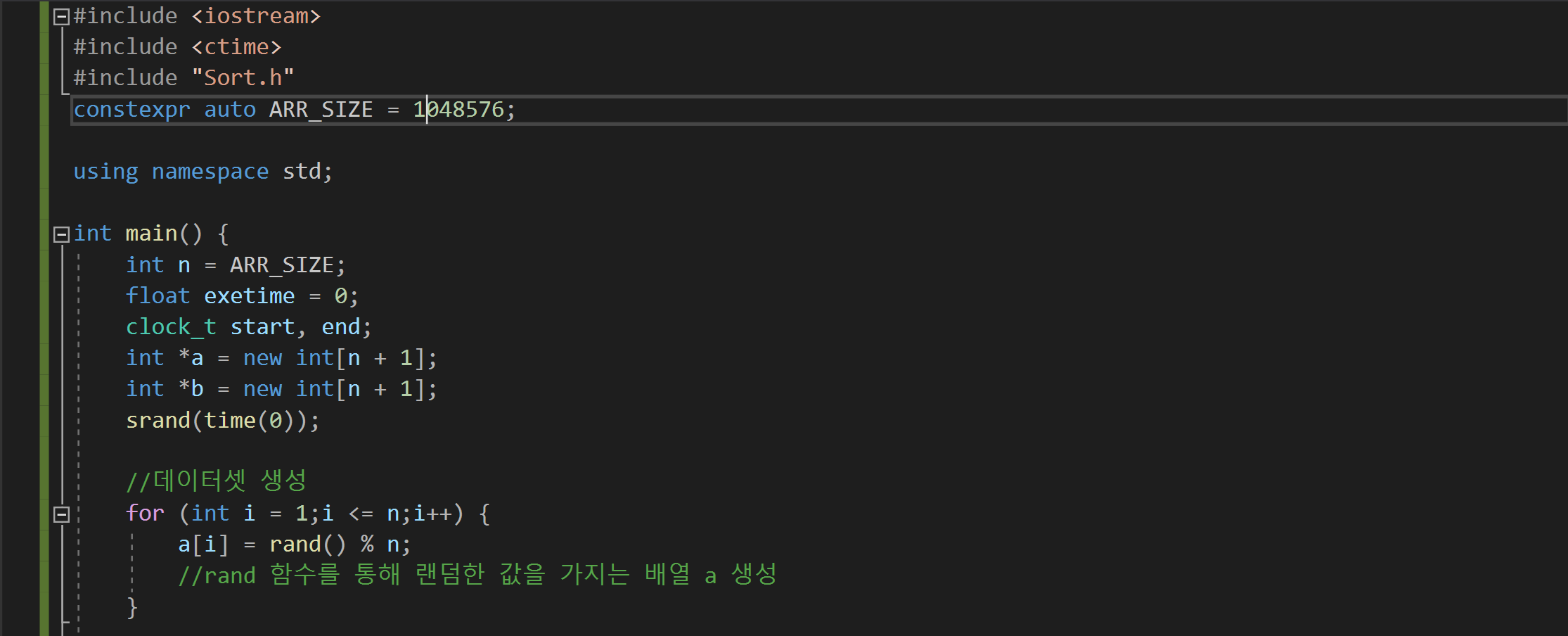


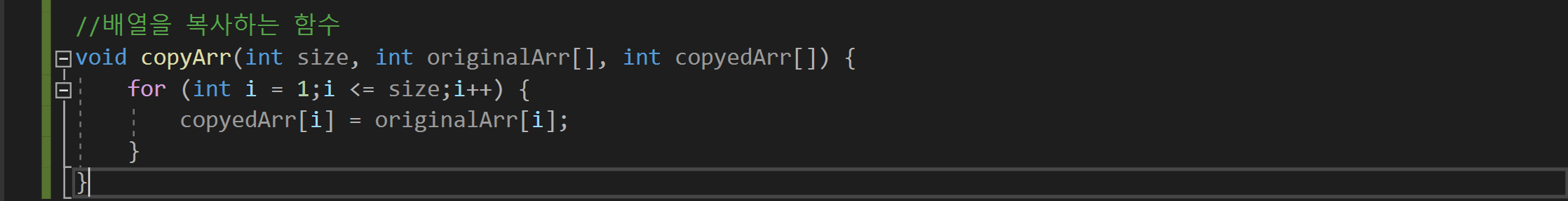
1. 개요

* 상세 설계내용
  + 데이터셋 설정
  + 실행시간 측정
  + Selection Sort
  + Median-of-three Quick Sort
  + Shell Sort
  + Bitonic Sort
  + Odd-Even Merge Sort
* 실행화면
  + Selection Sort
  + Median-of-three Quick Sort
  + Shell Sort
  + Bitonic Sort
  + Odd-Even Merge Sort
* 분석 및 결론

1. 상세 설계내용

* 데이터셋 설정





ARR\_SIZE를 설정하고 그 값을 n에 넣는다.

동적으로 배열을 할당하여 배열의 사이즈가 n+1인 a와 b라는 배열을 만든다.

(이때 배열의 사이즈가 n+1인 이유는 index가 0인 배열은 사용하지 않도록 프로그램을 구현

하였기 때문에 배열의 사이즈가 n+1이 되어야 index가 1~n까지인 배열을 사용할 수 있기

때문이다)

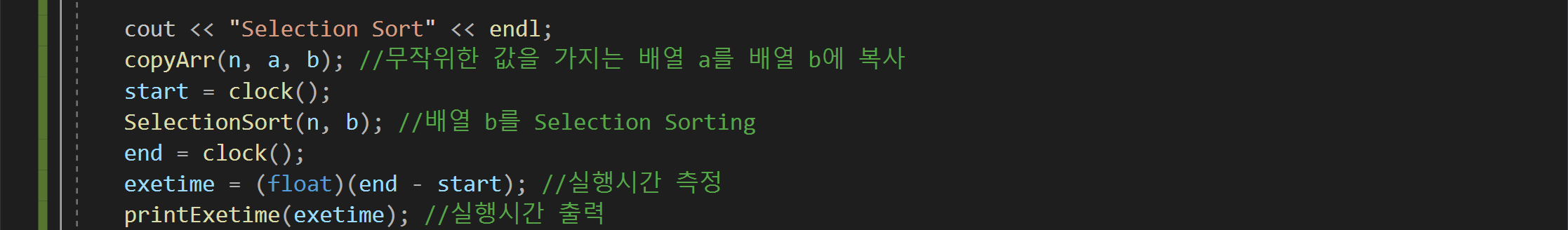
그리고 for문을 보면 rand함수를 통해 배열 a에 random한 값들을 배정한다.

