알고리즘 과제(02)

201204025 김대래

과목명 : 알고리즘

담당 교수 : 김동일

분반 : 02

Index

1. 실습 환경

OS : Linux Ubuntu (Virtualbox) / Windows 10

Language : C

Tool : vim, gcc(Linux) / Visual Studio 2015(in Windows)

- 파일 입출력을 확인하기에 Linux 환경이 더 적합하여 주로 Linux환경에서 작업
- 한글의 깨짐 현상에 따라 코드 이미지 캡쳐는 Visual Studio 2015에서 작업

2. 과제 개요

임의의 나열된 수가 저장된 파일(test_test case.txt)을 입력 받아 각각의 정렬 방법(Merge, Merge + Insertion, Quick, Quick + randomize)에 따라 오름차순으로 정렬하여 결과를 파일로 출력한다.

입력 예시 : test_100.txt / test_1000.txt

출력 예시 : merge.txt / merge_insertion.txt / quick.txt / quick_rand.txt

3. 주요 개념

3.1. 정렬 알고리즘

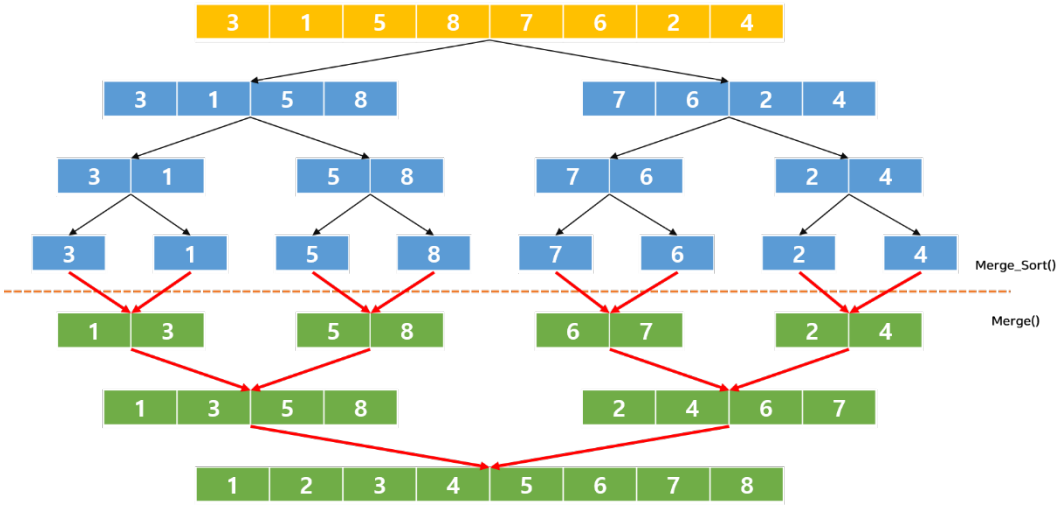
정렬 알고리즘이란 데이터를 번호순 사전 순서와 같이 일정한 순서대로(오름차순, 내림차순 등) 열거하는 방식의 알고리즘이다.

대부분 $O(n^2)$과 $O(n\log n)$의 복잡도를 가진다.

평균 $O(n\log n)$의 복잡도를 가진 고급 정렬 알고리즘은 병합, 퀵, 힙정렬이 있다.

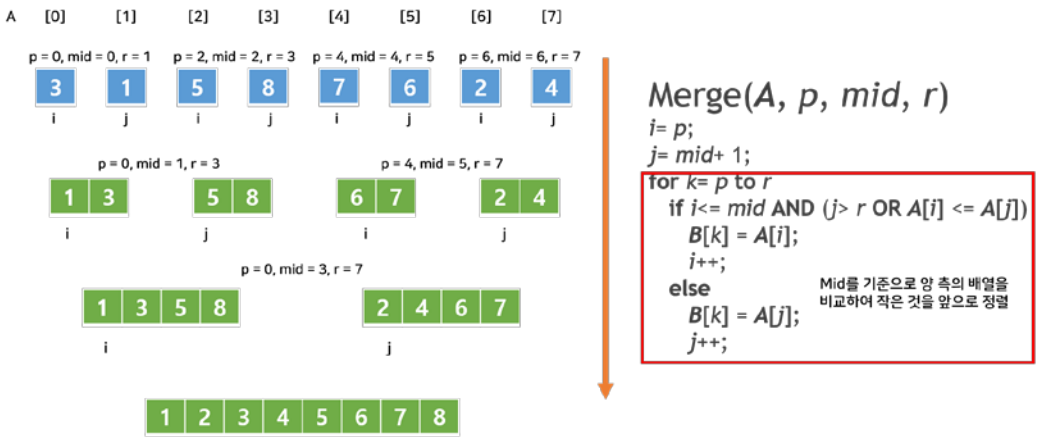
본 과제에서는 고급 정렬 알고리즘 중 병합 정렬과 퀵 정렬을 다룬다.

3.1.1. 병합 정렬



병합 정렬은 분할-정복 알고리즘 중 하나로 정렬되지 않은 리스트를 절반으로 잘라 비슷한 크기의 두 부분 리스트로 나누고 각 부분 리스트를 재귀적으로 리스트의 길이가 0 또는 1 이 될 때까지 나눈다.

