[알고리즘 설계와 분석]

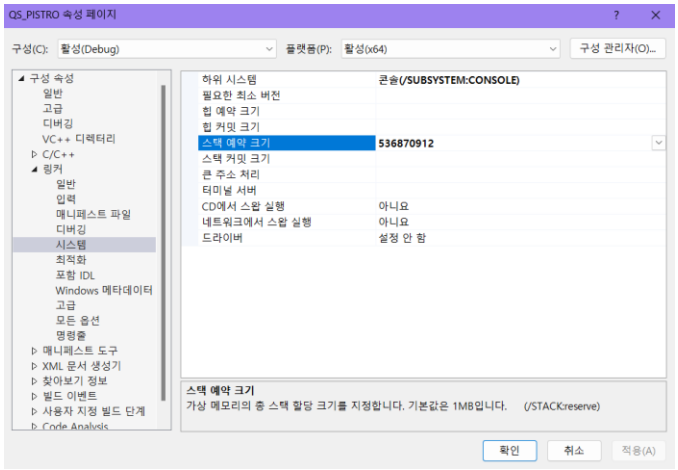
2023 학년도 1 학기 – HW2

학번	20190785
이름	박수빈
과목코드	CSE3081
분반	01
담당 교수님	임인성 교수님

1. 기본 설정

본격적인 실험 시작에 앞서, 실험에 사용할 컴퓨터의 CPU 의 속도 및 메인 메모리의 용량과 같은 기본 정보를 먼저 확인한다. 실험하고자 하는 프로그램은 N_ELEMENTS 라는 이름으로 처리해야 하는 입력 데이터의 사이즈를 정의하고 있다. 본 실험에서 실행하고자 하는 N_ELEMENTS 의 최대 크기는 1,048,576(=2^20)으로, 약 1,048,576 개의 unsigned int 형 데이터를 컴퓨터가 처리할 수 있어야 한다. 프로그램 수행에 필요한 컴퓨터 메모리는 최소 32MB (=1,048,576 * 32 bytes + 32bytes) 이상이다. 아래의 정보에 따르면 본 컴퓨터는 7.80GB 의 메모리 사용이 가능하므로, 해당 프로그램 수행에 충분한 크기의 메모리를 장착하고 있음을 확인할 수 있다. 참고로, 컴파일러로는 채점 환경과 동일한 조건으로 맞추기 위해 Visual Studio 2022 Release Mode 를 선택하였다.

장치 이름	DESKTOP-HUR7U9Q
프로세서	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz
설치된 RAM	8.00GB(7.80GB 사용 가능)
장치 ID	96F26FC7-5BA8-44F8-BFF1-7DCE5CDD67D1
제품 ID	00325-81573-07053-AAOEM
시스템 종류	64 비트 운영 체제, x64 기반 프로세서
컴파일러	Visual Studio 2022 Release Mode



또한, Visual Studio 2022 에서 '스택 예약 크기'를 536,870,912(=2^29)으로 설정도 바꾸어 실험에 적합한 환경으로 조성하였다.

2. 구현 방법

본 실험의 주된 목적 중 하나는 여러 종류의 정렬 방법을 구현한 다음, 수업 시간에 배운 이론적인 시간 복잡도와 실제 프로그램 실행 시간과의 관계를 분석하는 것이다. 따라서, 정렬 방법을 구현하는 각각의 함수 코드는 다르지만, 전체적인 틀은 아래와 같다. 아래에서 서술하는 'n'은 input data size 로, 앞서 언급한 N_ELEMENTS 와 동일하다.

2-1) Skeleton code

1. 바이너리 파일 읽기 (바이너리 파일의 첫 번째 원소는 입력 받을 데이터의 개수 기입)
2. 만일 읽을 바이너리 파일이 없다면 직접 데이터 생성, input binary file 생성
3. 바이너리 파일을 읽어서 버퍼에 원소들을 차례대로 저장
4. 정렬 함수 실행 시간 시작점 측정
5. 정렬 함수 실행
6. 정렬 함수 실행 시간 종점 측정
7. 정렬 함수 실행하는 데 걸린 시간 출력

2-2) Input data

(a) Random: 과제에서 제공한 Test_data_generation.cpp 에 있는 코드 사용

(b) Ascending: 1 부터 n 까지 오름차순으로 정리된 데이터로, 아래의 for 문을 통해 생성

```
buffer_as[0] = N_ELEMENTS;
for (int i = 1; i <= n; i++) {//input ascending number to the buffer_as
    buffer_as[i] = i - 1;
}
```

(c) Descending: n 부터 1 까지 내림차순으로 정리된 데이터로, 아래의 for 문을 통해 생성

```
buffer_de[0] = N_ELEMENTS;
for (int i = 1; i <= n; i++) { //input descending number to the buffer_de
    buffer_de[i] = n - i ;
}
```

(d) Few swaps: 과제에서 제시한 조건을 만족하는 데이터로, 아래와 같이 구현

```
buffer_fw[0] = N_ELEMENTS;
for (int i = 1; i <= n; i++) { //copy buffer_as to buffer_fw
    buffer_fw[i] = buffer_as[i];
}
int index1, index2, n_fw = int(sqrt(n)), temp;
for (int i = 1; i <= n; i++) { //input 'few swap' number to the buffer_fw
    index1 = rand() % n+1; //create random number pair ==> (index1, index2)
    index2 = rand() % n+1;
    while (abs(index1 - index2) >= n_fw) { //if the pair doesn't meet the condition, recreate the pair
        index1 = rand() % n+1;
        index2 = rand() % n+1;
    }
    temp = buffer_fw[index1];
    buffer_fw[index2] = buffer_fw[index1];
    buffer_fw[index1] = temp; //swap each other in the pair
}
```

2-3) Sorting function code

모든 정렬 함수는 어떠한 이유에 의해서든 정렬을 마치지 못한 경우에는 아래 사진과 같이 '-1'을 에러 코드로 반환하도록 설계하였다. 그리고 만일 정렬 함수가 -1 을 반환하였을 경우에는 'The execution of (정렬 방법 이름) sort for (입력 데이터 크기) elements in (데이터 정렬 상태) order is failed.'와 같은 에러 메시지를 출력하도록 설정하였다.

```
if (left < 0 || right < 0 || left > right || data==NULL) {
    return -1;
}
```

(a) Insertion sort

```
int Insertion_Sort(unsigned int* data, int left, int right) {
    if (left < 0 || right < 0 || left > right || data==NULL) {
        return -1;
    } //return error code when execution is not available

    int i, j;
    unsigned int temp;
    int size = right - left+1; //size of data
    for (i = 1; i <= size; i++) {
        temp = data[i];
        j = i;
        while ((temp < data[j - 1]) && (j > 1)) {
            data[j] = data[j - 1];
            j--;
        }
        data[j] = temp;
    }
    return 0;

}
```

우선 for loop 은 검사 대상의 index 를 나타낸다. 이 검사 대상과 비교할 대상이 while loop 의 data[j-i]에 해당한다. 만일 검사 대상보다 비교 대상이 크면 검사 대상은 비교 대상의 앞에 삽입되어야 하므로, data[j]=data[j-1]이라는 구문을 통해 비교 대상이었던 것을 뒤로 밀어낸다. 이런 식으로 while loop 을 돌다가 j 가 1 에 도달하거나 비교 대상보다 커지는 순간이 오면 while loop 을 빠져나와 data[j]=temp 라는 구문을 통해 본인이 해당 자리에 들어간다.

(b) Merge sort

```
int Merge_Sort(unsigned int* data, int left, int right) {
    if (left < 0 || right < 0 || left > right || data == NULL) {
        return -1;
    }

    int middle;

    if (left < right) {
        middle = (right + left) / 2; //Divide 단계
        Merge_Sort(data, left, middle); //Conquer 단계
        Merge_Sort(data, middle + 1, right); //Conquer 단계

        Merge(data, left, middle, right);//Merge 단계

    }
    return 0;
}
```

합병 정렬은 데이터를 계속 나누다가 다시 합치는 divide -and-conquer 방식을 사용한다. 중간에 위치한 원소를 기점으로 left 부터 중간까지, 중간 그 다음부터 right 까지, 이렇게 두 덩어리로 divide 하여 recursion 을 통해 conquer 단계를 거친 다음, Merge 라는 함수를 통해 다시 합쳐진다. Merge 함수는 아래와 같다. 이 함수에 대한 자세한 설명은 line by line 주석으로 달려있다.

```
void Merge(unsigned int* data, int left, int middle, int right) {
    unsigned int buffer[N_ELEMENTS+1] = { 0 };
    int data_index, left_walker, right_walker;

    for (int i = 1; i <= N_ELEMENTS; i++) {
        buffer[i] = data[i];
    }

    data_index = left; //buffer에서의 comparison result를 data에 기입하기 위한 index
    left_walker = left; //왼쪽 영역 walker의 시작점
    right_walker = middle + 1; //오른쪽 영역 walker의 시작점


    while ((right_walker <= right) && (left_walker <= middle)) {  //left_walker가 왼쪽 영역의 종점(middle)에, right_walker가 오른쪽 영역의 종점(right)에 닿을 때까지
        if (buffer[right_walker] > buffer[left_walker]) data[data_index++] = buffer[left_walker++]; //만일 왼쪽 영역의 숫자가 오른쪽 영역의 숫자보다 작으면 data의 해당 index에 왼쪽 영역의 숫자 먼저 기입
        else data[data_index++] = buffer[right_walker++]; //대소관계가 앞과 반대라면, data에는 오른쪽 영역의 숫자를 기입
    }

    while (right_walker <= right) data[data_index++] = buffer[right_walker++]; //만일 여전히 오른쪽 영역의 walker가 끝(right)에 도달하지 않았다면, 남은 오른쪽 영역의 숫자들을 차례대로 data에 반영
    while (left_walker <= middle) data[data_index++] = buffer[left_walker++]; //만일 여전히 왼쪽 영역의 walker가 끝(middle)에 도달하지 않았다면, 남은 왼쪽 영역의 숫자들을 차례대로 data에 반영


}
```

간략하게 요약하자면, 입력 받은 데이터를 임시 버퍼에 복사한 다음, 임시 버퍼에서 비교 과정을 거쳐서 그 비교 결과를 원래의 데이터에 반영하는 식으로 작동한다. 비교하는 방법은 중간을 기준으로 왼쪽과 오른쪽을 나누어서, 왼쪽의 맨 왼쪽 원소와 오른쪽의 맨 왼쪽 원소에서 각각 시작하여, 둘 중 작은 것을 먼저 원래 데이터에 반영한 다음 차례대로 넘어가는 식이다.

(c) Quick sort NAIVE

```
int Quick_Sort_NAIVE(unsigned int* data, int left, int right) {
    if (left < 0 || right < 0 || left > right || data == NULL) {
        return -1;
    }
    int pivot;
    int size = right - left + 1;
    if (size>=4) { //원소의 개수가 4개 이상일 때 recursive하게 해결
        pivot = partition(data, left, right); //Divide 단계
        Quick_Sort_NAIVE(data, left, pivot - 1); //Conquer 단계
        Quick_Sort_NAIVE(data, pivot+1, right);//Conquer 단계
    }
    else if (size==3) { //원소의 개수가 3개일 때, 직접 조정
        if (data[left] > data[left + 1]) SWAP(data[left], data[left + 1]);
        if (data[left + 1] > data[right]) SWAP(data[left + 1], data[right]);
        if (data[left] > data[left + 1]) SWAP(data[left], data[left + 1]);
    }
    else if (size == 2) { //원소의 개수가 2개일 때, 직접 조정
        if (data[left] > data[right]) SWAP(data[left], data[right]);
    }
    return 0;
}
```

퀵소트는 피벗 원소를 하나 뽑아, 그 피벗 원소보다 작은 값들은 왼쪽으로, 큰 값들은 오른쪽으로 밀어 넣는 과정을 반복하여 정렬하는 방식이다. 이 정렬의 특이한 점은 Divide 단계와 Conquer 단계는 있지만, Merge 단계는 필요 없다는 점이다. 그리고 과제의 요구 조건을 충족시키기 위해, 입력 데이터의 크기(위의 사진에서는 변수'size')가 3 이하인 경우에는 직접 SWAP을 하여 조정하는 식으로 설계하였다. 이 퀵소트에서는 피벗 원소를 기준으로 원래의 데이터를 나눠주는 partition 함수가 매우 중요하다. 이 partition 함수에서 피벗 원소를 어떤 기준으로 뽑는지에 따라 퀵소트의 효율성이 결정되기 때문이다. 우선, NAIVE 버전에서는, 과제에서 요구하는 조건을 따라 피벗 원소를 단순히 맨 왼쪽 원소로 결정하고 나누는 방식으로 partition 함수를 작성하였다. 이 partition 함수에 대한 자세한 사항은 아래에 사진으로 첨부하였다.

```
int partition(unsigned int* data, int left, int right) {
    int i, pivot;

    pivot = right; //pivot의 시작점
    for (i = right; i> left; i--) { //i와 pivot은 right부터 조사
        if (data[i] > data[left]) { //pivot으로 뽑은 data[left]와 비교
            SWAP(data[pivot], data[i]); //pivot으로 뽑은 data[left]보다 작으면 왼쪽편으로, 크면 오른쪽편으로 이동시키는 역할
            pivot--;
        }
    }
    SWAP(data[left], data[pivot]); //마지막으로 data[left]를 pivot 자리에 넣어주기
    return pivot;
}
```

참고로, 위 사진의 주석들은 코드의 각 줄의 의미를 문법적으로 정확하게 해석하는 데보다는 전반적인 알고리즘을 이해하는 데 초점을 맞추어 작성되었다.