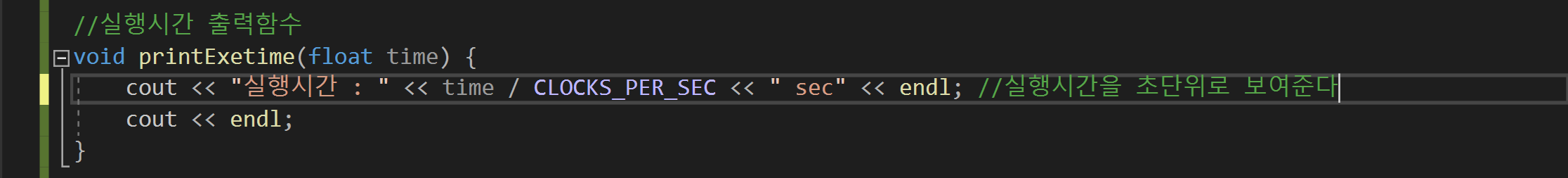
이 배열 a가 원본 배열이 되고 copyArr라는 배열을 복사하는 함수를 통해 b에 a를 복사하고

정렬을 진행하도록 한다. 이렇게 되면 원본배열의 값들은 바뀌지 않게 되고 다섯개의 정렬

알고리즘이 random하게 배정된 동일한 데이터셋을 정렬하게 된다.

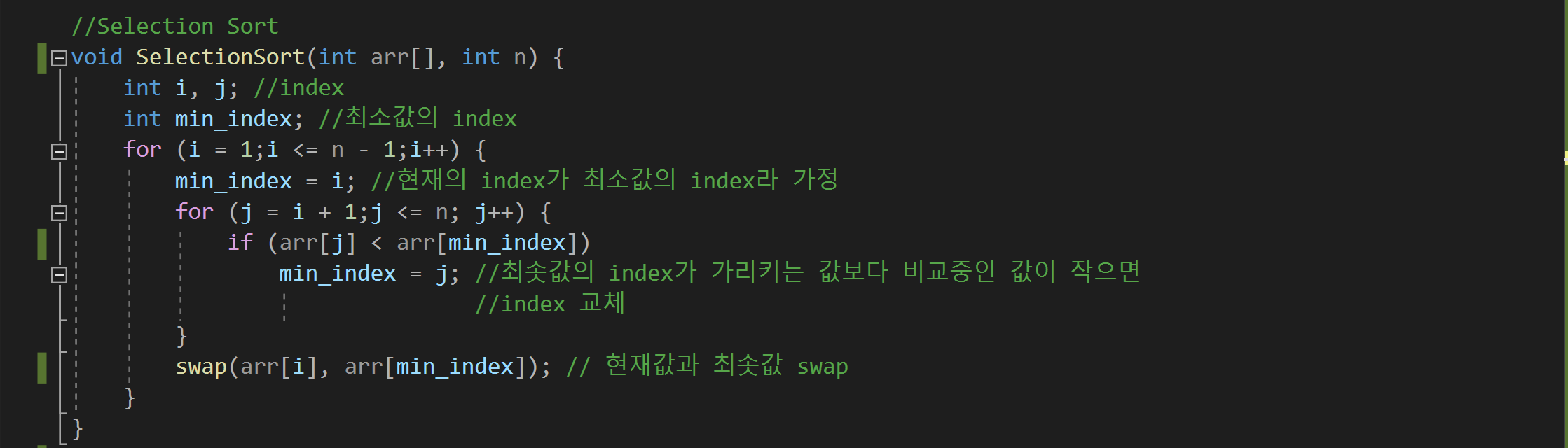
* 실행시간 측정



실행시간을 측정하기 위해 일단 start와 end를 clock\_t로 선언한다.

그리고 정렬이 진행되기 전의 시간을 start에 넣고 정렬이 끝난 후의 시간을 end에 넣는다. 이제 end에서 start값을 빼게 되면 이 값이 정렬에 소요된 실행시간이 되고 값을 exetime에 넣는다. 그리고 printExetime이라는 함수를 통해 실행시간을 출력하는데 원래는 ms(밀리세컨드)단위로 시간이 출력되지만 CLOCK\_PER\_SEC를 활용하여 실행시간을 초단위로 출력할 수 있게 만든다.

* Selection Sort



최소값을 가지는 배열의 index를 min\_index라고 한다.