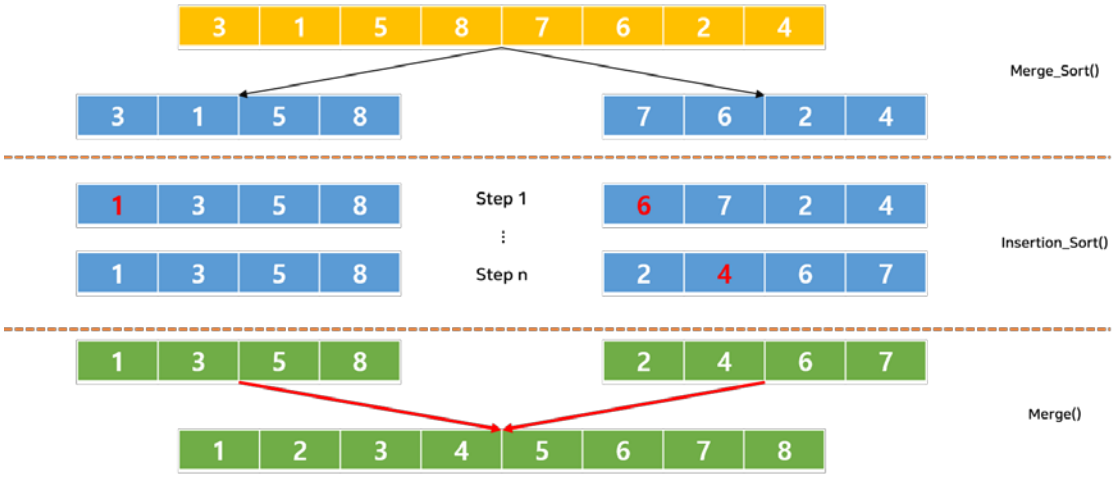
두 부분 리스트를 다시 하나의 정렬된 리스트로 병합하면서 각각의 원소를 정렬시키는 방식이다.



과제에서 주어진 의사 코드를 기준으로 설명하자면

리스트의 왼쪽의 끝 인덱스(p)와 오른쪽의 끝 인덱스(r)까지 mid 를 기준으로 두 부분 배열의 index 를 i 와 j 를 순서대로 비교하며 작은 수를 병합 정렬의 오름차순으로 정렬하여 넣는 방식이다.

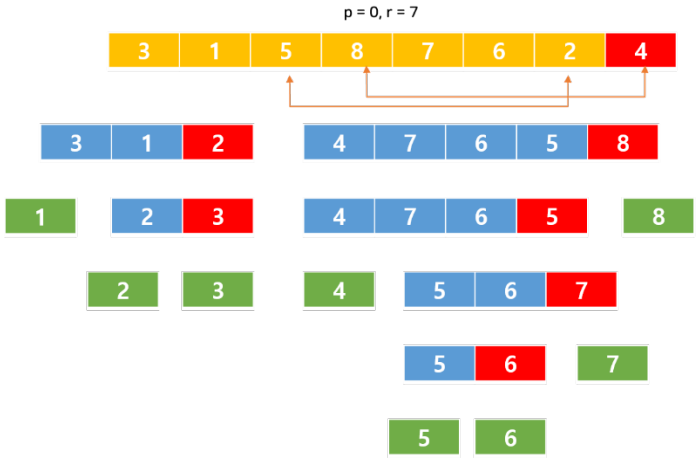
3.1.2. 병합 + 삽입 정렬



앞서 설명한 병합 정렬의 응용으로 부분 배열의 크기가 일정 수준 아래일 때 분할하지 않고 삽입 정렬을 통해 몇 단계의 분할-정복을 생략하는 방법의 정렬 알고리즘이다.

삽입 정렬의 경우 평균 복잡도는 $O(n^2)$이지만 Best-case 의 경우에는 $O(n)$으로 특수한 상황에 따라 $O(n \log n)$보다 더 빠른 성능을 보일 수 있는 장단점이 있다.

3.1.3. 퀵 정렬




```
partition(A, p, r)
  x= A[r]
  i= p-1
  for j= p to r-1
    if A[j] <= x
      i++
      Swap A[i] with A[j]
  Swap A[i+ 1] with A[r]

  return i+ 1
```

pivot의 위치는 i+1이며
작은 수는 앞에, 큰 수는 뒤에 위치

퀵 정렬은 다른 원소와의 비교만으로 정렬을 수행하는 비교 정렬으로 최악의 경우에는 $O(n^2)$번의 비교를 수행하지만 평균적으로 $O(n\log n)$번의 비교를 수행한다.

병합 정렬과 유사하게 분할-정복 방법으로 리스트를 정렬하는데 리스트 중 하나의 원소를 pivot 으로 선정하고 pivot 보다 작은 수는 왼쪽, pivot 보다 큰 수는 오른쪽으로 부분 배열로 나누는 과정을 재귀를 통해 반복한다.