(d) Quick sort P

```
int Quick_Sort_P(unsigned int* data, int left, int right) {
    if (left < 0 || right < 0 || left > right || data == NULL) {
        return -1;
    }
    int pivot;
    int size = right - left + 1;
    if (size >= 4) { //원소의 개수가 4개 이상일 때 recursive하게 해결
        pivot = partition(data, left, right); //Divide 단계
        Quick_Sort_P(data, left, pivot - 1); //Conquer 단계
        Quick_Sort_P(data, pivot + 1, right);//Conquer 단계
    }
    else if (size == 3) { //원소의 개수가 3개일 때, 직접 조정
        if (data[left] > data[left + 1]) SWAP(data[left], data[left + 1]);
        if (data[left + 1] > data[right]) SWAP(data[left + 1], data[right]);
        if (data[left] > data[left + 1]) SWAP(data[left], data[left + 1]);
    }
    else if (size == 2) { //원소의 개수가 3개일 때, 직접 조정
        if (data[left] > data[right]) SWAP(data[left], data[right]);
    }
    return 0;
}
```

피벗 원소를 (c)에서와 같이 무조건 맨 왼쪽 원소로 정하는 식으로 partition 함수를 작성하면, 맨 왼쪽 원소가 해당 입력 데이터의 최소값이거나 최대값인 경우 skewed 형태로 divide 되면서 정렬의 비효율성이 극대화된다. 계속 skewed 형태로 나뉘어지는 최악의 경우, 시간 복잡도가 O(n^2)까지 되기 때문에 개선의 필요성이 보인다. 이에 따라, 완벽하지는 않지만 그래도 매번 최악의 상황이 되는 경우를 조금이라도 피하기 위해, partition 함수에서 피벗 원소를 left, middle, right 이 세 개의 값 중 중앙값에 해당하는 값을 피벗 원소로 정하는 median-of-three 방식을 선택할 수 있다. Quick_Sort_P 함수는 다른 부분은 모두 (c)에서 구현한 Quick_Sort_NAIVE 와 동일하지만, divide 단계에 해당하는 partition 함수에서 median-of-three 방식으로 피벗 원소를 선택한다는 점만 다르다. 아래 코드는 이 바뀐 partition 함수에 대한 코드이다. 세 값의 중앙값으로 피벗 원소를 정한 다음 data 의 맨 왼쪽 원소와 SWAP 함으로써, 그 이후에는 partition 함수를 NAIVE 버전에서 더 수정할 필요없이 그대로 작성하여 보다 쉽게 구현할 수 있다.

```
int partition(unsigned int* data, int left, int right) {
    int middle = (left + right) / 2;
    unsigned int pivot_index,i;

    //median-of-three method
    int a = data[left]; int b = data[middle]; int c = data[right];
    if (((a <= b) && (b <= c)) || ((c<=b)&&(b<=a))) pivot_index = middle;
    else if (((b <= a) && (a <= c)) || ((c<=a)&&(a<=b))) pivot_index = left;
    else pivot_index = right;

    SWAP(data[pivot_index], data[left]);
    //이후로는 Quick_Sort_NAIVE에서 구현했던 partition과 동일
    pivot_index = right; //pivot의 시작점
    for (i = right; i > left; i--) { //i와 pivot은 right부터 조사
        if (data[i] > data[left]) { //pivot으로 뽑은 data[left]와 비교
            SWAP(data[pivot_index], data[i]); //pivot으로 뽑은 data[left]보다 작으면 왼쪽편으로, 크면 오른쪽편으로 이동시키는 역할
            pivot_index--;
        }
    }
    SWAP(data[left], data[pivot_index]); //마지막으로 data[left]를 pivot 자리에 넣어주기
    return pivot_index;

}
```

(e) Quick Sort PIS

앞선 (d)번 문항에서 구현한 Quick Sort P 는 NAIVE 버전에서 partition 함수를 수정하여 시간 복잡도 측면에서 개선시킬 수 있었다. 퀵소트를 개선시킬 수 있는 또다른 방법이 이번 문항에서 구현할 Quick Sort PIS 에 해당한다. 이 PIS 버전은 사전에 정의한 숫자 M 보다 크거나 같은 경우에만 퀵소트를 실행하고, M 보다 작은 경우에는 퀵소트를 실행하지 않는다. 이러면 만일 퀵소트를 재귀적으로 실행하면서 심각한 skewed 형태에 도달하는 것을 방지할 수 있다. 이는 앞선 (c)와 (d)에서 구현한 퀵소트의 문제점을 해결해주는 역할을 한다. 그렇다면 M 보다 작은 경우에는 정렬이 완료되지 않는다는 것을 의미하기도 하는 것인데, 이에 대한 설명은 아래 문단에 서술하도록 하겠다.

지난 (a)번 문항에서 Insertion sort 를 구현했었다. 이 삽입 정렬은 데이터가 어느 정도 정렬되어 있을 때, 특히 검사 대상이 삽입될 위치가 멀지 않는 경우, 비교와 할당 과정 실행 횟수가 적기 때문에 매우 유용하다. 이와 같은 Best case 의 경우, 비교 1 회 당 할당이 1 회 이어진다고 가정하면, 1+1+...+1 (비교 총 n 회)= 1*n 이므로, 삽입 정렬의 시간 복잡도는 O(n)까지도 낮아진다. 이러한 삽입 정렬의 장점을 이용하여 Quick sort PIS 버전을 만든 것이다. M 이상이면 재귀적으로 퀵소트를 실행하여 대략적으로 정렬을 해놓고, 마지막에 Insertion_Sort 함수를 호출하여 삽입 정렬로 정렬하면 정렬이 완료된다. 이로써 우리는 퀵소트의 문제점을 어느 정도 개선시킬 수 있는 것이다.

```
int Quick_Sort_P(unsigned int* data, int left, int right) {
    if (left < 0 || right < 0 || left > right || data == NULL) {
        return -1;
    }
    int pivot_index;
    int size = right - left +1;

    if (size >= M) {
        //원소의 개수가 M개 이상일 때 recursive하게 해결
        pivot_index = partition(data, left, right); //Divide 단계
        Quick_Sort_P(data, left, pivot_index - 1); //Conquer 단계
        Quick_Sort_P(data, pivot_index + 1, right);//Conquer 단계
    }
    else return 0;
    return 0;
}

int Quick_Sort_PIS(unsigned int* data, int left, int right) {
    Quick_Sort_P(data, left, right);
    Insertion_Sort(data, left, right);
    return 0;
}
```

이러한 Quick_Sort_PIS 함수는 (d)번 문항에서 구현한 Quick_Sort_P 함수에서 3 이하인 경우에는 직접 SWAP 하여 조정하는 구간을 빼고 M 이하인 경우는 0 을 바로 반환하는 식으로 바꾸고, 이 함수 다음에는 (a)번 문항에서 구현한 Insertion_Sort 함수를 호출하는 구조로 구현하면 된다.

(f) Quick sort PISTRO

이전 (d)번 문항의 three-of-median 방식으로 피벗 원소를 고르고, (e)번 문항과 같이 M 보다 작으면 재귀 호출을 하지 않고 마지막에 삽입 정렬로 정렬하는 방식과 더불어, '꼬리 재귀 최적화 (Tail Recursion Optimization)' 방식도 추가하여 구현한 함수가 바로 이번 문항에서 구현할 퀵소트 PISTRO 버전이다. 앞의 두 방식은 이미 이전에 서술하였으므로 생략하고, 꼬리 재귀 최적화 방식에 대해 아래 문단에 서술하도록 하겠다.

꼬리 재귀 최적화란, 함수의 재귀 호출 완료 이후에 현재 함수 내에서 추가적인 연산을 실행하지 않도록 하는 재귀의 형태를 말한다. 이 방식을 퀵소트에 적용하면, 다음과 같다. 피벗 원소를 뽑은 이후에는 피벗 원소보다 작은 값을 지닌 왼쪽 덩어리, 큰 값을 지닌 오른쪽 덩어리, 이렇게 두 덩어리가 생긴다. 덩어리의 크기가 크면 클수록 재귀 호출을 하는 횟수가 증가하므로, 피벗 원소를 뽑은 이후 덩어리가 두 개 생긴 이후에는 작은 덩어리 쪽으로 재귀 호출을 하면 system stack overflow 를 방지할 수 있다. 덩어리 크기가 큰 쪽은 iteration 으로 처리하고 덩어리가 작은 쪽은 recursive 하게 처리하면 된다. 이와 같은 PISTRO 버전은 아래 사진과 같이 코드를 작성하면 구현할 수 있다. 코드에 대한 자세한 설명은 아래 사진에서 주석으로 서술하였다.

```
int Quick_Sort_PISTRO(unsigned int* data, int left, int right) {
    if (left < 0 || right < 0 || left > right || data == NULL) {
        return -1;
    }
    int left1, right1; //iteration으로 처리하는 segment (덩어리가 큰 쪽)
    int left2, right2; //recursion으로 처리하는 segment (덩어리가 작은 쪽)
    int pivot_index, pivot, size;

    size = right - left + 1;
    if (size < M) {
        Insertion_Sort(data, left, right); //size가 M보다 작으면 '삽입 정렬'로 처리
    }

    left2 = left;
    right2 = right;
    int middle = (left2 + right2) / 2;
    while (right2 - left2 > 1) {

        pivot_index = partition(data, left2, right2);  //select a pivot element
        pivot = data[pivot_index];

        if (pivot_index < middle) { //pivot이 왼편에 있는 경우 => 왼편이 '덩어리가 작은 쪽'
            left1 = left2;  right1 = pivot_index - 1;  //덩어리가 작은 쪽의 index인 left1, right1을 왼편 덩어리로 설정
            left2 = pivot_index + 1; right2 = right2; //덩어리가 큰 쪽의 index인 left2, right2를 오른편 덩어리로 설정
        }
        else { //pivot이 오른쪽에 있는 경우 => 오른편이 '덩어리가 작은 쪽'
            left1 = pivot_index + 1; right1 = right2; //덩어리가 작은 쪽의 index인 left1, right1을 오른편 덩어리로 설정
            left2 = left2; right2 = pivot_index - 1; //덩어리가 큰 쪽의 index인 left2, right2를 왼편 덩어리로 설정
        }
        Quick_Sort_PISTRO(data, left1, right1); //덩어리가 작은 쪽은 recursion으로 처리

    }

    return 0;
}
```

3. 실험방법 및 결과 분석

이전 2-1)번 문항에서 서술한 것과 같이 각 정렬 함수의 실행 시간을 측정하였다. 이렇게 실제로 실험에서 측정한 함수의 실행 시간과 이론적인 시간 복잡도를 비교하는 방식으로 실험을 진행하였다. 후술할 내용에서 n 은 N_ELEMENTS, 즉 input data size 를 의미한다. 실험은 n 의 크기를 달리 하여, 각 함수 별로 3 번씩 수행하였다. 1 번 실험은 n 을 32(=2^5)로, 2 번 실험은 n 을 1024(=2^10)로, 3 번 실험은 n 을 1048578(=2^20)로, 이렇게 3 가지 값을 각 실험에서 넣은 다음 컴파일하여 출력된 데이터 생성 시간 및 함수 실행 시간 측정 결과를 토대로 결과 분석을 진행하였다.

(a) Insertion Sort

1) 이론적인 시간 복잡도

삽입 정렬의 이론적으로 구한 시간 복잡도는 아래와 같다.