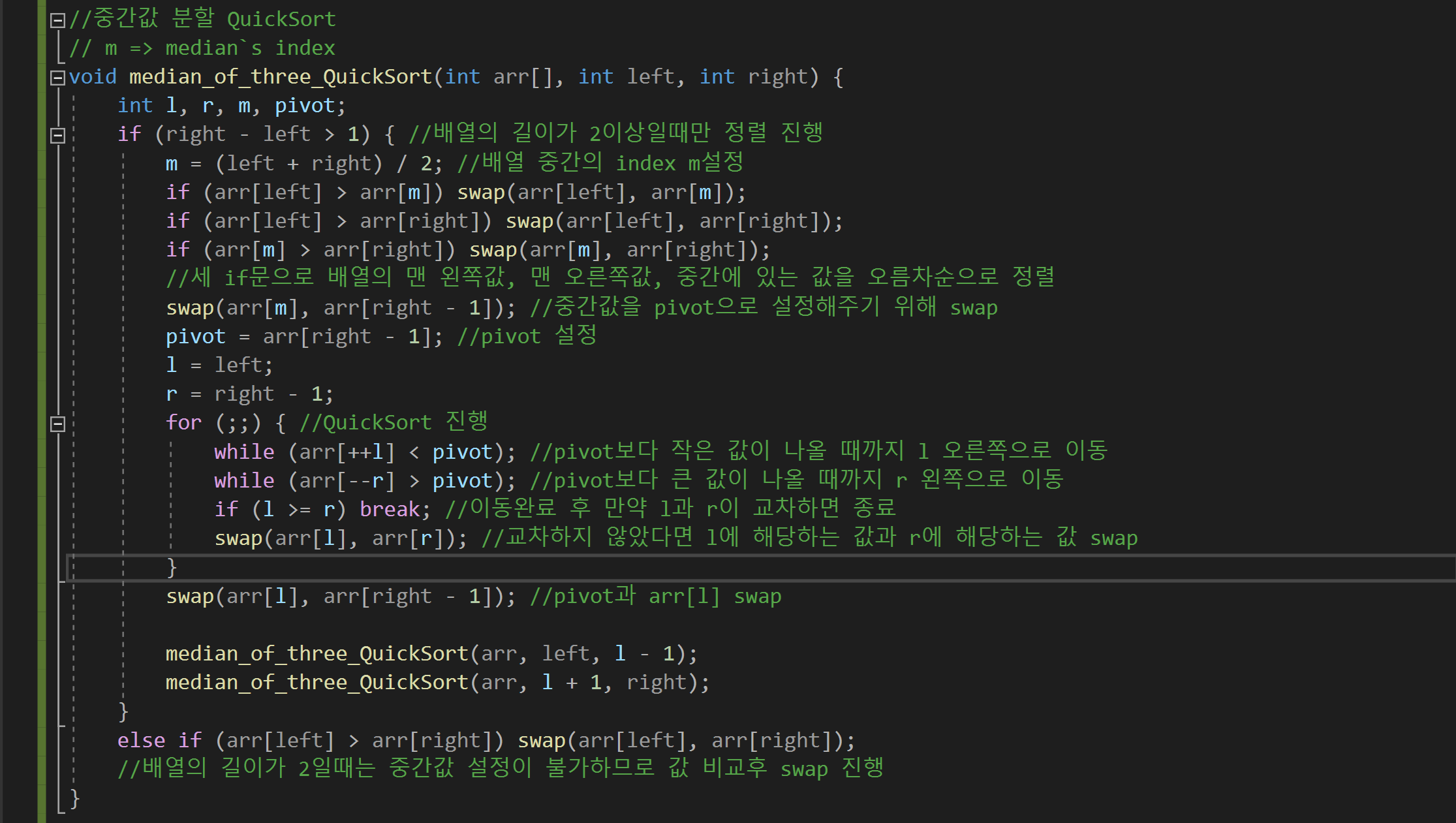
맨처음에는 우선 min\_index = 1로 한다. 즉, arr[1]값을 최소값이라고 가정한다.

그리고 그 다음 index부터 비교를 차례대로 하면서 만약 min\_index가 가리키는 값보다 더 작은 값이 나타난다면 더 작은 값의 index가 min\_index가 된다. 마지막까지 비교가 끝나면 arr[1]에는 배열에 있는 값들 중 제일 최솟값이 오게 되고 맨 처음 정렬을 마친다.

다음에는 min\_index = 2로 한다. 즉, arr[2]값을 최소값이라고 가정한다. 그리고 위와 같은 과정을 반복하면 arr[1]과 arr[2]가 오름차순으로 정렬이 되고 이 둘은 정렬이 완료된 상태가 된다.

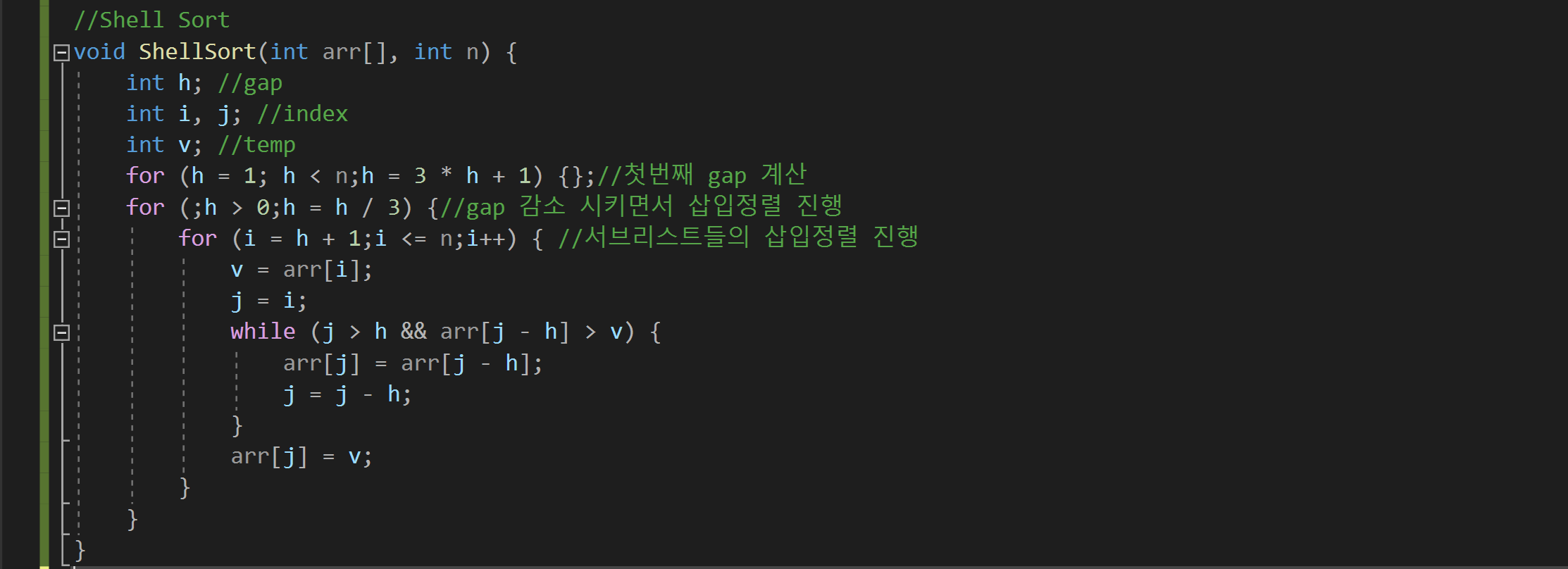
for문을 이용하여 1부터 마지막의 전까지 반복하면 정렬이 완료된다.