4. 과제 2-1 Merge Sort

4.1. Merge Sort 의사 코드

```
Merge_Sort(A, p, r)
if p < r
  mid = (p + r) / 2
  Merge_Sort(A, p, mid)
  Mege_Sort(A, mid + 1, r)
  merge(A, p, mid, r)
```

```
Merge(A, p, mid, r)
i = p;
j = mid + 1;
for k = p to r
  if i <= mid AND
     (j > r OR A[i] <= A[j])
    B[k] = A[i];
    i++;
  else
    B[k] = A[j];
    j++;
```

4.2. MergeSort(int arr[], int p, int r)

```
void MergeSort(int arr[], int p, int r) {
    merge_count++;
    if (p < r) {
        int mid = (p + r) / 2;

        MergeSort(arr, p, mid);
        MergeSort(arr, mid+1, r);
        Merge(arr, p, mid, r);
    }
}
```

배열의 양 끝을 의미하는 p 와 r 을 인자로 받아

p 가 r 보다 작을 때. 즉, 부분 배열의 크기가 1 보다 크면 재귀(Top-down)를 이용하여 부분 배열로 나눈다.

4.3. Merge(int arr[], int p, int mid, int r)

```
void Merge(int arr[], int p, int mid, int r) {
    int k;
    int i = p;
    int j = mid + 1;
    int arrSize = getSizeArr(arr);
    int *sortArr = (int *)malloc(sizeof(int) * arrSize);

    for (k = p; k <= r; k++) {
        if (i <= mid && (j > r || arr[i] <= arr[j])) {
            sortArr[k] = arr[i];
            i++;
        }
        else {
            sortArr[k] = arr[j];
            j++;
        }
    }
    for (i = p; i <= r; i++) {
        arr[i] = sortArr[i];
    }
}
```