	Big-O	Example of input data order
Worst case	O(n^2)	Descending
Average case	O(n^2)	Random, Few swaps
Best	O(n)	Ascending

2) 실험 결과 출력 화면

Input data size	Result
32 (=2^5)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 32 elements in random order= 0.040ms ^^^ Time for executing merge sort for 32 elements in random order = 0.014ms ^^^ Time for generating 32 elements in ascending order= 0.002ms ^^^ Time for executing merge sort for 32 elements in ascending order = 0.013ms ^^^ Time for generating 32 elements in descending order= 0.001ms ^^^ Time for executing merge sort for 32 elements in descending order = 0.011ms ^^^ Time for generating 32 elements in few swap order= 0.073ms 0 1 2 5 0 1 6 7 10 9 10 13 12 14 15 15 12 17 19 12 20 21 21 26 26 26 30 30 26 31 30 31 ^^^ Time for executing merge sort for 32 elements in few swap order = 0.010ms

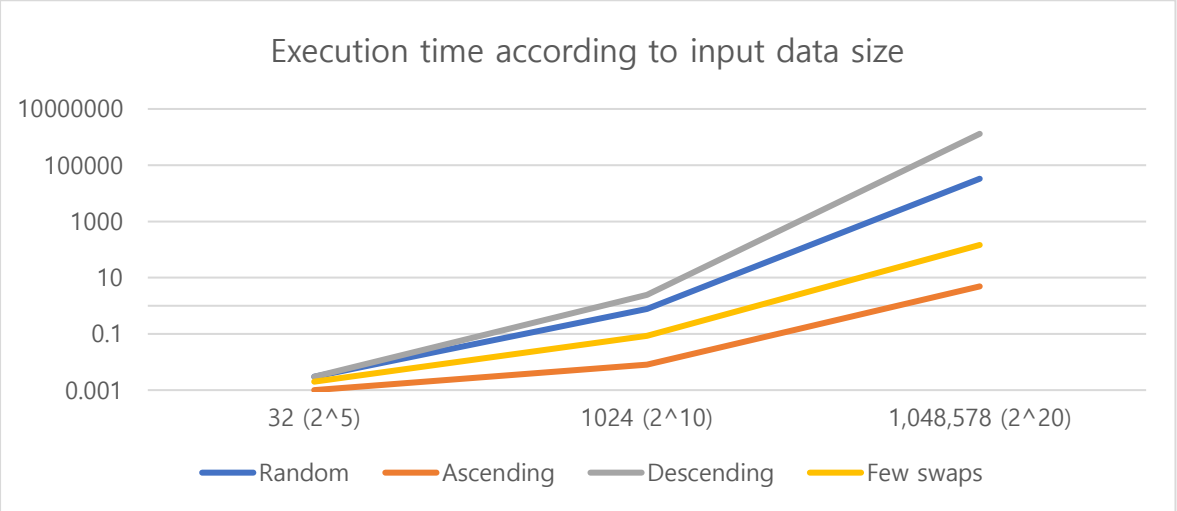
1024 (=2^10)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 1.068ms ^^^ Time for executing insertion sort for 1024 elements in random order = 0.765ms ^^^ Time for generating 1024 elements in ascending order= 0.524ms ^^^ Time for executing insertion sort for 1024 elements in ascending order = 0.008ms ^^^ Time for generating 1024 elements in descending order= 0.426ms ^^^ Time for executing insertion sort for 1024 elements in descending order = 2.418ms ^^^ Time for generating 1024 elements in few swap order= 5.266ms ^^^ Time for executing insertion sort for 1024 elements in few swap order = 0.086ms
1048576 (=2^20)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1048576 elements in random order= 195.453ms ^^^ Time for executing insertion sort for 1048576 elements in random order = 330009.000ms ^^^ Time for generating 1048576 elements in ascending order= 6.096ms ^^^ Time for executing insertion sort for 1048576 elements in ascending order = 4.888ms ^^^ Time for generating 1048576 elements in descending order= 4.952ms ^^^ Time for executing insertion sort for 1048576 elements in descending order = 1307699.375ms ^^^ Time for generating 1048576 elements in few swap order= 2398.059ms ^^^ Time for executing insertion sort for 1048576 elements in few swap order = 145.419ms

3) 실험 결과 정리 표

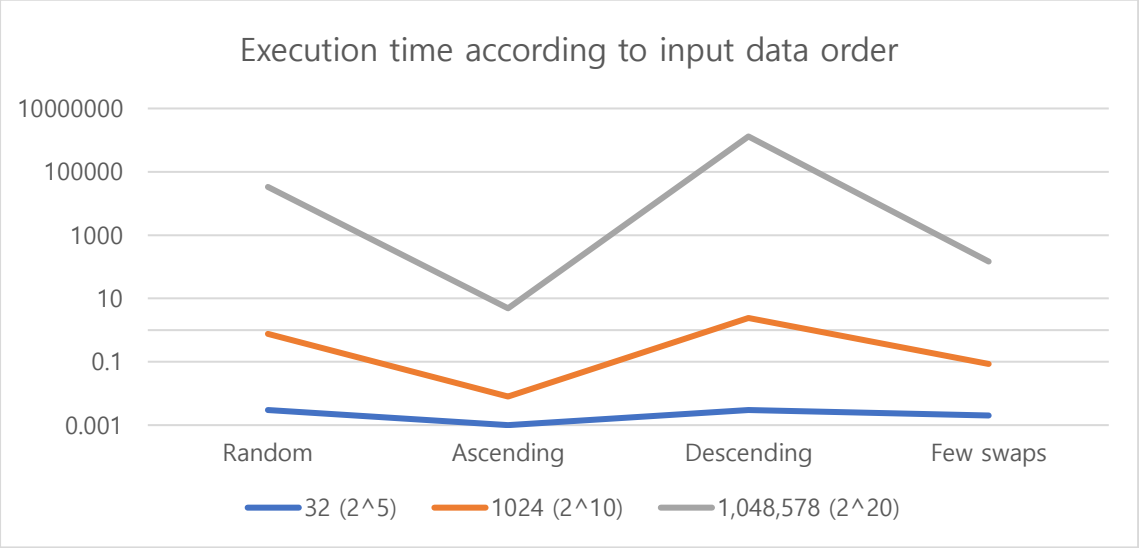
Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
IS	Random	0.003	0.765	33009.000
	Ascending	0.001	0.008	4.888
	Descending	0.003	2.418	1307699.375
	Few swaps	0.002	0.086	145.419

합병 정렬은 하나의 검사 대상과 그 검사 대상의 앞에 있는 원소들과 차례로 비교해가며 정렬하므로, 뒤의 원소가 앞의 원소보다 항상 같거나 큰 ascending order 에서는 가장 짧은 실행 시간이 나오고 반대로 뒤의 원소가 앞의 원소보다 항상 같거나 작은 descending order 에서는 가장 긴 실행 시간이 나오는 것을 위의 실험 결과에서도 확인할 수 있다.

4) 실험 결과 정리 그래프



이론적인 시간 복잡도와 마찬가지로, 입력한 데이터의 개수 증가폭보다 실행 시간의 증가폭이 훨씬 큰 것으로 나타난다. n^2 의 그래프와 유사하게 아래로 볼록인 것을 확인할 수 있다.

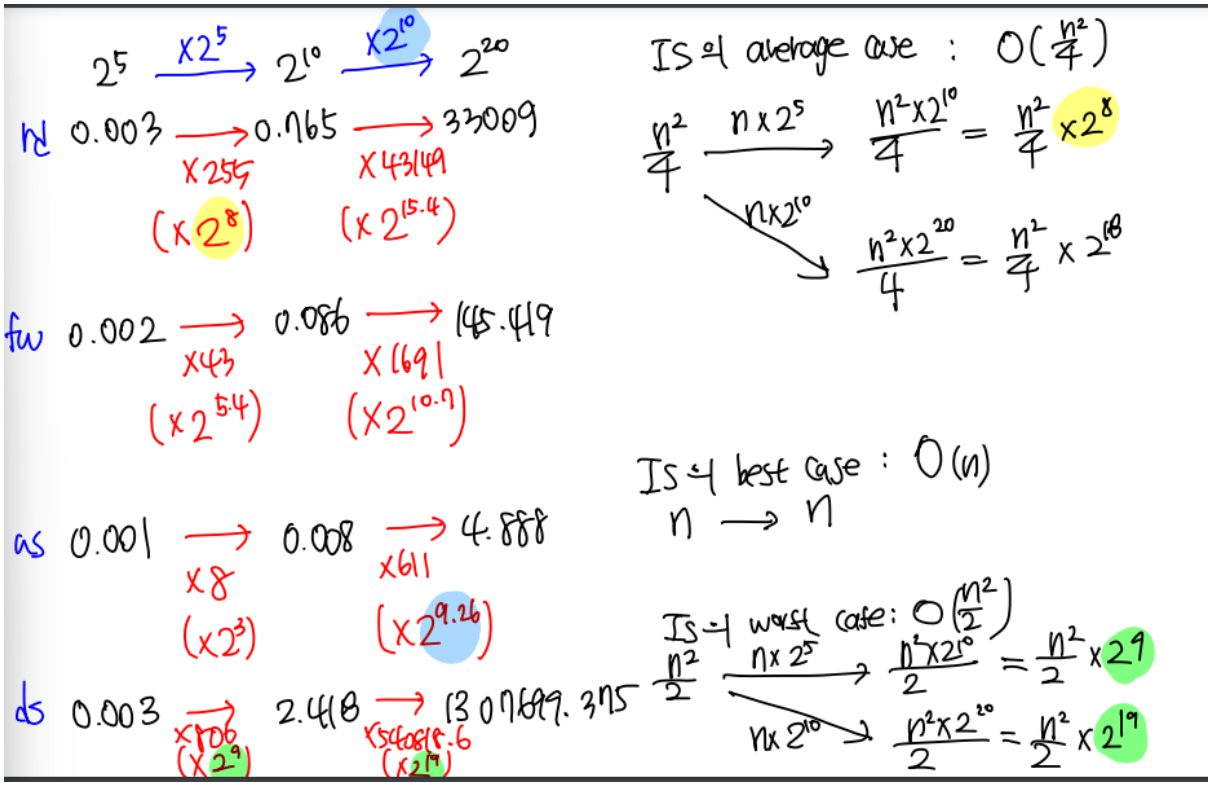


Ascending 이 가장 실행시간이 짧고, Descending 이 가장 실행시간이 긴 것을 확인할 수 있다. random 과 few swaps 은 그 둘 사이에 존재하는 값으로 average case 에 속한다는 것도 알 수 있다.

5) 이론과 실험 결과 분석

	Big-O	Example of input data order
Worst case	O((n^2)/2) = O(n^2)	Descending
Average case	O((n^2)/4) = O(n^2)	Random, Few swaps
Best case	O(n)	Ascending

4 가지 input data order 별로 각각 worst case, average case, best case 에 속한다는 것을 앞선 그래프에서 확인하였다. 각 4 가지 data order 별로 이론적인 시간 복잡도와 실제 실험 결과와 유사한지 수학적으로 계산하였다. 계산한 과정은 아래에 pdf 로 첨부하였다. 계산 결과, worst case 인 경우 이론적인 시간 복잡도와 거의 유사하게 (n^2)/2 인 것을 확인할 수 있었다. best case 의 경우 이론보다는 덜 걸리는 것으로, average case 는 worst case 만큼은 아니지만 대체적으로 이론과 비슷하다는 것을 확인할 수 있었다. 참고로, 이론과 비슷하게 나온 값은 같은 색의 형광펜으로 칠해놓았다.