* Median-of-three Quick Sort



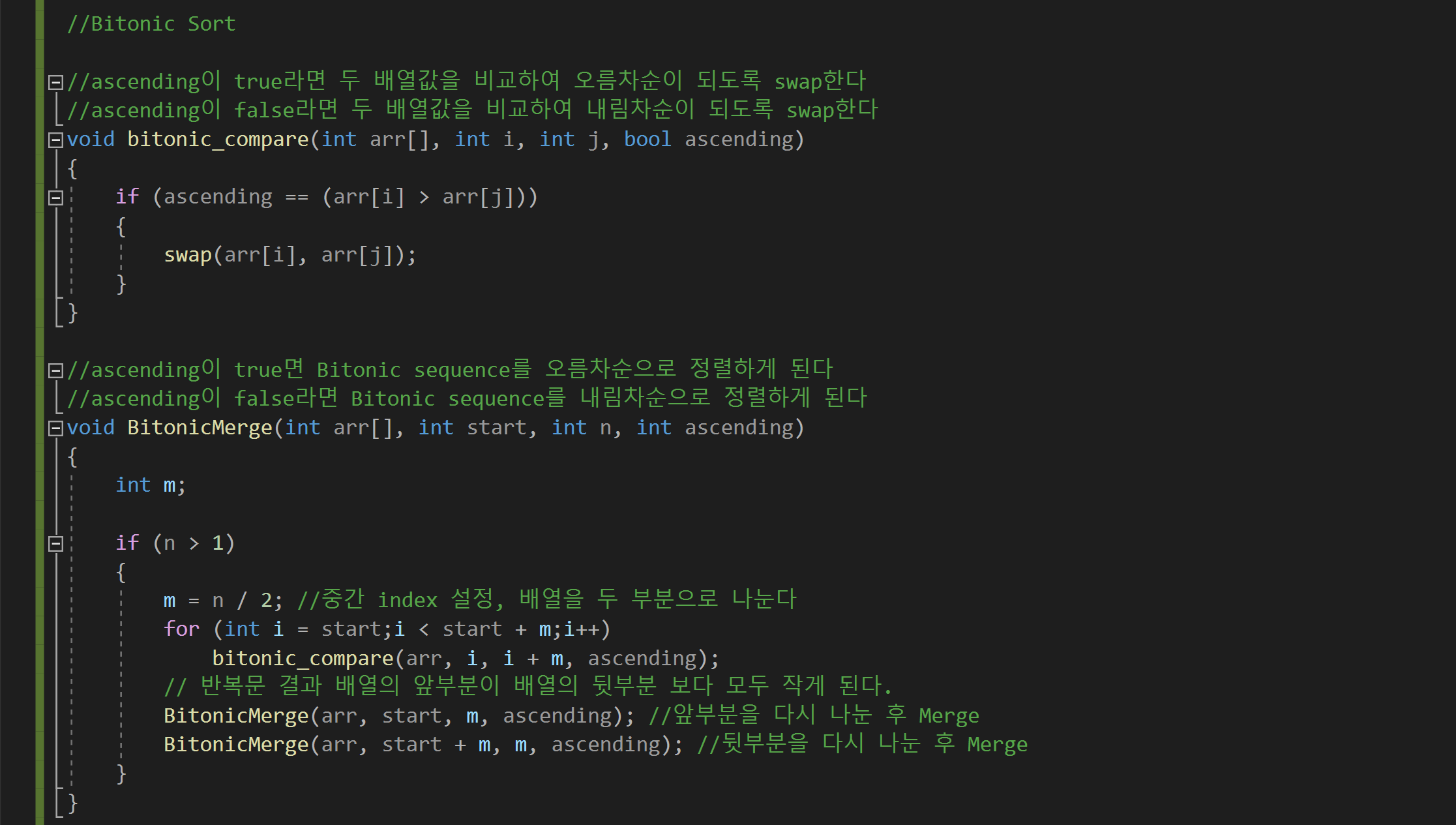
Median-of-three Quick Sort는 Quick Sort에서 pivot을 좀 더 효율적으로 선택하여 worst case를 줄이기 위한 정렬이다. 보통의 Quick Sort에서는 배열의 맨 왼쪽 혹은 오른쪽 값을 pivot으로 지정하고 정렬을 진행하는데 Median-of-three Quick Sort에서는 맨 왼쪽 값, 맨 오른쪽 값, 중간에 있는 값을 비교하여 셋 중에 중간 값을 pivot으로 설정해 worst case를 최대한 피하도록 한다. 그래서 위 코드를 살펴보면 맨 왼쪽 index와 맨 오른쪽 index를 더하여 절반으로 나누어 중간에 있는 index(m)를 구하고 세 값을 비교한 후 중간 값을 pivot으로 설정하는 것을 볼 수 있다. 그리고 나머지 과정은 모두 Quick Sort와 동일하게 진행된다.