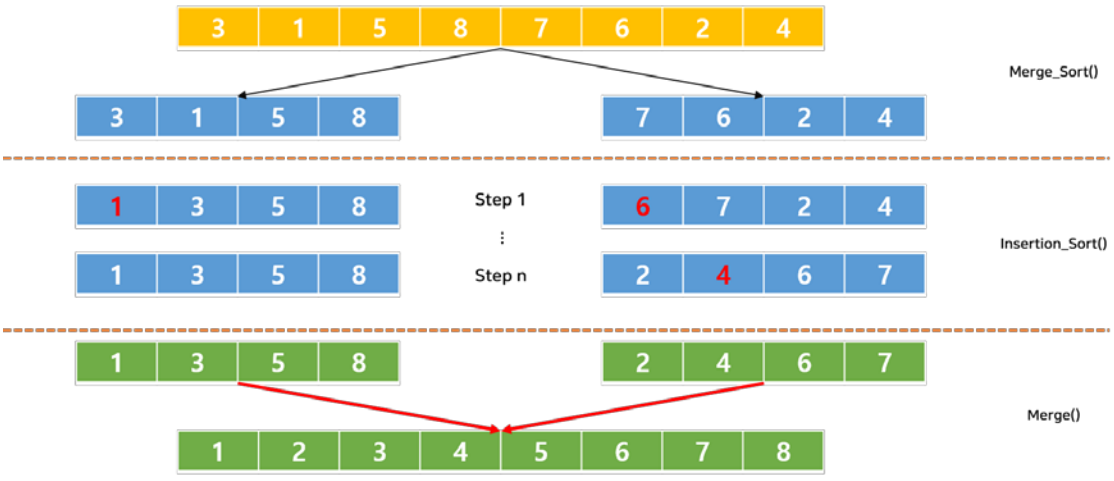
정렬된 배열을 다시 arr 로 넣기 위해 임의의 sortArr 를 선언

왼쪽 배열의 순서(i)와 오른쪽 배열의 순서(j)대로 서로 비교하며 작은 수를 sortArr 에 넣고 다음 원소로 넘어가며 반복

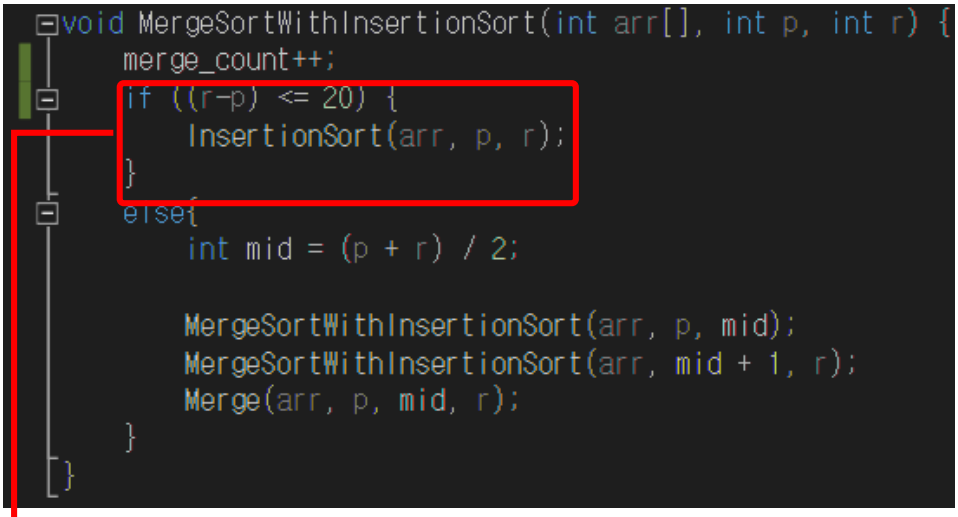
sortArr 를 다시 arr 배열에 복사

5. 과제 2-2 Merge Sort Combine Insertion Sort

5.1. 개념도



5.2. MergeSortWithInsertionSort(int arr[], int p, int r)

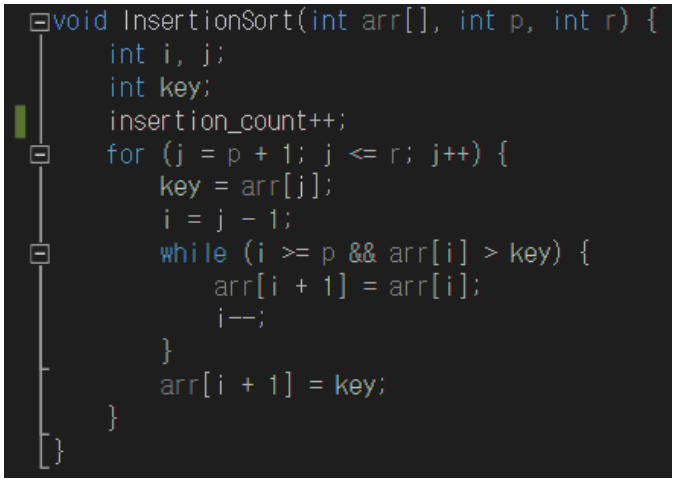


```
void MergeSortWithInsertionSort(int arr[], int p, int r) {
    merge_count++;
    if ((r-p) <= 20) {
        InsertionSort(arr, p, r);
    }
    else{
        int mid = (p + r) / 2;

        MergeSortWithInsertionSort(arr, p, mid);
        MergeSortWithInsertionSort(arr, mid + 1, r);
        Merge(arr, p, mid, r);
    }
}
```

전반적으로 Merge Sort 와 구조는 같음

조건문에서 특정 범위 안의 부분 배열은 Insertion Sort 를 진행하도록 구현

5.3. InsertionSort(int arr[], int p, int r)



```
void InsertionSort(int arr[], int p, int r) {
    int i, j;
    int key;
    insertion_count++;
    for (j = p + 1; j <= r; j++) {
        key = arr[j];
        i = j - 1;
        while (i >= p && arr[i] > key) {
            arr[i + 1] = arr[i];
            i--;
        }
        arr[i + 1] = key;
    }
}
```

지난 과제 hw01 에서 주어진 의사 코드를 참고하여 p~r 까지의 arr 배열을 삽입 정렬하도록 구현

i 번째 수의 자리에 따라 shift 하며 자기 자리에 삽입해주는 방식

6. 과제 2-3 Quick Sort

6.1. Quick Sort 의사 코드

Quick_Sort(*A*, *p*, *r*)

if *p* < *r*

q = partition(*A*, *p*, *r*)

Quick_Sort(*A*, *p*, *q* - 1)

Quick_Sort(*A*, *q* + 1, *r*)

partition(*A*, *p*, *r*)

x = *A*[*r*] / *A*[Random(*p*, *r*)]

i = *p* -1

for *j* = *p* **to** *r* -1

if *A*[*j*] <= *x*

i++

Swap *A*[*i*] **with** *A*[*j*]

Swap *A*[*i* + 1] **with** *A*[*r*]

return *i* + 1

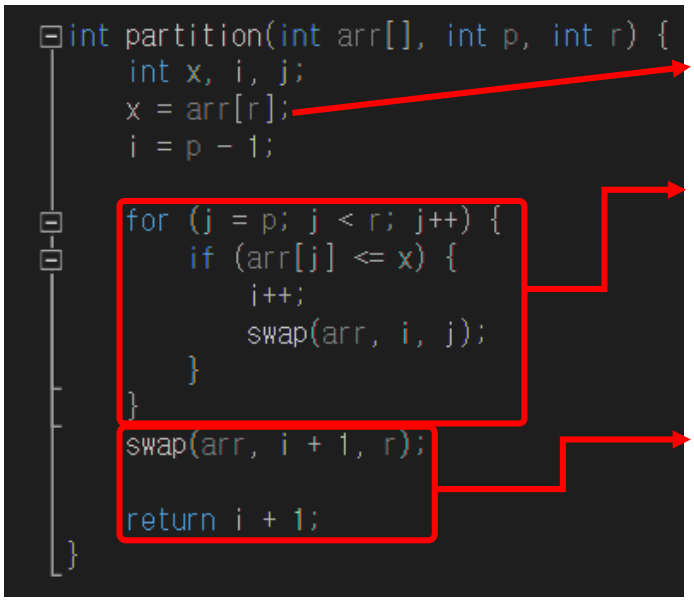
6.2. QuickSort(int arr[], int p, int r)

```
void QuickSort(int arr[], int p, int r) {
    quick_count++;
    int q;
    if (p<r) {
        q = partition(arr, p, r);
        QuickSort(arr, p, q - 1);
        QuickSort(arr, q + 1, r);
    }
}
```

의사 코드에 따라

Partition() 함수로 pivot 에 따른 index 값을 반환하여 해당 index(q)를 기준으로 왼쪽, 오른쪽을 재귀(Top-down)으로 반복하도록 구현

6.3. partition(int arr[], int p, int r)