(b) MS

1) 이론적인 시간 복잡도

	Big-O	Example of input data order
Worst case	O(n*logn)	Random, Ascending, Descending, Few swaps
Average case	O(n*logn)	Random, Ascending, Descending, Few swaps
Best	O(n*logn)	Random, Ascending, Descending, Few swaps

2) 실험 결과 출력 화면

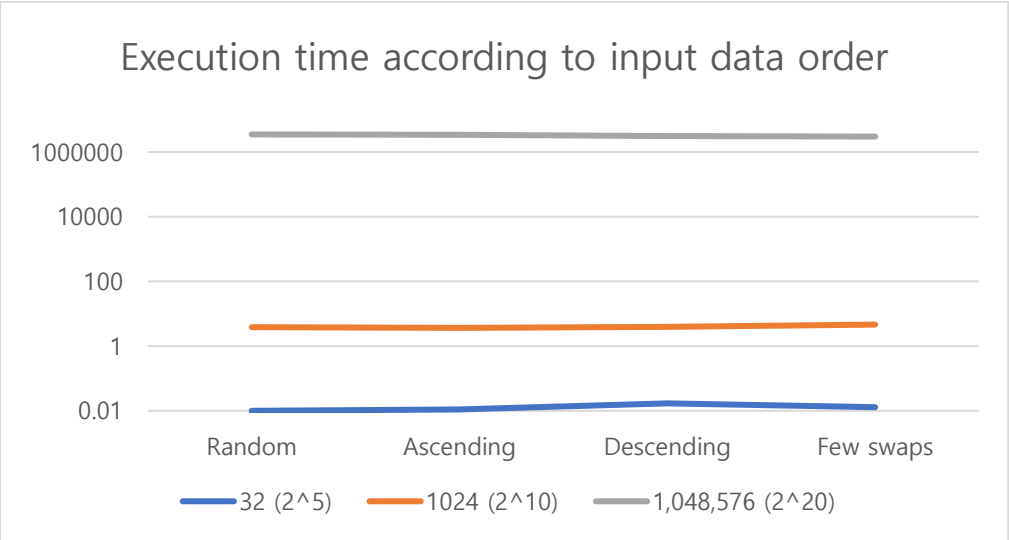
Input data size	Result
32 (=2^5)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 32 elements in random order= 0.037ms ^^^ Time for executing merge sort for 32 elements in random order = 0.010ms ^^^ Time for generating 32 elements in ascending order= 0.002ms ^^^ Time for executing merge sort for 32 elements in ascending order = 0.011ms ^^^ Time for generating 32 elements in descending order= 0.002ms ^^^ Time for executing merge sort for 32 elements in descending order = 0.017ms ^^^ Time for generating 32 elements in few swap order= 0.056ms ^^^ Time for executing merge sort for 32 elements in few swap order = 0.013ms
1024 (=2^10)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.858ms ^^^ Time for executing merge sort for 1024 elements in random order = 3.882ms ^^^ Time for generating 1024 elements in ascending order= 0.225ms ^^^ Time for executing merge sort for 1024 elements in ascending order = 3.659ms ^^^ Time for generating 1024 elements in descending order= 0.232ms ^^^ Time for executing merge sort for 1024 elements in descending order = 3.946ms ^^^ Time for generating 1024 elements in few swap order= 6.028ms ^^^ Time for executing merge sort for 1024 elements in few swap order = 4.622ms

1048576 (=2^20)	^^^ RAND_MAX = 32767 ^^^ Time for generating 1048576 elements in random order= 1162.376ms ^^^ Time for executing merge sort for 1048576 elements in random order = 3495509.500ms ^^^ Time for generating 1048576 elements in ascending order= 9.492ms ^^^ Time for executing merge sort for 1048576 elements in ascending order = 3406883.000ms ^^^ Time for generating 1048576 elements in descending order= 8.989ms ^^^ Time for executing merge sort for 1048576 elements in descending order = 3078886.250ms ^^^ Time for generating 1048576 elements in few swap order= 3869.250ms ^^^ Time for executing merge sort for 1048576 elements in few swap order = 2996757.000ms

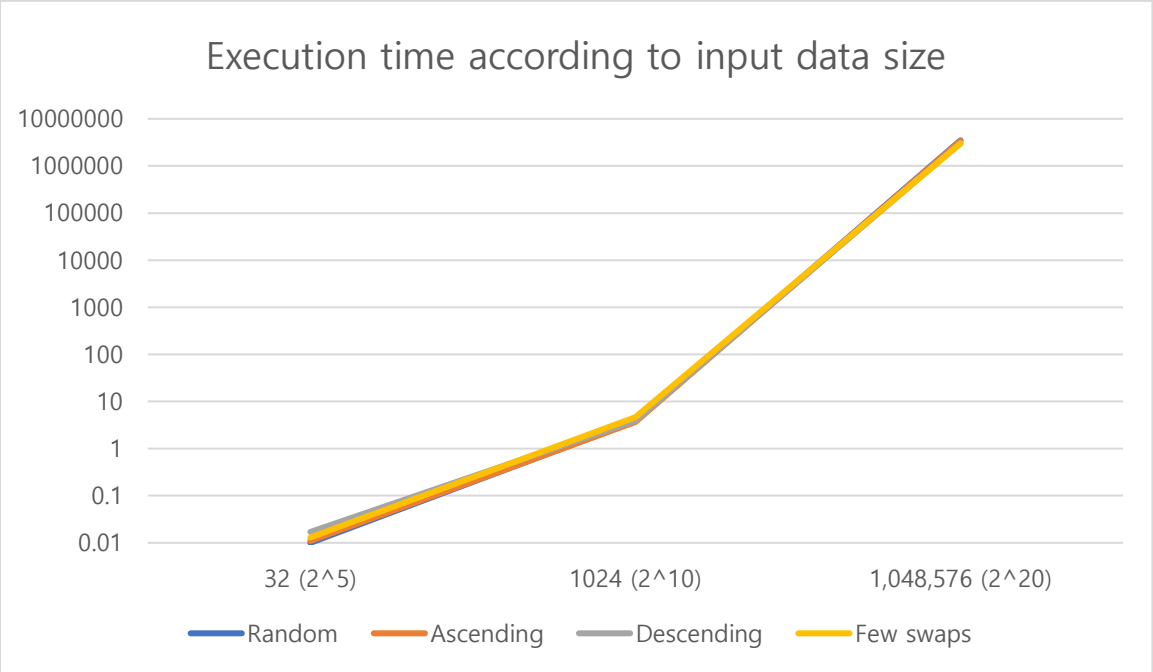
3) 실험 결과 정리 표

Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
MS	Random	0.010	3.882	3495509.500
	Ascending	0.011	3.659	3406883.000
	Descending	0.017	3.946	3078886.250
	Few swaps	0.013	4.622	2996757.000

4) 실험 결과 정리 그래프



위 그래프를 통해 한 눈에 알 수 있듯이, 합병 정렬은 input data order 와는 연관이 거의 없다는 것을 알 수 있다.

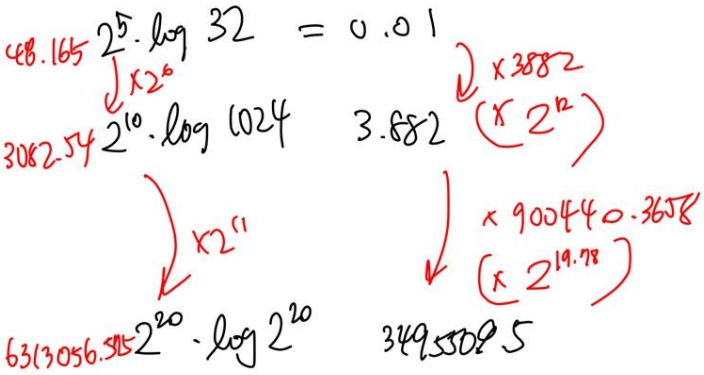


합병 정렬은 input data 를 원소가 1 개가 될 때까지 무조건 나누고, 다시 combine 하는 과정에서 정렬하는 것이므로 실행 시간이 input data order 의 영향을 거의 받지 않는다. 위 그래프에서 확인할 수 있듯이, 합병 정렬은 input data size 의 영향도 거의 안 받는다는 것을 확인할 수 있다.

5) 이론과 실험 결과 분석

	Big-O	Example of input data order
Worst case	$O(n*\log_2 n) = O(n*\log n)$	Random, Ascending, Descending, Few swaps
Average case	$O(n*\log_2 n) = O(n*\log n)$	Random, Ascending, Descending, Few swaps
Best	$O(n*\log_2 n) = O(n*\log n)$	Random, Ascending, Descending, Few swaps

앞서 살펴보았듯이, 합병 정렬은 input data order 의 영향은 거의 안 받으므로, 임의로 random order 인 경우를 골라 이론적인 시간 복잡도와 실험 결과 값을 비교하였다. 그 계산 과정은 아래 사진에 나와 있다. 이로써, 이론적인 시간 복잡도 O(n*logn)은 사실인 것으로 드러났다.



(c) Quick_NAIVE

1) 이론적인 시간 복잡도

	Big-O	Example of input data order
Worst case	O(n^2)	Descending, Ascending
Average case	O(n*logn)	Random, Few swaps
Best case	O(n*logn)	Selected pivot is near to the median of the data

퀵소트에서 worst case 인 경우는 이전 항목에서 서술하였듯이, 피벗 원소를 뽑은 이후 skewed 형태로 덩어리가 나뉠 때이다. 이는 맨 왼쪽 원소를 피벗 원소로 뽑는 Quick_NAIVE 정렬에서는 input data order 가 ascending 과 descending 이 이에 해당하므로 worst case 의 시간 복잡도는 이론적으로 O(n^2)까지 증가하는 것이다.

2) 실험 결과 출력 화면

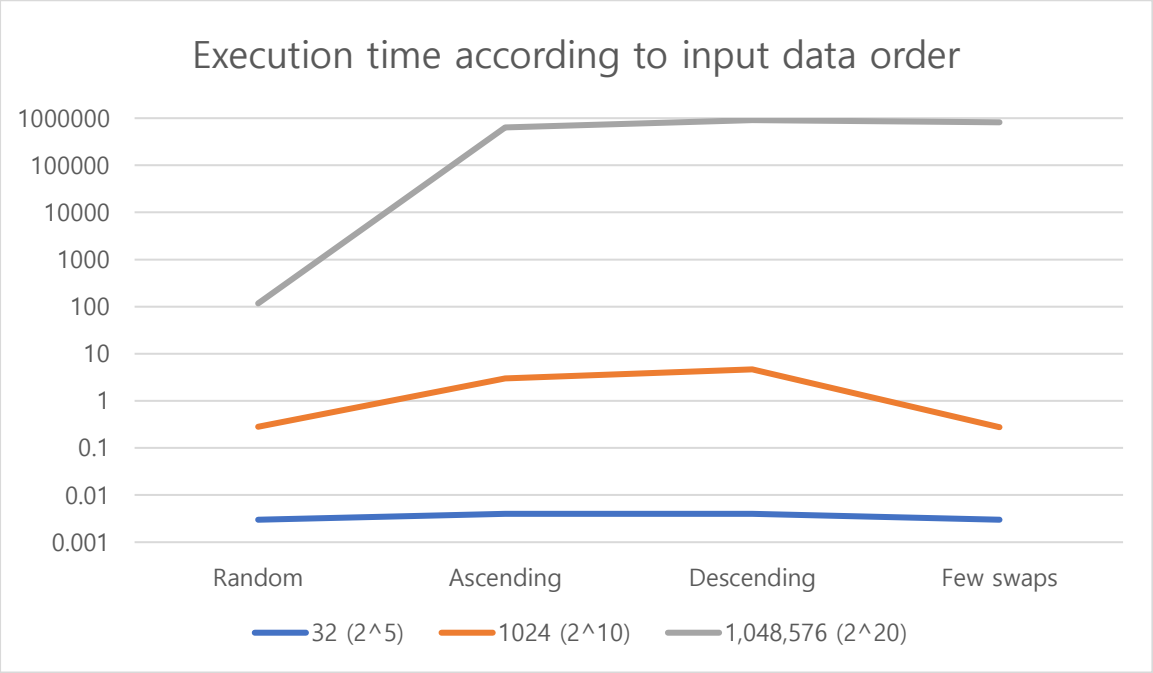
Input data size	Result
32 (=2^5)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 32 elements in random order= 0.018ms ^^^ Time for executing quick sort(NAVIE) for 32 elements in random order = 0.003ms ^^^ Time for generating 32 elements in ascending order= 0.002ms ^^^ Time for executing quick sort(NAIVE) for 32 elements in ascending order = 0.004ms ^^^ Time for generating 32 elements in descending order= 0.002ms ^^^ Time for executing quick sort(NAIVE) for 32 elements in descending order = 0.004ms ^^^ Time for generating 32 elements in few swap order= 0.034ms ^^^ Time for executing quick sort(NAIVE) for 32 elements in few swap order = 0.003ms

1024 (=2^10)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.659ms ^^^ Time for executing quick sort(P) for 1024 elements in random order = 0.284ms ^^^ Time for generating 1024 elements in ascending order= 0.211ms ^^^ Time for executing quick sort(P) for 1024 elements in ascending order = 2.990ms ^^^ Time for generating 1024 elements in descending order= 0.206ms ^^^ Time for executing quick sort(P) for 1024 elements in descending order = 4.662ms ^^^ Time for generating 1024 elements in few swap order= 4.428ms ^^^ Time for executing quick sort(P) for 1024 elements in few swap order = 0.276ms
1048576 (=2^20)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1048576 elements in random order= 198.411ms ^^^ Time for executing quick sort(NAVIE) for 1048576 elements in random order = 117.171ms ^^^ Time for generating 1048576 elements in ascending order= 3.948ms ^^^ Time for executing quick sort(NAIVE) for 1048576 elements in ascending order = 631558.750ms ^^^ Time for generating 1048576 elements in descending order= 6.446ms ^^^ Time for executing quick sort(NAIVE) for 1048576 elements in descending order = 909972.688ms ^^^ Time for generating 1048576 elements in few swap order= 1731.983ms ^^^ Time for executing quick sort(NAIVE) for 1048576 elements in few swap order = 822311.125ms