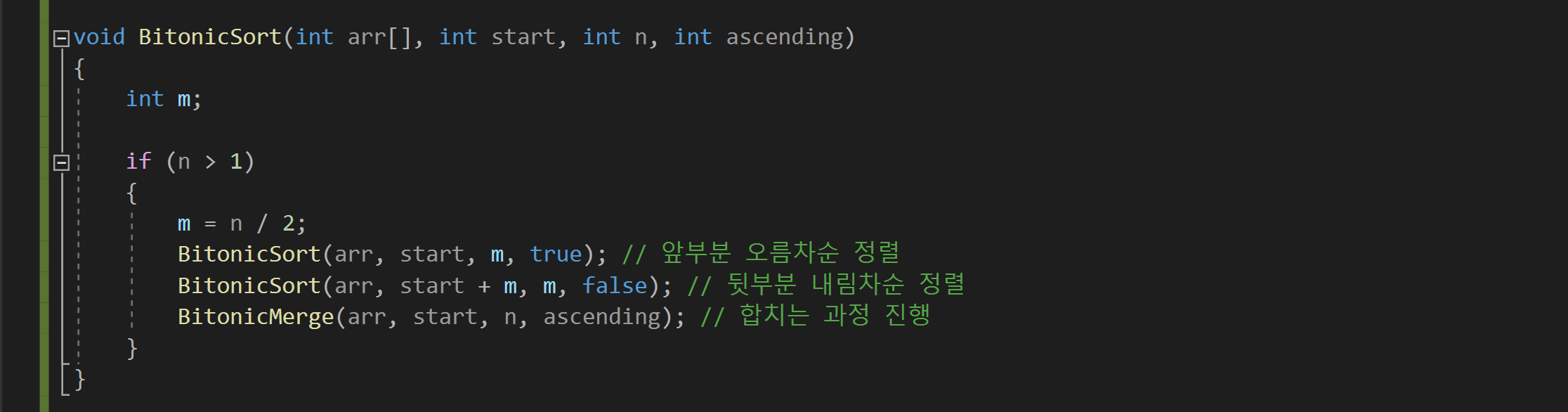
* Shell Sort



Shell Sort는 Insertion Sort를 변형한 정렬로 정렬해야 할 리스트의 각 h번재 요소들을 추출해서 h개의 서브리스트를 만들고 그 서브리스트들에 대해 Insertion Sort를 진행한다. 첫번째 반복문에서 h를 결정하게 되는데 h는 서브리스트들의 간격(gap)이 된다. 첫번째 반복문을 살펴보면 h는 n(배열 크기) 보다는 작고 3h+1이라는 증가식과 함께 값이 커진다. 이렇게 h가 결정되면 두번째 반복문에서는 h값을 줄여가면서 서브리스트들의 Insertion Sort를 진행하게 된다.

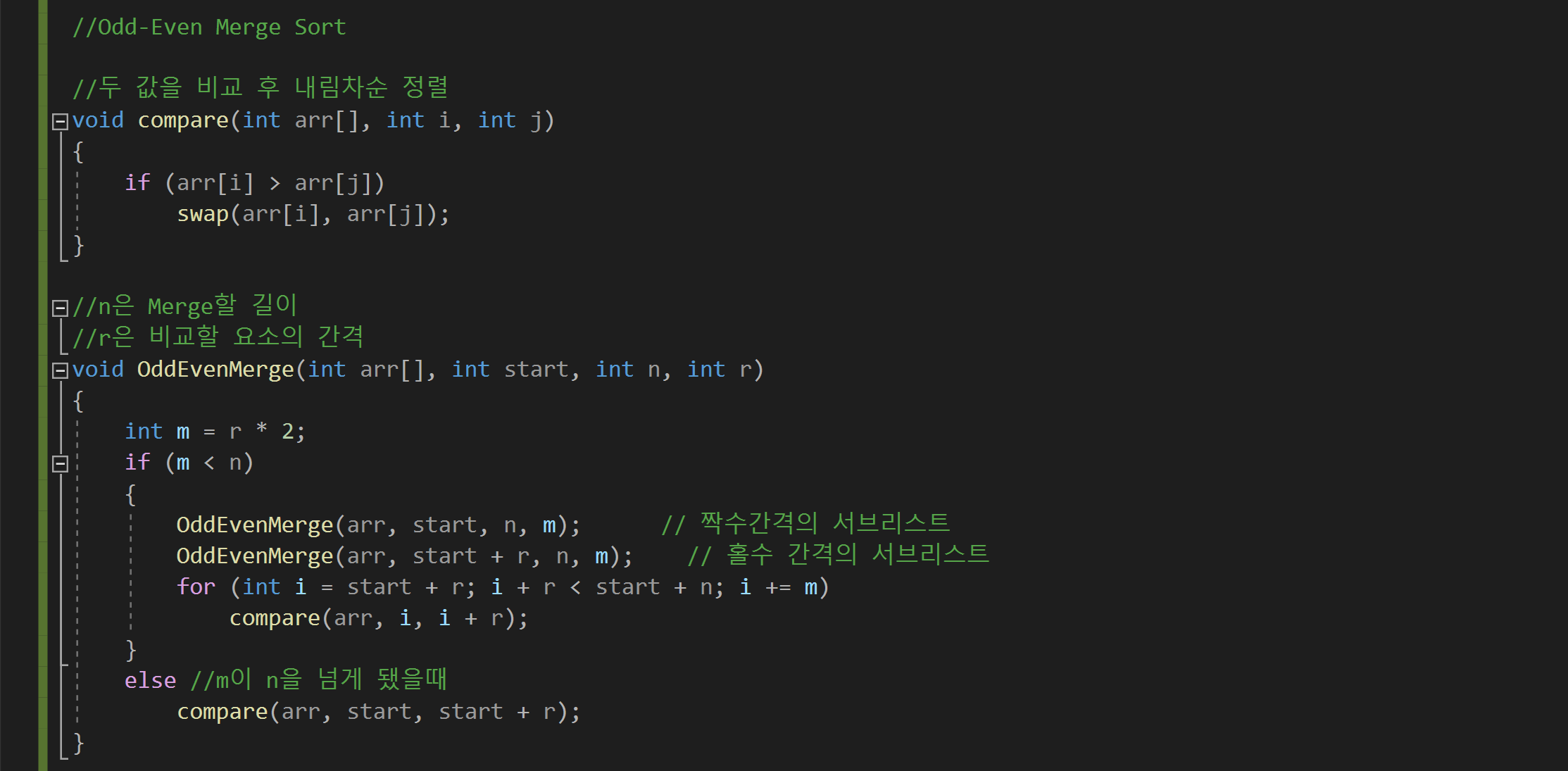
* Bitonic Sort

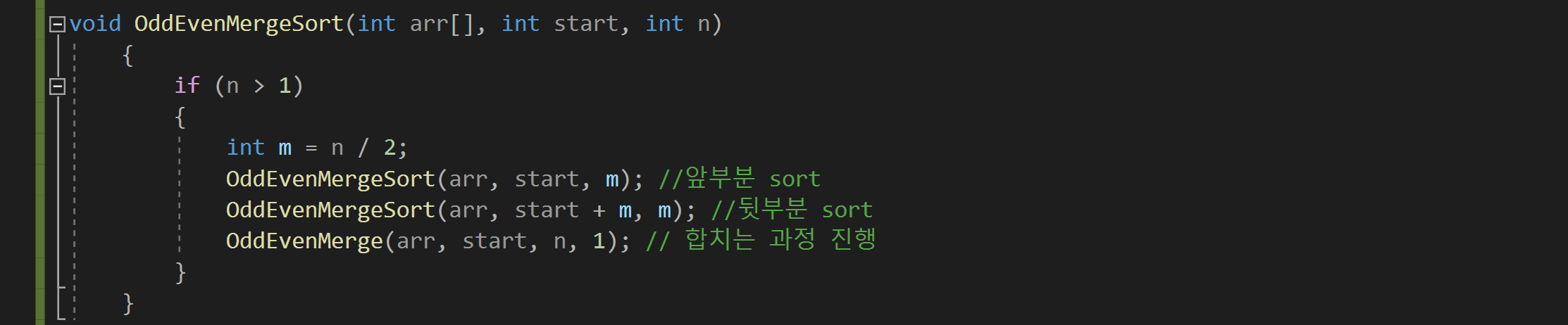




BitonicSort 함수는 먼저 배열의 두 개의 부분을 반대 방향(↗↘ 혹은 ↘↗)으로 재귀적으로 정렬해서 Bitonic sequence를 만든 후 BitonicMerge함수를 호출한다.  
BitonicMerge 함수는 ascending이 true인 경우 Bitonic sequence를 오름차순으로 정렬하고 false면 내림차순으로 반복적으로 정렬한다.