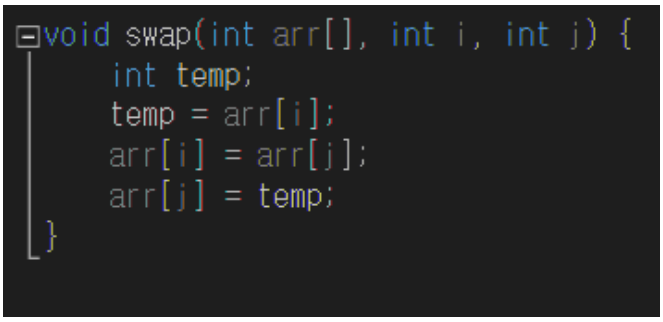


마지막 수(arr[r])를 Pivot(x)로 하고

p~r 까지 순서대로 진행하며 pivot 보다 작으면 왼쪽에 올 수 있도록 swap

자신보다 작은 수를 i 번째 index 까지 나열했으므로 i+1 을 swap 하며 자리를 정하며 해당 자리를 반환

6.4. Swap(int arr[], int l,int j)



임의로 swap 할 데이터를 저장할 temp 를 선언하고 arr[i] 번째와 arr[j]번째 데이터를 교환

7. 과제 2-4 Quick Sort (Randomize Partition)

7.1. Quick Sort(randomize) 의사 코드

```
Quick_Sort(A, p, r)
if p < r
    q = partition(A, p, r)
    Quick_Sort(A, p, q - 1)
    Quick_Sort(A, q + 1, r)
```

```
partition(A, p, r)
x = A[r] A[Random(p, r)]
i = p -1
for j = p to r -1
    if A[j] <= x
        i++
        Swap A[i] with A[j]
Swap A[i + 1] with A[r]
return i + 1
```

7.2. QuickRandSort(int arr[], int p, int r)

```
void QuickRandSort(int arr[], int p, int r) {
    quick_count++;
    int q;
    if (p < r) {
        q = Rand_Partition(arr, p, r);
        QuickRandSort(arr, p, q - 1);
        QuickRandSort(arr, q + 1, r);
    }
}
```

(Quick Sort 와 큰 차이는 없다.)

의사 코드에 따라

Partition() 함수로 pivot 에 따른 index 값을 반환하여 해당 index(q)를 기준으로 왼쪽, 오른쪽을 재귀(Top-down)으로 반복하도록 구현

7.3. Rand_Partition(int arr[], int p, int r)

```
int Rand_Partition(int arr[], int p, int r) {
    int x, i, j;

    /* p~r 사이에 있는 수를 index를 pivot으로 설정*/
    int rand_index = Random(p, r);
    /* 해당 하는 수를 r과 swap함으로써 p~r까지 Quick Sort 과정이 동일*/
    swap(arr, rand_index, r);

    x = arr[r];
    i = p - 1;
    for (j = p; j < r; j++) {
        if (arr[j] <= x) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, r);

    return i + 1;
}
```

Quick Sort 와의 차이점이라면 Pivot 의 선정에 있어 차이만 있다.

Random(p,r)함수를 이용하여 p~r 사이의 임의의 난수 index 값을 마지막 r 값과 swap 해준 뒤 기존 quick 정렬과 동일한 행동을 하도록 구현

7.4. Random(int p, int r)

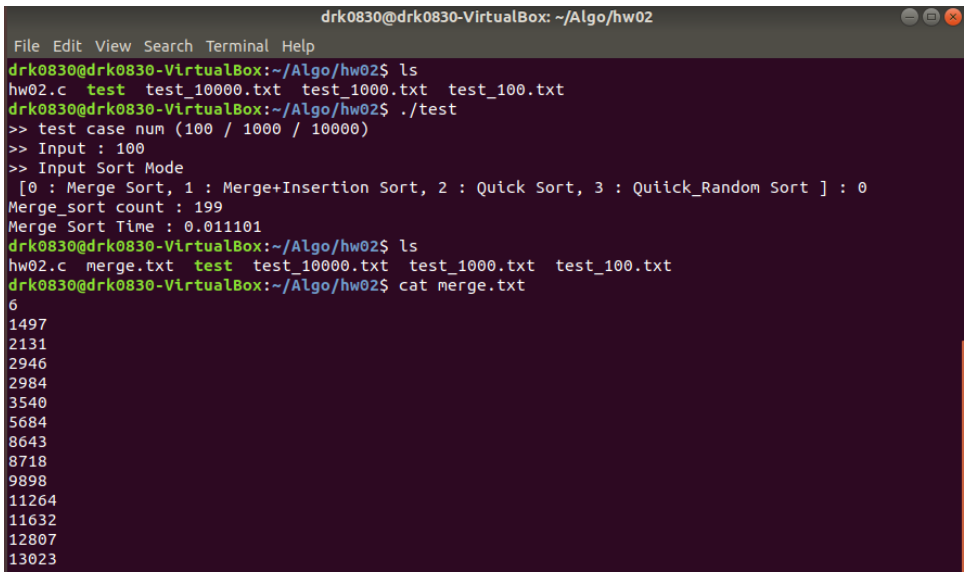
```
int Random(int p, int r) {
    srand(time(NULL));
    int random = (rand() % (r - p)) + p;

    return random;
}
```

난수 생성을 위한 srand()함수와 rand()함수를 이용하여 난수를 생성하고 p~r 범위로 생성을 위하여 (r-p)만큼 나머지 연산 후 p 를 더함