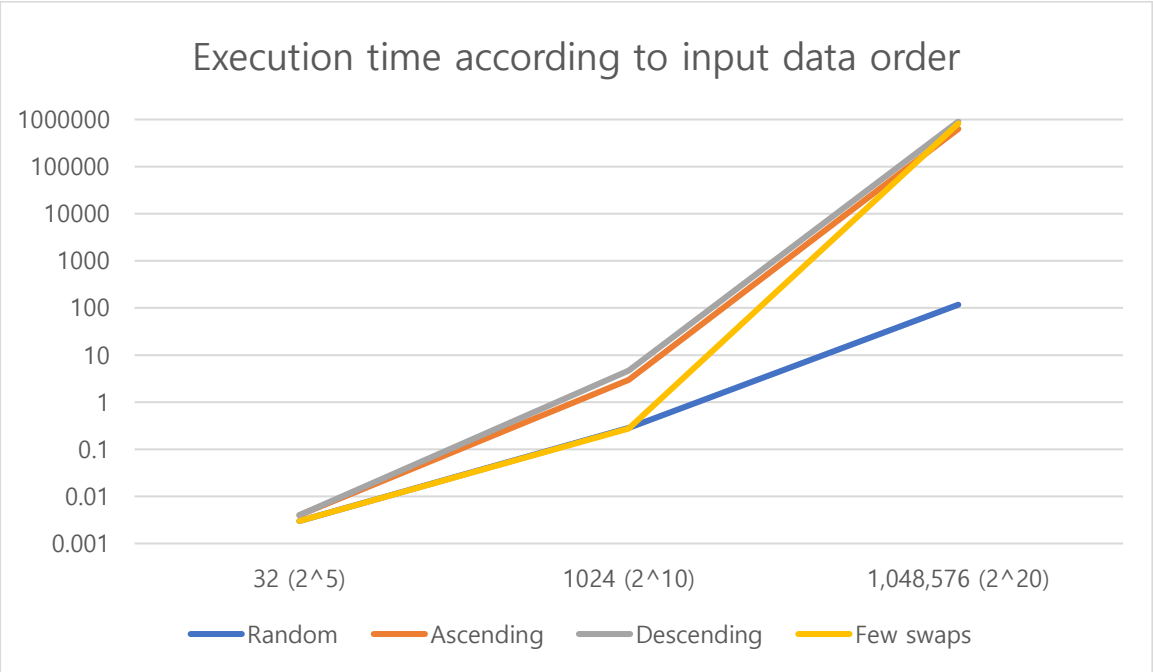
3) 실험 결과 정리 표

Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
Quick_NAIVE	Random	0.003	0.284	117.171
	Ascending	0.004	2.990	631558.750
	Descending	0.004	4.662	909972.688
	Few swaps	0.003	0.276	822311.125

4) 실험 결과 정리 그래프



입력 데이터의 order 가 Random 인 경우, 즉 평균적인 경우일 때는 퀵소트가 이전에 구현한 합병 정렬과 삽입 정렬보다는 훨씬 빠르지만, worst case 에 해당하는 ascending 이나 descending 의 경우에는 실행시간이 급격히 늘어나는 것을 확인할 수 있었다. 바로 이 점이 퀵소트의 최대 단점에 해당한다.



데이터 크기에 따른 실행 시간 차이가 이전에 구현한 다른 정렬들보다는 작다는 것을 위의 그래프를 통해 확인할 수 있다.

5) 이론과 실험 결과 분석

Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
IS	Random	0.003	0.765	33009.000
	Ascending	0.001	0.008	4.888
	Descending	0.003	2.418	1307699.375
	Few swaps	0.002	0.086	145.419
MS	Random	0.010	3.882	3495509.500
	Ascending	0.011	3.659	3406883.000
	Descending	0.017	3.946	3078886.250
	Few swaps	0.013	4.622	2996757.000
Quick _NAIVE	Random	0.003	0.284	117.171
	Ascending	0.004	2.990	631558.750
	Descending	0.004	4.662	909972.688
	Few swaps	0.003	0.276	822311.125

위의 표에서도 확인할 수 있듯이, 다른 정렬들보다 퀵소트는 이름 그대로 정렬속도가 빠르다는 특징이 있다. 하지만, 이 특징은 best 나 average case 에만 해당하고, worst case 인 ascending 이나 descending 의 경우에는 정렬을 진행하는 과정에서 skewed 형태로 데이터들이 나뉘면서 정렬되기 때문에 시간 복잡도가 O(n^2)까지 증가한다는 결정적인 단점이 있다. 특히, input data order 가 ascending 인 경우 insertion sort 가 quicksort 보다 훨씬 빠른 것을 위의 표에서도 확인이 가능하다. 이처럼 input data 가 ascending 에 가까울수록 삽입 정렬이 퀵소트보다 시간 복잡도의 측면에서 더 우월하다는 것을 실험에서도 확인할 수 있었다. 이러한 퀵소트의 단점을 개선하는 데 보통 크게 3 가지 방법이 있는데, 다음 항목부터는 이 퀵소트를 개선하는 데 중점을 두고 실험을 진행할 예정이다.

(d) Quick_P

이번에 구현할 퀵소트는 이전에 2 번 항목에서 설명했던 median-of-three 방식을 이용하여 피벗 원소를 뽑는 식으로 개선되었다. NAIVE 버전보다 개선된 것이 맞는지 실험에서 확인하고자 한다.

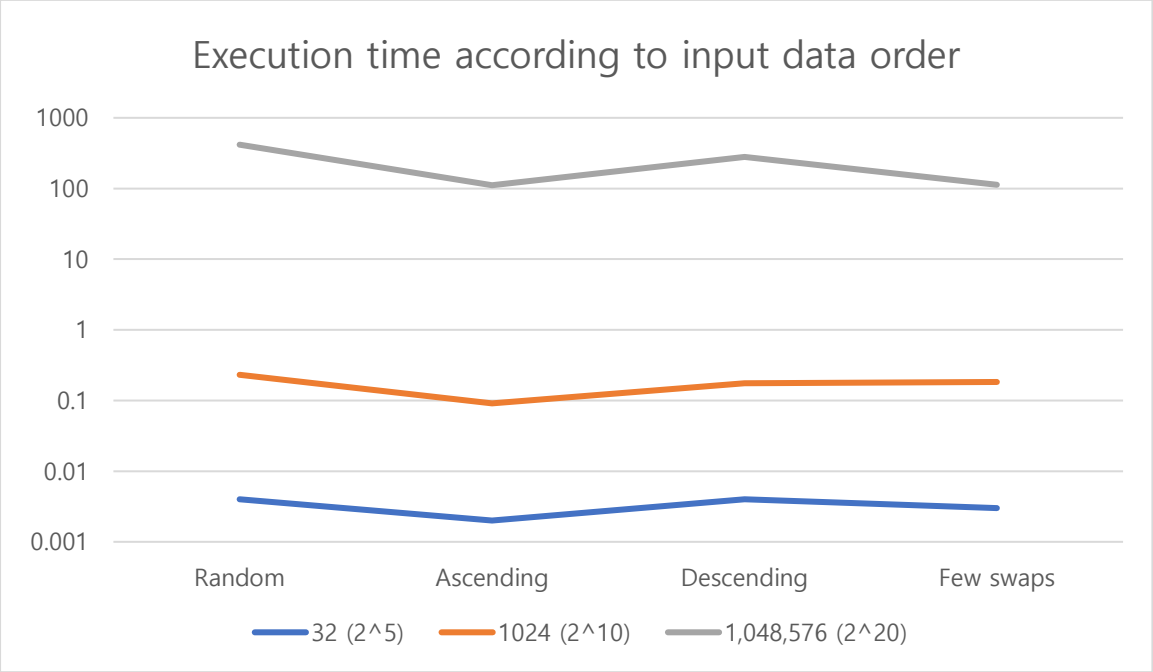
1) 실험 결과 출력 화면

Input data size	Result
32 (=2^5)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 32 elements in random order= 0.012ms ^^^ Time for executing quick sort(P) for 32 elements in random order = 0.004ms ^^^ Time for generating 32 elements in ascending order= 0.001ms ^^^ Time for executing quick sort(P) for 32 elements in ascending order = 0.002ms ^^^ Time for generating 32 elements in descending order= 0.001ms ^^^ Time for executing quick sort(P) for 32 elements in descending order = 0.004ms ^^^ Time for generating 32 elements in few swap order= 0.036ms ^^^ Time for executing quick sort(P) for 32 elements in few swap order = 0.003ms
1024 (=2^10)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.737ms ^^^ Time for executing quick sort(P) for 1024 elements in random order = 0.231ms ^^^ Time for generating 1024 elements in ascending order= 0.195ms ^^^ Time for executing quick sort(P) for 1024 elements in ascending order = 0.091ms ^^^ Time for generating 1024 elements in descending order= 0.215ms ^^^ Time for executing quick sort(P) for 1024 elements in descending order = 0.176ms ^^^ Time for generating 1024 elements in few swap order= 4.808ms ^^^ Time for executing quick sort(P) for 1024 elements in few swap order = 0.183ms
1048576 (=2^20)	^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1048576 elements in random order= 352.890ms ^^^ Time for executing quick sort(P) for 1048576 elements in random order = 416.791ms ^^^ Time for generating 1048576 elements in ascending order= 5.001ms ^^^ Time for executing quick sort(P) for 1048576 elements in ascending order = 111.181ms ^^^ Time for generating 1048576 elements in descending order= 6.403ms ^^^ Time for executing quick sort(P) for 1048576 elements in descending order = 279.841ms ^^^ Time for generating 1048576 elements in few swap order= 3223.401ms ^^^ Time for executing quick sort(P) for 1048576 elements in few swap order = 112.574ms

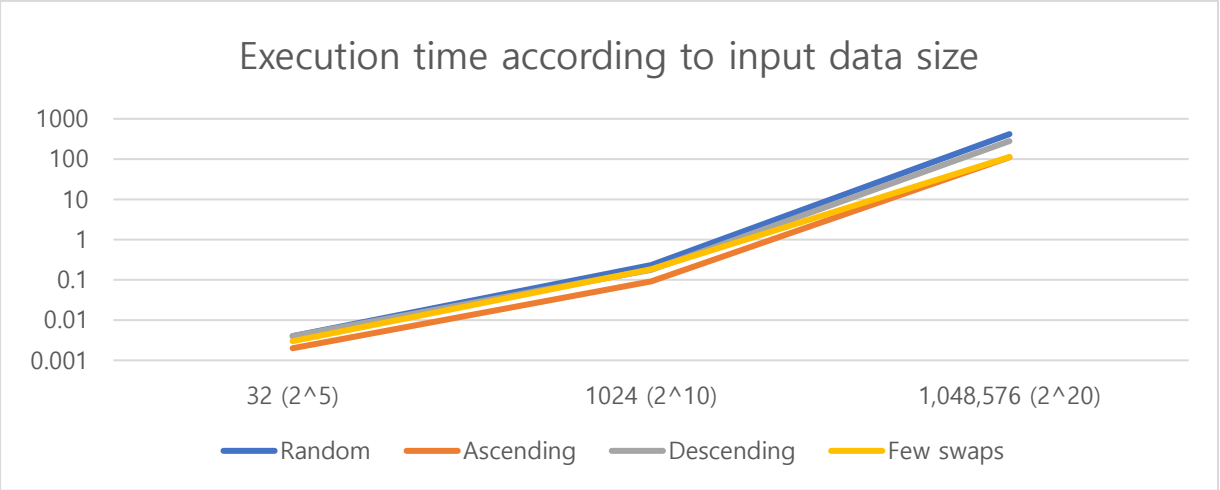
2) 실험 결과 정리 표

Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
Quick_P	Random	0.004	0.231	416.791
	Ascending	0.002	0.091	111.181
	Descending	0.004	0.176	279.841
	Few swaps	0.003	0.183	112.574

3) 실험 결과 정리 그래프



이전의 NAIVE 버전에서는 ascending 과 descending 에서 극명하게 시간 복잡도가 증가하는 것을 확인할 수 있었는데, P 버전에서는 ascending 과 descending 도 다른 input data order 들과 극명한 차이를 보이지는 않는 것을 확인할 수 있다. 퀵소트의 단점을 어느 정도 해결했다고 볼 수 있는 것이다.