* Odd-Even Merge Sort

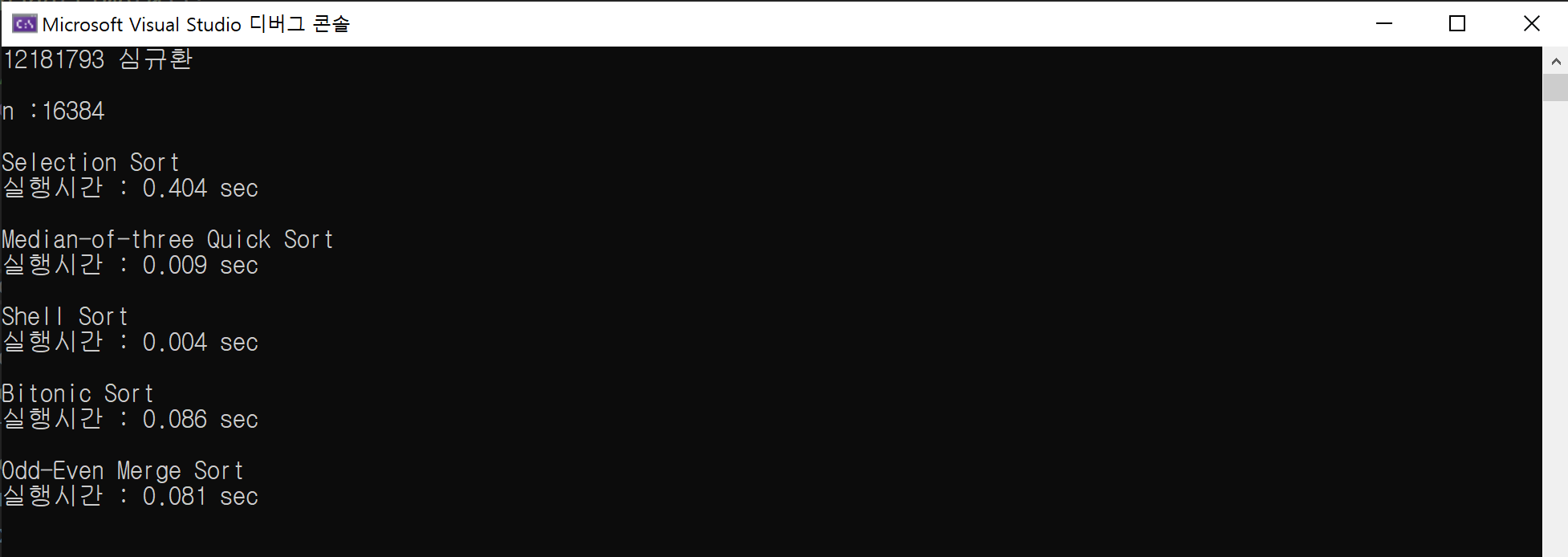




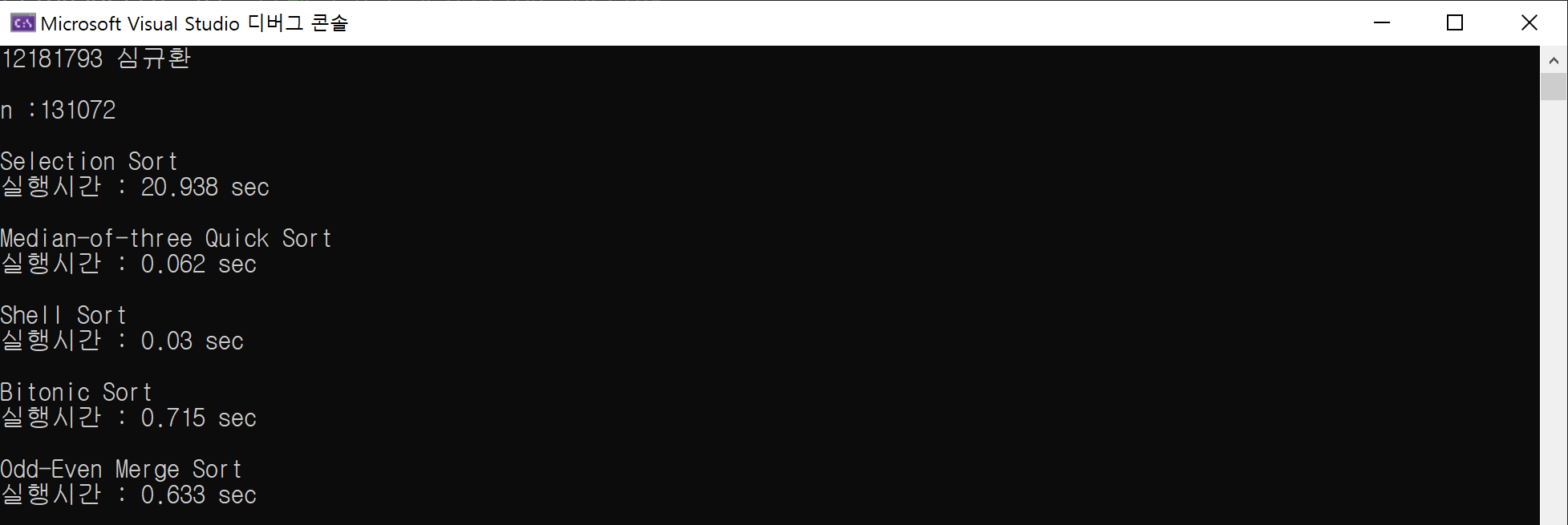
OddEvenMergeSort함수는 배열의 두 부분을 재귀적으로 정렬한다. 그 다음 두 부분을 OddEvenMerge함수를 호출하여 합치는 과정을 진행한다.  
OddEvenMerge함수는 start와 start+r에서 시작하는 2r번째 요소를 선택하여 서브리스트들을 만들게 되는데 이는 각각 짝수간격의 서브리스트와 홀수간격의 서브리스트를 형성한다.