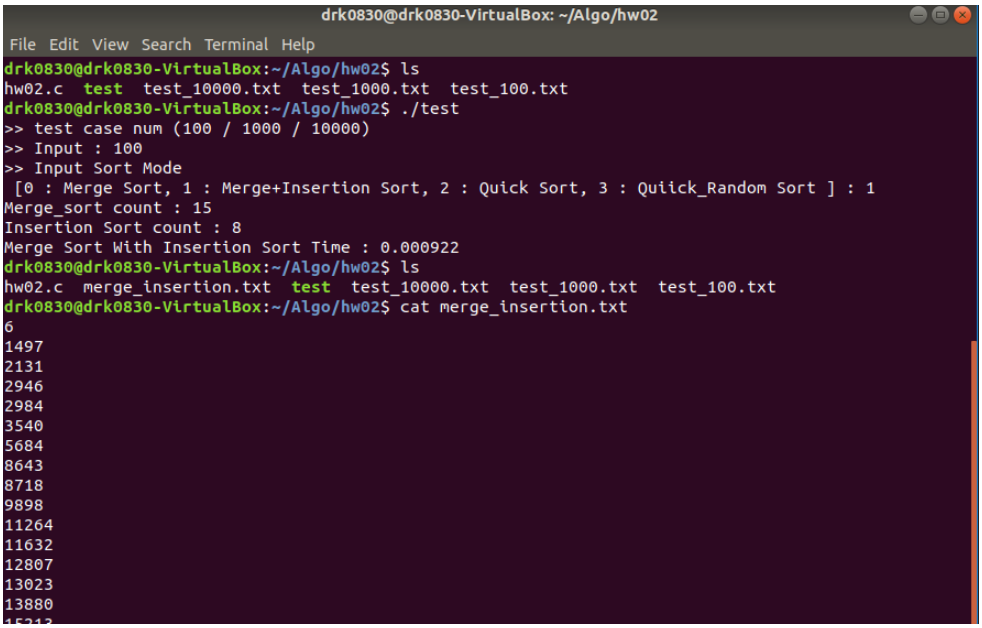
8. 실행 결과

8.1. 정렬 결과 확인

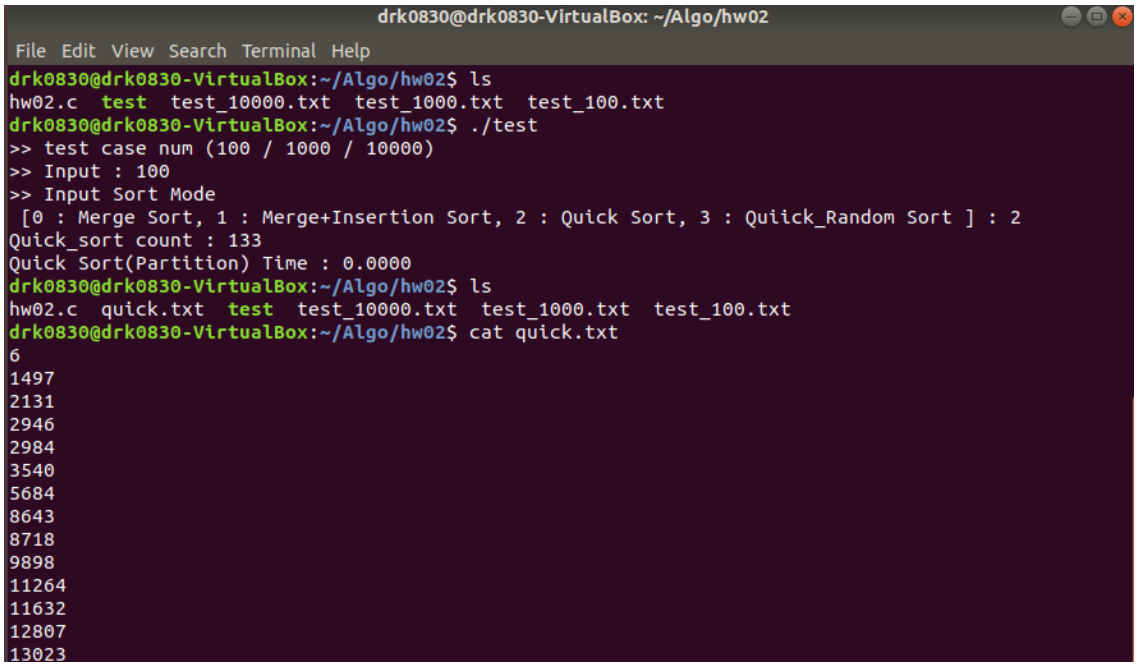
8.1.1. Merge Sort



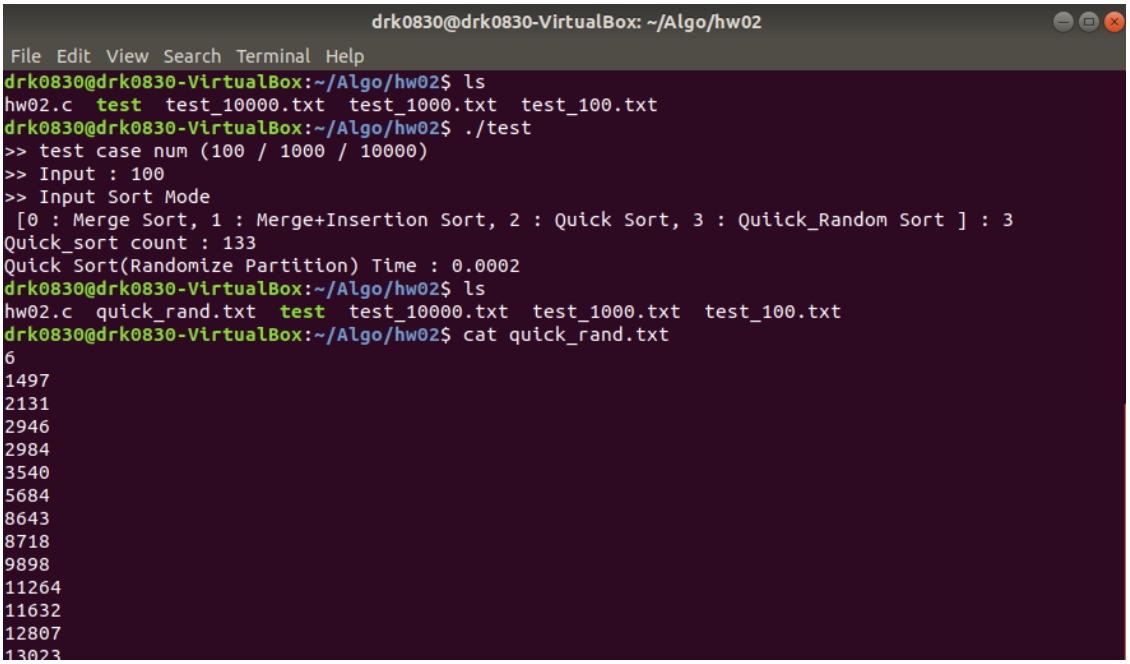
8.1.2. Merge + Insertion Sort