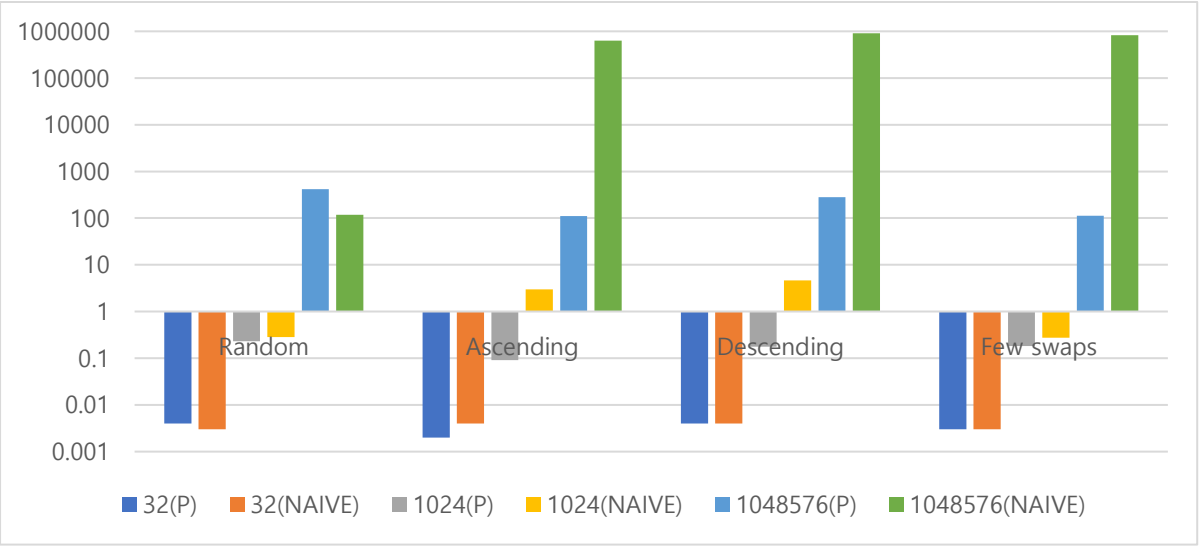


데이터 크기에 따른 실행 시간 차이도 NAIVE 버전보다도 많이 줄어든 것을 위 그래프를 통해 육안으로 확인이 가능하다.

4) 이론과 실험 결과 분석

Quick _NAIVE	Random	0.003	0.284	117.171
	Ascending	0.004	2.990	631558.750
	Descending	0.004	4.662	909972.688
	Few swaps	0.003	0.276	822311.125
Quick_P	Random	0.004	0.231	416.791
	Ascending	0.002	0.091	111.181
	Descending	0.004	0.176	279.841
	Few swaps	0.003	0.183	112.574

input data size 가 32 와 같이 다소 작은 경우에는 NAIVE 와 P 버전이 큰 차이를 확인하기 어렵다. 하지만, input data size 가 커지면 커질수록 두 버전 간의 실행 시간 차이 폭이 점점 증가하는 것을 확인할 수 있다. 시각적인 확인을 위해 그래프로 표현하면 아래와 같다.



(e) Quick_PIS

0) 적절한 M 사이즈 결정

실험한 N_ELEMENTS 의 크기는 1024 이므로, 그것의 절반인 512 부터 시작하였다. 이후로는 한 단계 넘어갈 때마다 M 사이즈를 나누기 4 해주다가 실행시간이 줄어드는 지점에서 점점 줄이는 식으로 진행하였다. 그리고 적어도 M 이 0 인 때보다는 실행시간이 짧아야 한다. 실험 결과는 아래 표와 같이 나왔다. 값이 감소하다가 다시 증가하는 구간을 위주로 탐색한 결과, M=15 인 경우가 가장 실행시간이 짧은 것으로 나왔다. 이후 실험에서도 M 의 값을 15 로 설정하고 진행하였다.

'M'	Result
0	Size of 'M': 0 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.732ms ^^^ Time for executing quick sort(PIS) for 1024 elements in random order = 0.249ms ^^^ Time for generating 1024 elements in ascending order= 0.005ms ^^^ Time for executing quick sort(PIS) for 1024 elements in ascending order = 0.173ms ^^^ Time for generating 1024 elements in descending order= 0.004ms ^^^ Time for executing quick sort(PIS) for 1024 elements in descending order = 0.203ms ^^^ Time for generating 1024 elements in few swaps order= 6.078ms ^^^ Time for executing quick sort(PIS) for 1024 elements in few swaps order = 0.320ms
512	Size of 'M': 512 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.646ms ^^^ Time for executing quick sort(PIS) for 1024 elements in random order = 0.410ms ^^^ Time for generating 1024 elements in ascending order= 0.005ms ^^^ Time for executing quick sort(PIS) for 1024 elements in ascending order = 0.013ms ^^^ Time for generating 1024 elements in descending order= 0.004ms ^^^ Time for executing quick sort(PIS) for 1024 elements in descending order = 0.809ms ^^^ Time for generating 1024 elements in few swaps order= 5.926ms ^^^ Time for executing quick sort(PIS) for 1024 elements in few swaps order = 0.079ms

128	Size of 'M': 128 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.615ms ^^^ Time for executing quick sort(PIS) for 1024 elements in random order = 0.173ms ^^^ Time for generating 1024 elements in ascending order= 0.005ms ^^^ Time for executing quick sort(PIS) for 1024 elements in ascending order = 0.022ms ^^^ Time for generating 1024 elements in descending order= 0.004ms ^^^ Time for executing quick sort(PIS) for 1024 elements in descending order = 0.305ms ^^^ Time for generating 1024 elements in few swaps order= 5.218ms ^^^ Time for executing quick sort(PIS) for 1024 elements in few swaps order = 0.086ms
32	Size of 'M': 32 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 1.510ms ^^^ Time for executing quick sort(PIS) for 1024 elements in random order = 0.315ms ^^^ Time for generating 1024 elements in ascending order= 0.010ms ^^^ Time for executing quick sort(PIS) for 1024 elements in ascending order = 0.077ms ^^^ Time for generating 1024 elements in descending order= 0.010ms ^^^ Time for executing quick sort(PIS) for 1024 elements in descending order = 0.348ms ^^^ Time for generating 1024 elements in few swaps order= 13.467ms ^^^ Time for executing quick sort(PIS) for 1024 elements in few swaps order = 0.482ms
15	Size of 'M': 15 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.261ms ^^^ Time for executing quick sort(PIS) for 1024 elements in random order = 0.074ms ^^^ Time for generating 1024 elements in ascending order= 0.001ms ^^^ Time for executing quick sort(PIS) for 1024 elements in ascending order = 0.015ms ^^^ Time for generating 1024 elements in descending order= 0.001ms ^^^ Time for executing quick sort(PIS) for 1024 elements in descending order = 0.033ms ^^^ Time for generating 1024 elements in few swaps order= 1.612ms ^^^ Time for executing quick sort(PIS) for 1024 elements in few swaps order = 0.058ms

10	Size of 'M': 10 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.951ms ^^^ Time for executing quick sort(PIS) for 1024 elements in random order = 0.195ms ^^^ Time for generating 1024 elements in ascending order= 0.006ms ^^^ Time for executing quick sort(PIS) for 1024 elements in ascending order = 0.071ms ^^^ Time for generating 1024 elements in descending order= 0.005ms ^^^ Time for executing quick sort(PIS) for 1024 elements in descending order = 0.149ms ^^^ Time for generating 1024 elements in few swaps order= 8.491ms ^^^ Time for executing quick sort(PIS) for 1024 elements in few swaps order = 0.188ms

1) 실험 결과 출력 화면

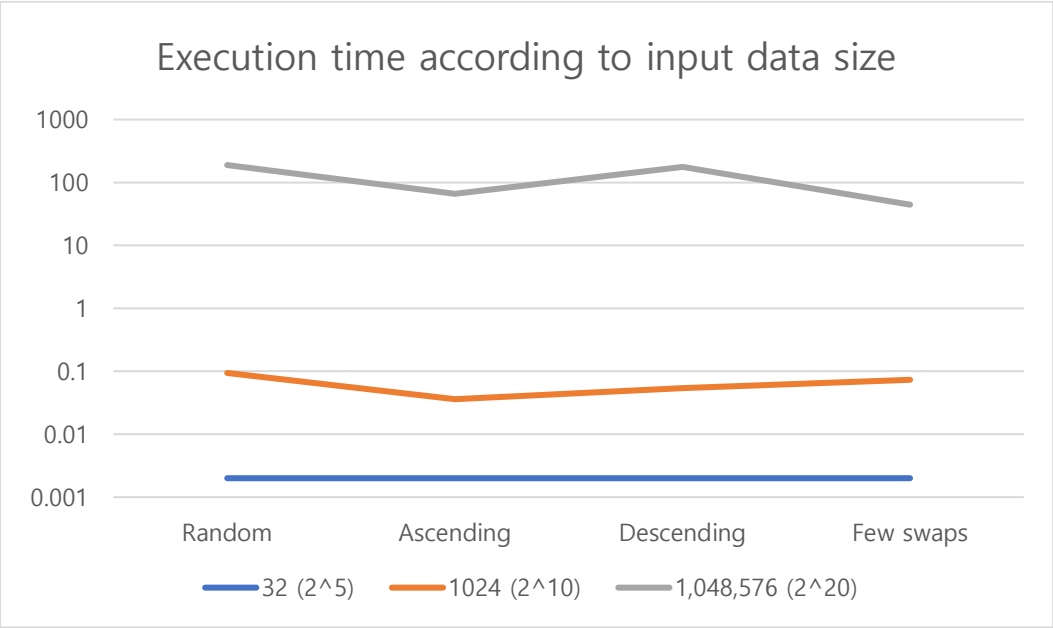
Input data size	Result
32 (=2^5)	Size of 'M': 15 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 32 elements in random order= 0.011ms ^^^ Time for executing quick sort(PIS) for 32 elements in random order = 0.002ms ^^^ Time for generating 32 elements in ascending order= 0.000ms ^^^ Time for executing quick sort(PIS) for 32 elements in ascending order = 0.002ms ^^^ Time for generating 32 elements in descending order= 0.000ms ^^^ Time for executing quick sort(PIS) for 32 elements in descending order = 0.002ms ^^^ Time for generating 32 elements in few swaps order= 0.022ms ^^^ Time for executing quick sort(PIS) for 32 elements in few swaps order = 0.002ms

1024 (=2^10)	Size of 'M': 15 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in few swaps order= 0.219ms ^^^ Time for executing quick sort(PIS) for 1024 elements in random order = 0.094ms ^^^ Time for generating 1024 elements in few swaps order= 0.001ms ^^^ Time for executing quick sort(PIS) for 1024 elements in ascending order = 0.036ms ^^^ Time for generating 1024 elements in few swaps order= 0.002ms ^^^ Time for executing quick sort(PIS) for 1024 elements in descending order = 0.054ms ^^^ Time for generating 1024 elements in few swaps order= 2.151ms ^^^ Time for executing quick sort(PIS) for 1024 elements in few swaps order = 0.073ms
1048576 (=2^20)	Size of 'M': 15 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1048576 elements in few swaps order= 217.705ms ^^^ Time for executing quick sort(PIS) for 1048576 elements in random order = 189.072ms ^^^ Time for generating 1048576 elements in few swaps order= 0.626ms ^^^ Time for executing quick sort(PIS) for 1048576 elements in ascending order = 66.055ms ^^^ Time for generating 1048576 elements in few swaps order= 0.808ms ^^^ Time for executing quick sort(PIS) for 1048576 elements in descending order = 176.066ms ^^^ Time for generating 1048576 elements in few swaps order= 1195.321ms ^^^ Time for executing quick sort(PIS) for 1048576 elements in few swaps order = 44.446ms

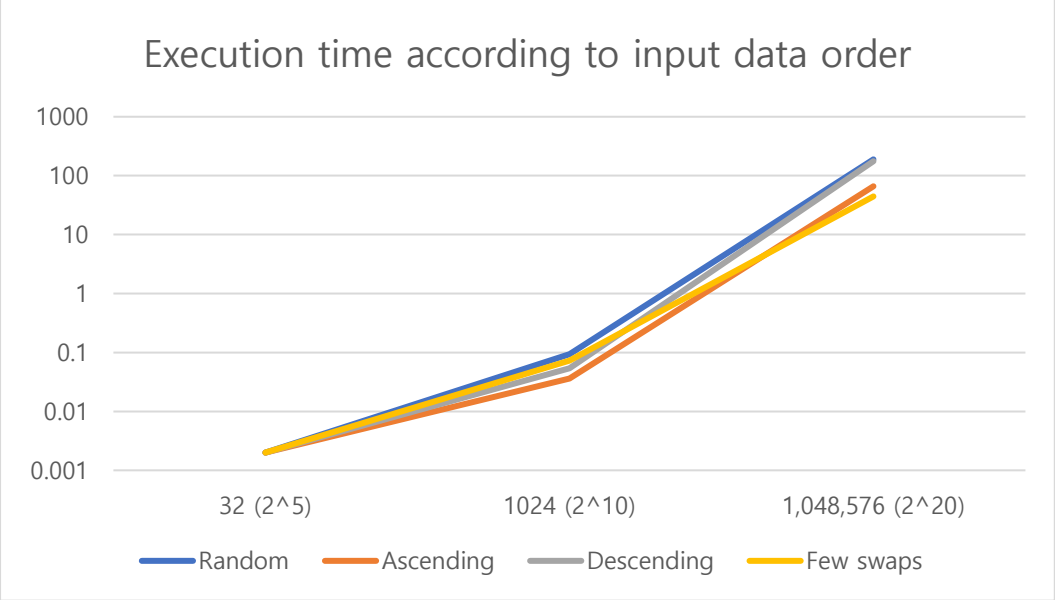
2) 실험 결과 정리 표

Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
Quick_PIS	Random	0.002	0.094	189.072
	Ascending	0.002	0.036	66.055
	Descending	0.002	0.054	176.066
	Few swaps	0.002	0.073	44.446