1. 실행 화면

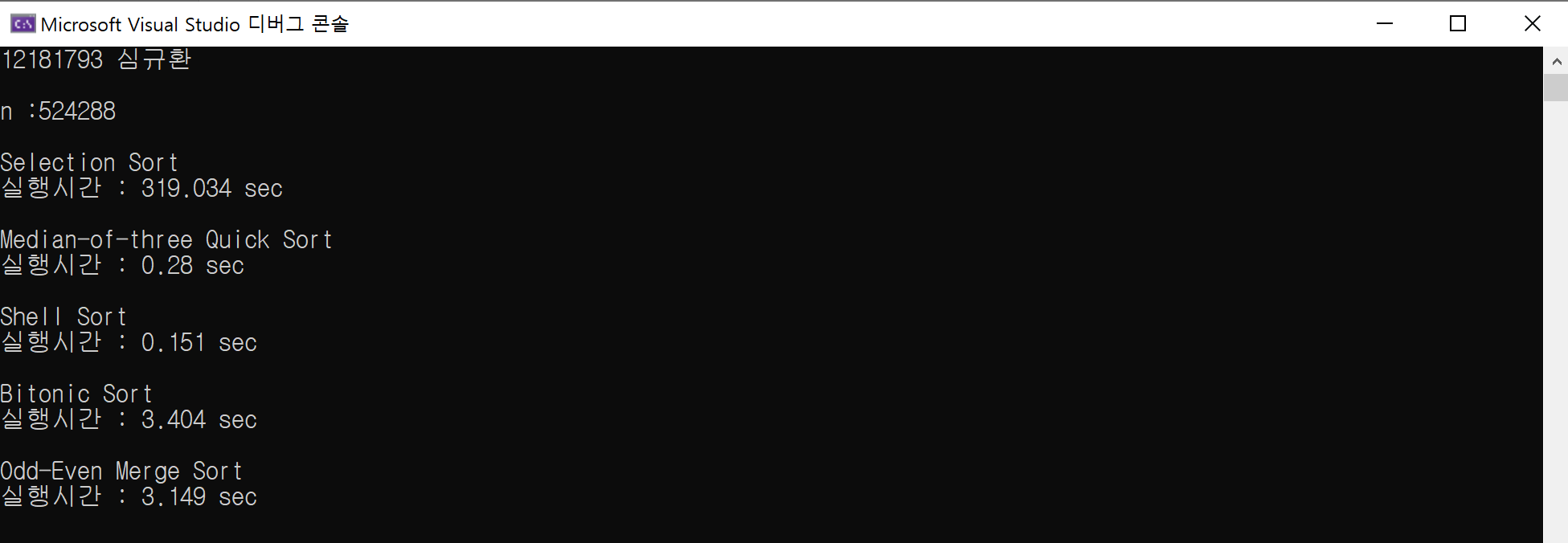
n=214일 때 실행결과



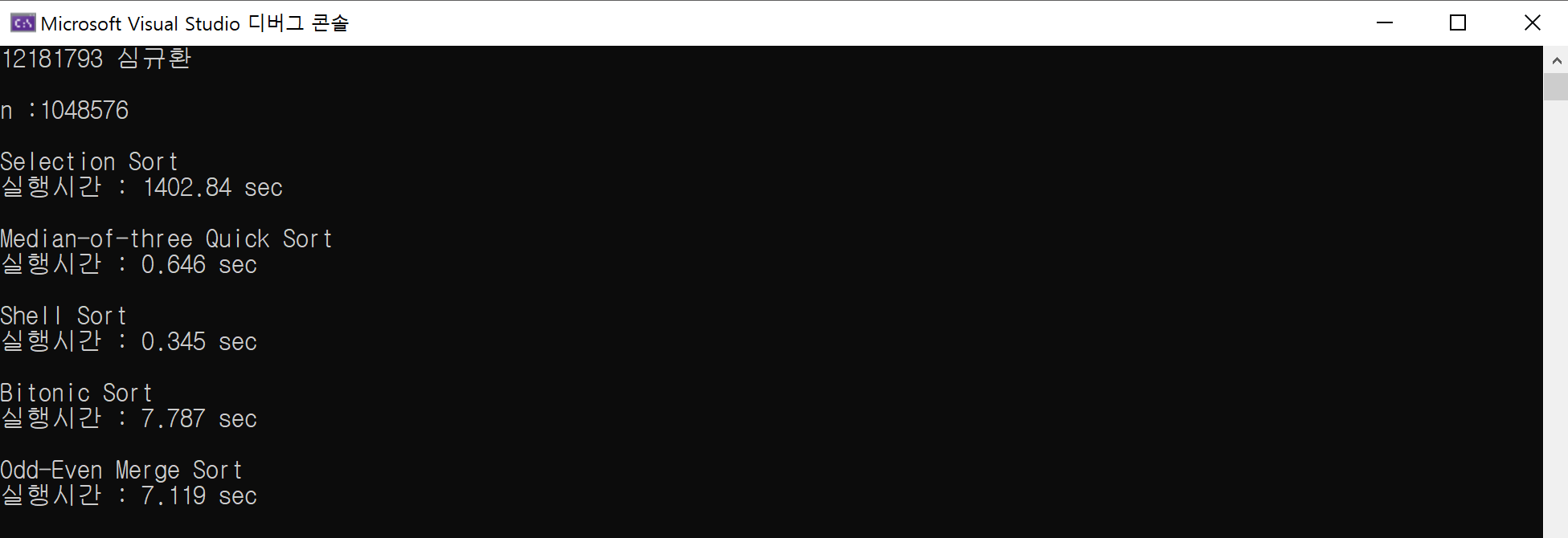
n=217일 때 실행결과



n=219일 때 실행결과



n=220일 때 실행결과

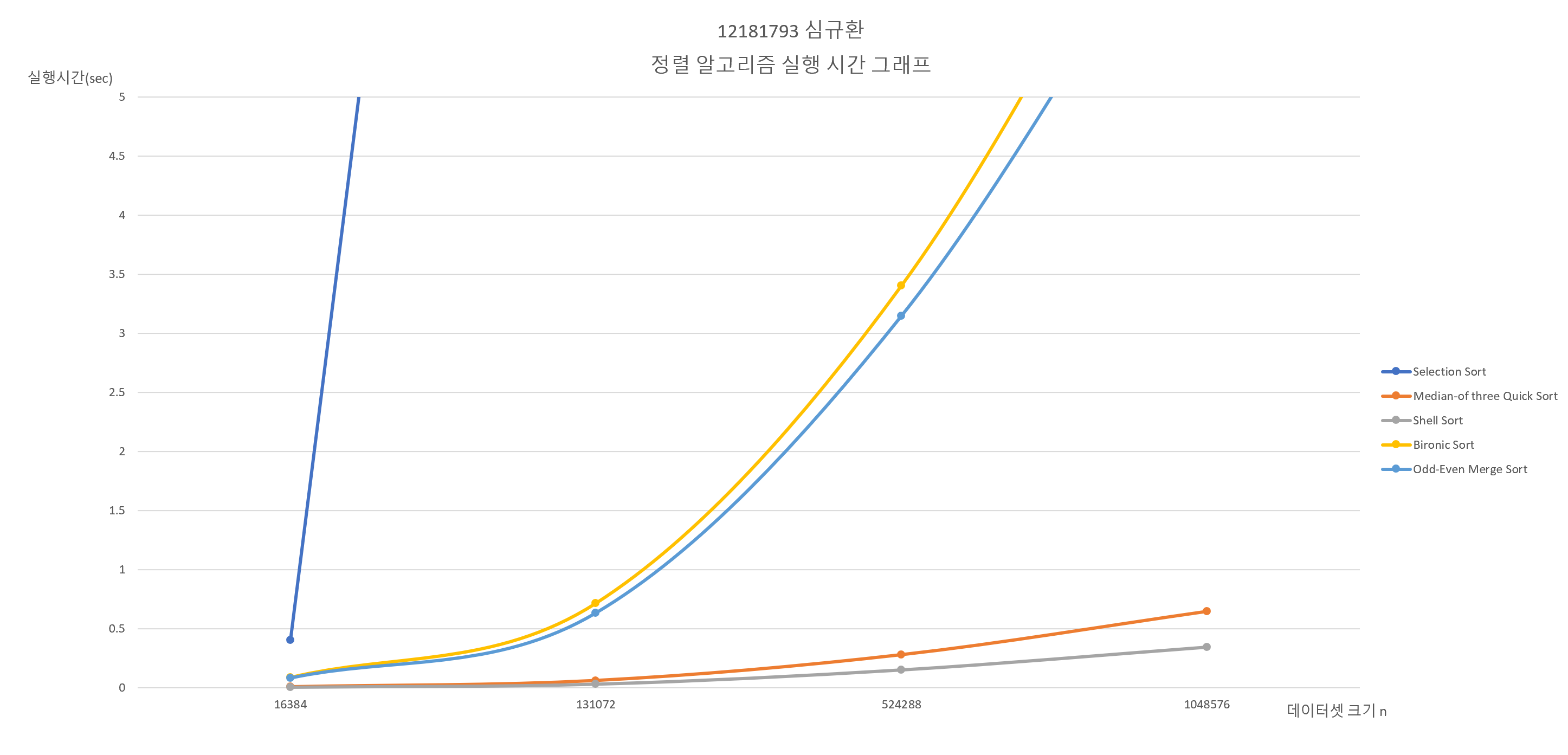


1. 분석 및 결론

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best | 평균 | Worst |
| Selection Sort | O(n2) | O(n2) | O(n2) |
| Median-of-three Quick Sort | O(nlogn) | O(nlogn) | O(n2) |
| Shell Sort | O(n log2(n))  (best known worst case gap sequence) | O(nlogn)  (most gap sequence) | O(n2)  (worst known worst case gap sequence) |
| Bitonic Sort  (parallel time) | O(log2(n)) | O(log2(n)) | O(log2(n)) |
| Odd-Even Sort  (parallel time) | O(log2(n)) | O(log2(n)) | O(log2(n)) |
| Bitonic Sort  (non-parallel time) |  |  | O(nlog2(n)) |
| Odd-Even Sort  (non-parallel time) |  |  | O(nlog2(n)) |