3) 실험 결과 정리 그래프

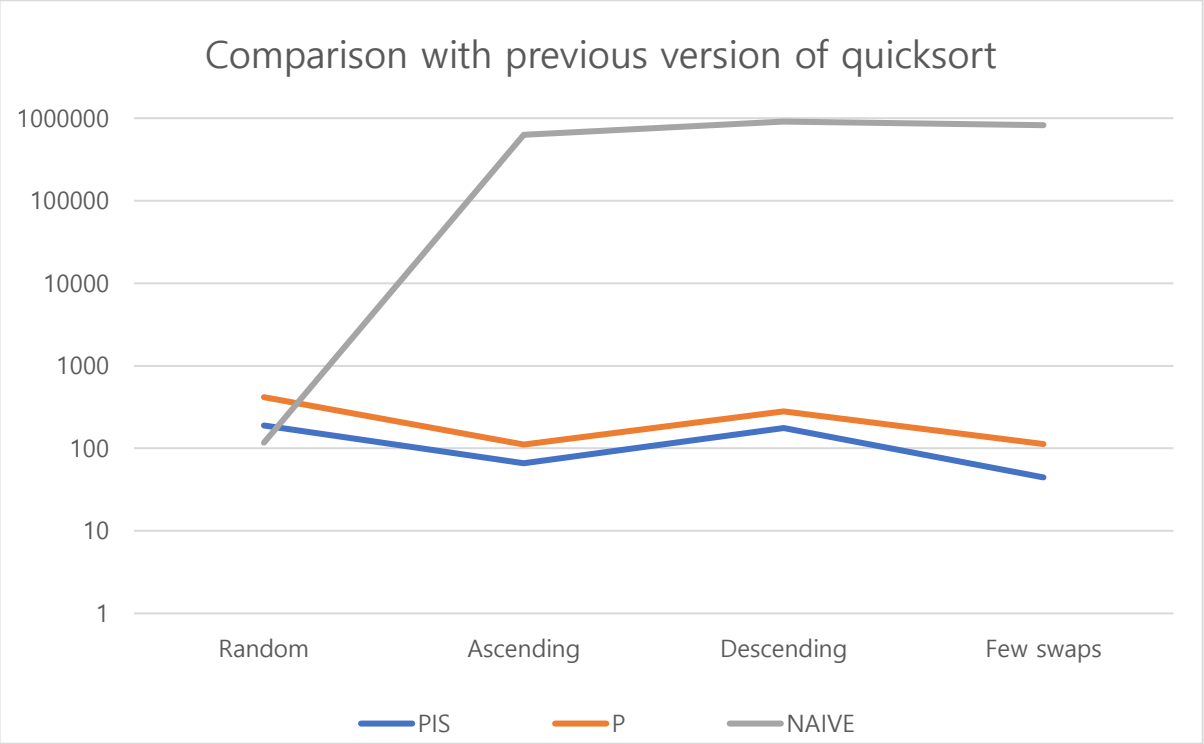


입력 크기가 32 정도로 작으면 input data order 별로 차이가 크지는 않다.

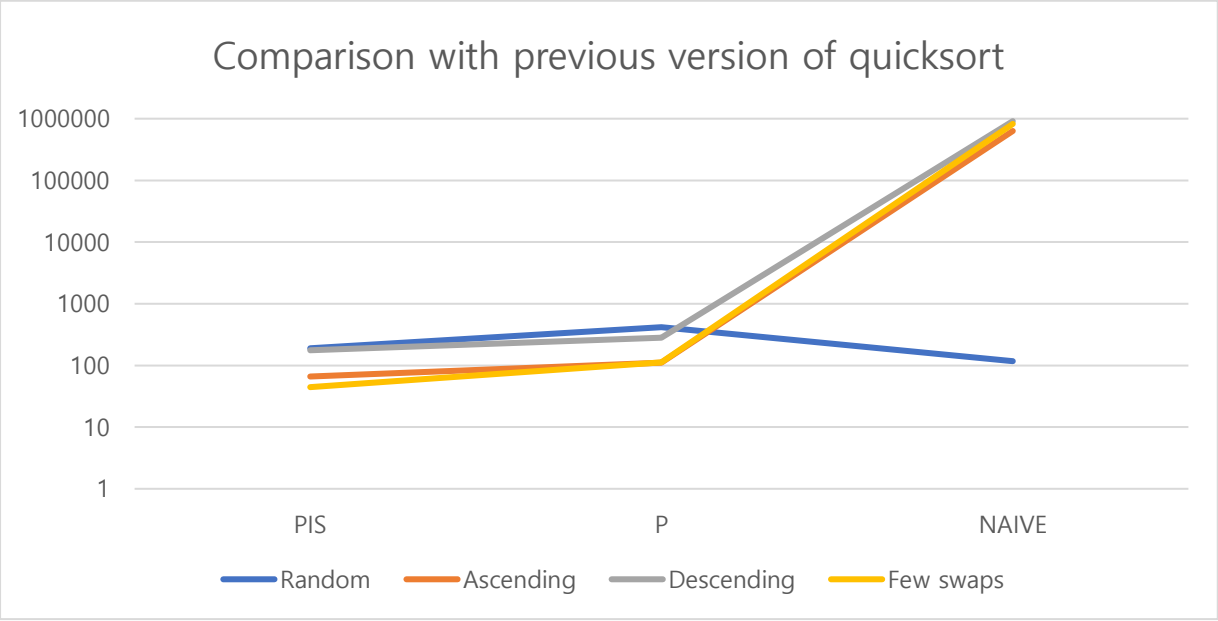


이렇게 PIS 전만 따로 보면 이전의 P 버전과 큰 차이를 느끼기 어렵다. 다음 페이지에서 다른 버전의 퀵소트들과 비교할 예정이다.

4) 이전 퀵소트들과의 비교



왼쪽의 그래프는 input data order (n 은 2^20 으로 고정)의 관점에서 바라본 퀵소트 세 버전의 차이이다. 이 그래프에서도 확인할 수 있듯이, NAIVE 버전이 가장 높은 시간 복잡도를 지니고, P 버전과 PIS 버전은 그보다는 작은 차이지만 PIS 버전이 더 낮은 실행 시간을 지님을 알 수 있다. 밑의 그래프는 이전 그래프를 x 와 y 축을 바꾼 관점에서 기록한 퀵소트 세 버전의 차이이다. 확실히 PIS 가 거의 모든 input data order 에서 가장 작은 시간 복잡도를 지닌다는 점이 확인 가능하다.



(f) Quick_PISTRO

이번에는 피벗 원소 선정 이후, 덩어리가 큰 쪽은 iteration 으로 처리하고, 작은 쪽은 recursion 으로 처리함으로써 system stack overflow 를 방지할 수 있는 PISTRO 버전을 구현하였다.

1) 실험 결과 출력 화면

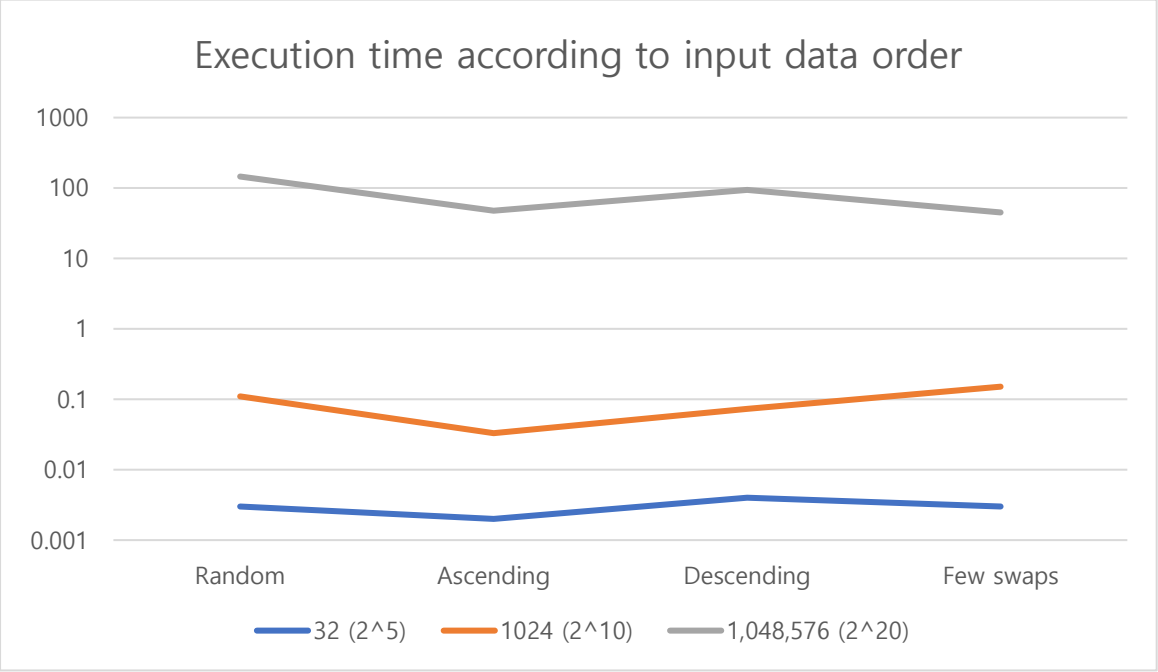
Input data size	Result
32 (=2^5)	Value of 'M': 15 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 32 elements in random order= 0.012ms ^^^ Time for executing quick sort(PISTRO) for 32 elements in random order = 0.003ms ^^^ Time for generating 32 elements in ascending order= 0.000ms ^^^ Time for executing quick sort(PISTRO) for 32 elements in ascending order = 0.002ms ^^^ Time for generating 32 elements in descending order= 0.000ms ^^^ Time for executing quick sort(PISTRO) for 32 elements in descending order = 0.004ms ^^^ Time for generating 32 elements in few swaps order= 0.017ms ^^^ Time for executing quick sort(PISTRO) for 32 elements in few swaps order = 0.003ms
1024 (=2^10)	Value of 'M': 15 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1024 elements in random order= 0.211ms ^^^ Time for executing quick sort(PISTRO) for 1024 elements in random order = 0.110ms ^^^ Time for generating 1024 elements in ascending order= 0.001ms ^^^ Time for executing quick sort(PISTRO) for 1024 elements in ascending order = 0.033ms ^^^ Time for generating 1024 elements in descending order= 0.002ms ^^^ Time for executing quick sort(PISTRO) for 1024 elements in descending order = 0.073ms ^^^ Time for generating 1024 elements in few swaps order= 1.474ms ^^^ Time for executing quick sort(PISTRO) for 1024 elements in few swaps order = 0.151ms

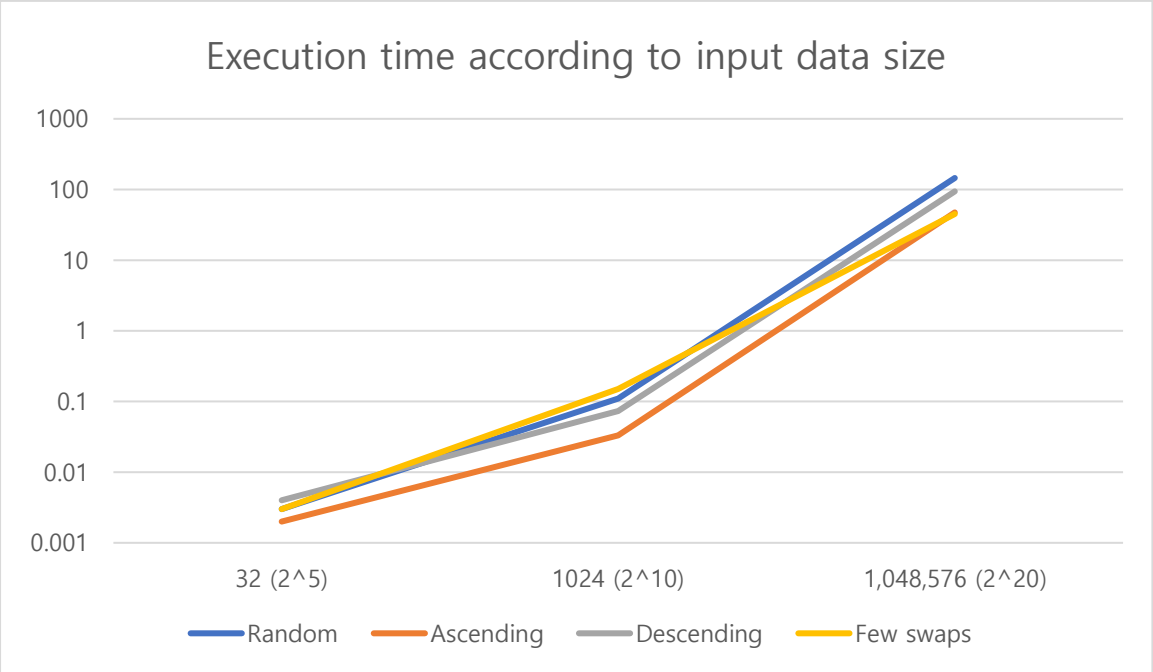
1048576 (=2^20)	Value of 'M': 15 ^^^ UNIT_MAX = 4294967295 ^^^ RAND_MAX = 32767 ^^^ Time for generating 1048576 elements in random order= 287.513ms ^^^ Time for executing quick sort(PISTRO) for 1048576 elements in random order = 145.507ms ^^^ Time for generating 1048576 elements in ascending order= 0.592ms ^^^ Time for executing quick sort(PISTRO) for 1048576 elements in ascending order = 47.344ms ^^^ Time for generating 1048576 elements in descending order= 0.857ms ^^^ Time for executing quick sort(PISTRO) for 1048576 elements in descending order = 94.028ms ^^^ Time for generating 1048576 elements in few swaps order= 1394.401ms ^^^ Time for executing quick sort(PISTRO) for 1048576 elements in few swaps order = 44.967ms