위 표는 위키피디아와 지금까지 공부했던 강의자료들을 참고해 각 정렬 알고리즘들의 시간 복잡도를 정리한 것이다. 이를 보면 병렬적으로 수행한 Bitonic Sort와 Odd-Even Sort의 성능이 가장 좋고 그 다음으로는 Shell Sort와 Median-of-three Quick Sort이고 다음으로는 병렬적으로 수행하지 않은 Bitonic Sort와 Odd-Even Sort이고 Selection Sort가 가장 안 좋을 것이라는 것을 예상할 수 있다.

즉, Bitonic Sort, Odd-Even Sort(parallel time)> Shell Sort Median-of-three Quick Sort> Bitonic Sort, Odd-Even Sort(non-parallel time)> Selection Sort 순으로 성능이 안 좋다는 것을 예측 할 수 있다.



위의 직접 정렬들을 구현하고 실행한 결과를 나타낸 그래프이다. 병렬적으로 수행된 Bitonic Sort와 Odd-Even Sort는 c++로 구현하기 어려워 직접 실행해보진 못하였지만 그래프를 살펴보면 병렬적으로 수행된 Bitonic Sort와 Odd-Even Sort를 제외했을 때 예상과 비슷하게 Shell Sort > Median-of-three Quick Sort > Odd-Even Sort> Bitonic Sort > Selection Sort순으로 실행시간이 느렸고 Shell Sort와 Median-of-three Quick Sort를 비교했을 때 큰 차이는 없지만 데이터셋의 크기가 커질수록 점점 차이가 날 것이라고 예상할 수 있었다. Odd-Even Sort와 Bitonic Sort를 비교했을 때도 마찬가지로 예상할 수 있었다.

결론적으로는 직접 실행해본 결과로는 시간 복잡도가 보여주듯이 Shell Sort의 성능이 가장 뛰어났고 Selection Sort의 성능이 가장 떨어진다는 것을 확인 할 수 있다.