2) 실험 결과 정리 표

Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
Quick_PISTRO	Random	0.003	0.110	145.507
	Ascending	0.002	0.033	47.344
	Descending	0.004	0.073	94.028
	Few swaps	0.003	0.151	44.967

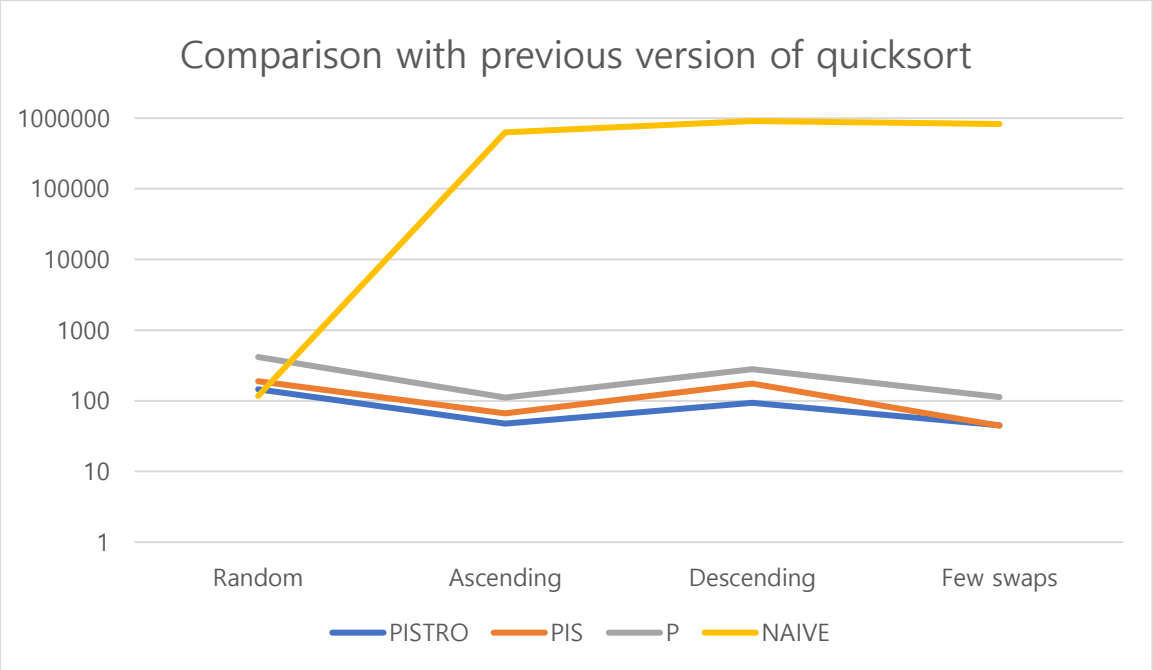
3) 실험 결과 정리 그래프



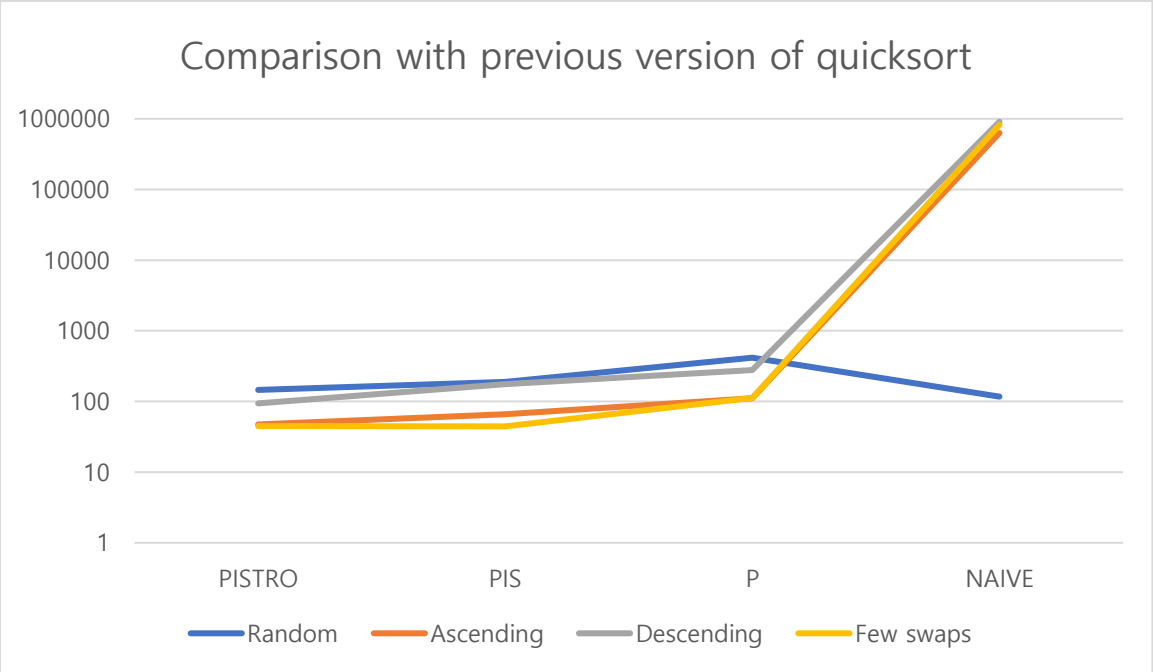


전반적으로 PIS 버전과 유사한 형태로 그래프가 그려졌다. 하지만, 실행시간의 정확한 수치는 PIS 보다 PISTRO 가 더 작았다. 이로써 PIS 보다 PISTRO 가 시간 복잡도의 측면에서 더 우월하다는 것을 이번 실험으로 직접 확인할 수 있었다.

4) 이전 퀵소트들과의 비교

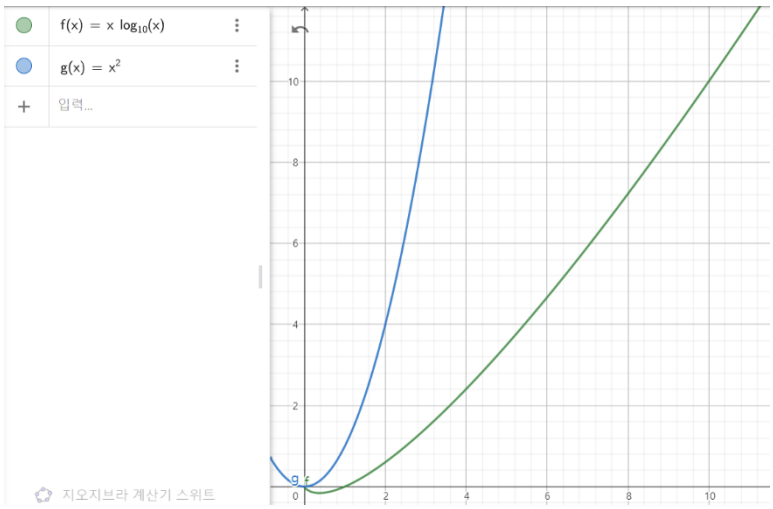


이제껏 구현한 퀵소트정렬은 총 4 가지였다. 확실히 퀵소트의 단점을 보완하는 방향으로 개선하면 기존의 NAIVE 버전보다 시간 복잡도의 측면에서 더 나아질 수 있다는 것을 확인할 수 있다.



위의 그래프와 같은 그래프지만, x 축과 y 축만 바꿔 표현한 것이다. PISTRO 에서 NAIVE 로 갈수록 그래프가 우상향하는 것으로 보아, 퀵소트 정렬이 개선되면 개선될수록 시간 복잡도가 줄어든다는 것을 확인할 수 있다.

4. 심화 분석



이론적으로 접근해보면, 일반적으로 insertion sort 는 O(n^2)의 시간 복잡도를 지니고, quick sort 는 O(n*logn)의 시간 복잡도를 지닌다. f(x) = x*logx, g(x)=x^2 인 두 그래프를 '지오지브라' 라는 그래프 프로그램을 통해 그린 다음 비교해보면 위의 사진과 같다. 위의 그래프에서 확인할 수 있듯이, x 의 값이 증가할수록 두 그래프 간의 차이도 커진다. 이에 의하면, x 의 값이 작을수록 두 그래프 간의 차이가 줄어든다는 것인데, 시간 복잡도의 측면에서 과연 어느 정도의 x 까지 두 그래프 간의 차이가 없다고 봐도 괜찮은 것인지를 중점으로 실험을 설계하였다.

해당 질문을 해결하기 위해 실험에 사용할 input data order 를 삽입 정렬과 퀵정렬에서 모두 worst case 에 속하는 descending order 로 고정시켜 놓고 실험을 진행하였다. 앞에서 실행한 실험들은 input data size 가 3 가지였다. 그 결과를 아래 표로 정리하였다. 32 까지는 비슷한데 1024 부터는 확실히 퀵정렬이 빠르다. 이에 따라 32 에서 차례로 input data size 를 줄여가며 두 정렬의 실행 시간이 비슷해지는 시점을 실험해보았다.

[이전 실험 결과 정리 표]

Sorting function	**Execution time according to input data size (N_ELEMENTS)**		
	32 (2^5)	**1024 (2^10)**	**1,048,576 (2^20)**
IS	0.003	2.418	1307699.375
Quick_P	0.004	0.176	279.841

[실험 정리 표]

Sorting function	**Execution time according to input data size (N_ELEMENTS)**			
	8	**15**	**16**	**32**
IS	0.001	0.001	0.001	0.003
Quick_P	0.001	0.001	0.002	0.004

실험 결과를 위의 표로 정리하였다. 16 보다 작으면 IS 와 Quick_P 는 실행 시간에 차이가 거의 없는 것으로 나왔다. 따라서 input data size 가 15 까지는 굳이 퀵정렬을 사용할 필요없이 삽입정렬을 사용해도 크게 문제가 되지 않는다는 것을 유추해낼 수 있었다. 이 값은 이전 문항에서 Quick_PIS 를 구현할 때 구한 M 의 값과 동일하다는 것도 확인할 수 있다.

[실험 최종 결과]

Sorting function	Order of input data	Execution time according to input data size (N_ELEMENTS)		
		32 (2^5)	1024 (2^10)	1,048,576 (2^20)
IS	Random	0.003	0.765	33009.000
	Ascending	0.001	0.008	4.888
	Descending	0.003	2.418	1307699.375
	Few swaps	0.002	0.086	145.419
MS	Random	0.010	3.882	3495509.500
	Ascending	0.011	3.659	3406883.000
	Descending	0.017	3.946	3078886.250
	Few swaps	0.013	4.622	2996757.000
Quick _NAIVE	Random	0.003	0.284	117.171
	Ascending	0.004	2.990	631558.750
	Descending	0.004	4.662	909972.688
	Few swaps	0.003	0.276	822311.125
Quick_P	Random	0.004	0.231	416.791
	Ascending	0.002	0.091	111.181
	Descending	0.004	0.176	279.841
	Few swaps	0.003	0.183	112.574
Quick_PIS	Random	0.002	0.094	189.072
	Ascending	0.002	0.036	66.055
	Descending	0.002	0.054	176.066
	Few swaps	0.002	0.073	44.446
Quick _PISTRO	Random	0.003	0.110	145.507
	Ascending	0.002	0.033	47.344
	Descending	0.004	0.073	94.028
	Few swaps	0.003	0.151	44.967