8.1.3. Quick Sort

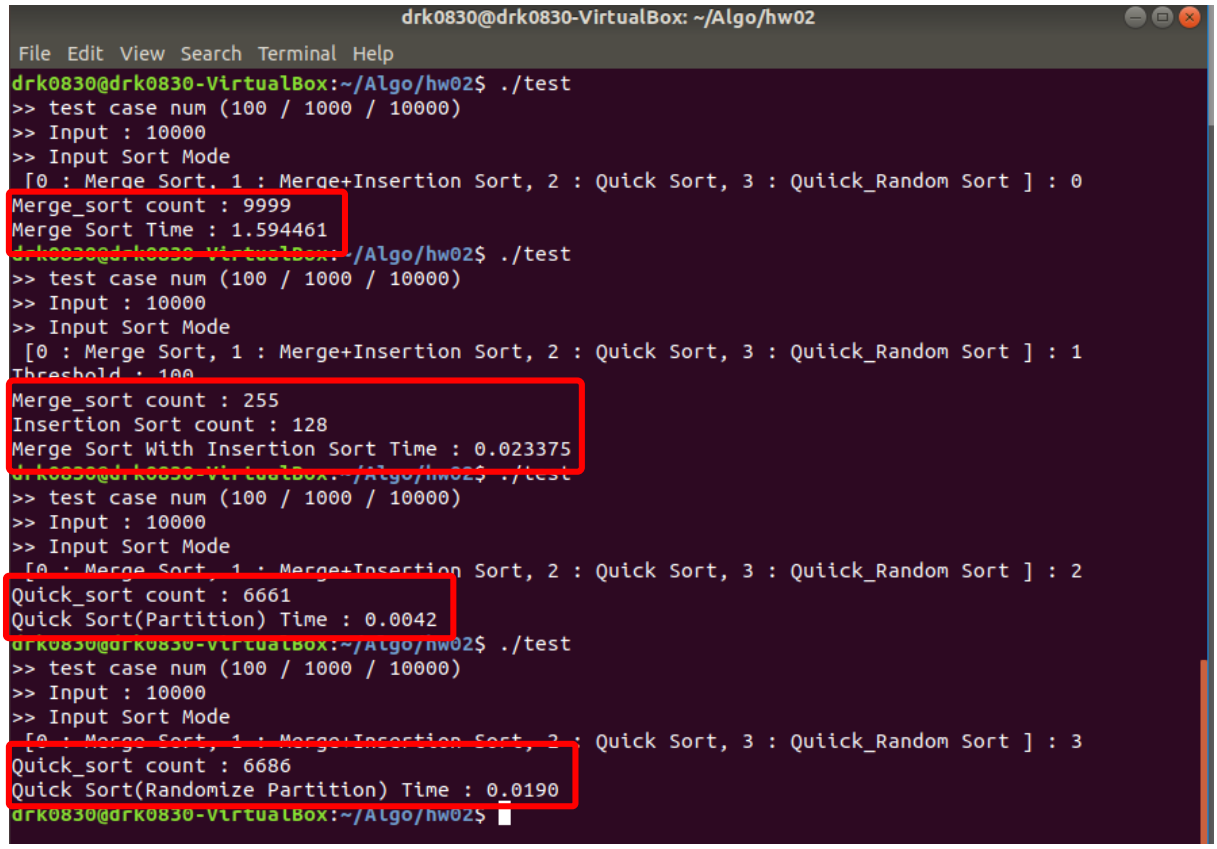


8.1.4. Randomized Quick Sort

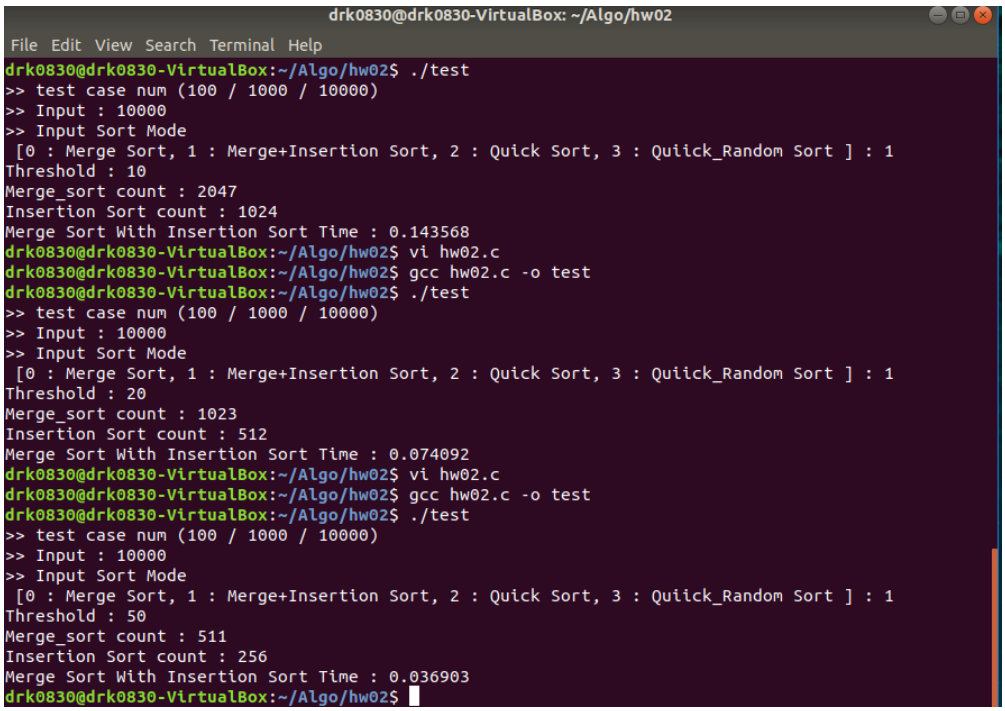


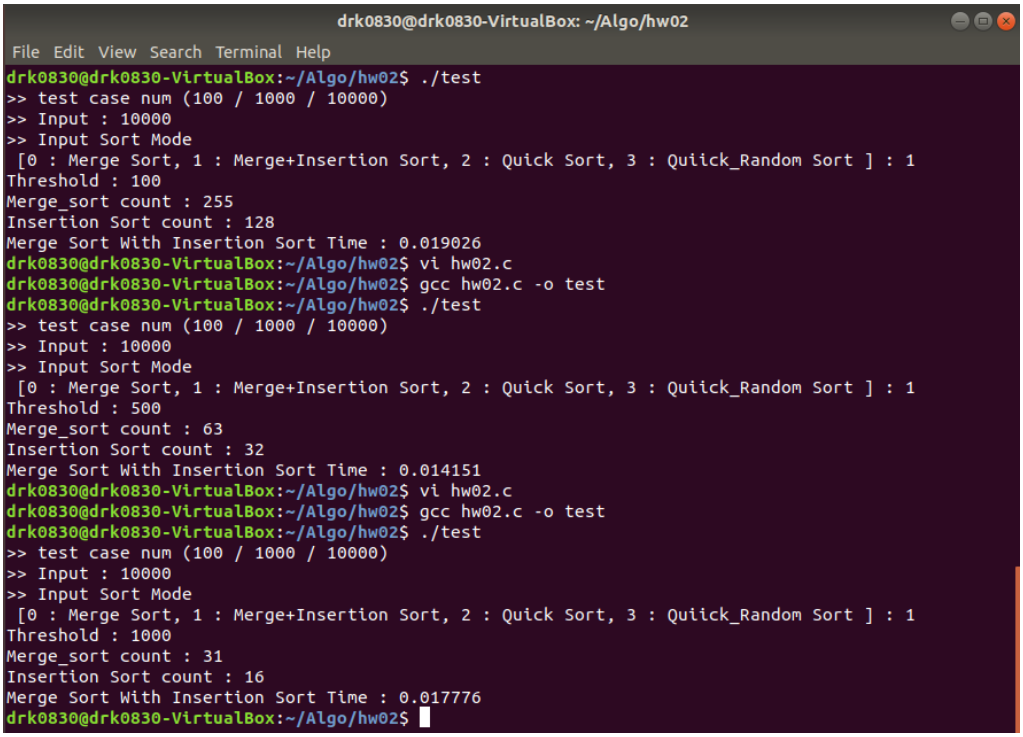
8.2. 알고리즘 속도 비교

- 적은 양의 데이터로는 속도 비교가 어려워 10000 개의 데이터를 통해 테스트



8.3. Merge + Insertion Sort Threshold 에 따른 비교





- Threshold 의 크기가 커짐에 따라 함수 호출의 빈도가 줄어들어 속도가 개선되지만 threshold 가 너무 크면 삽입 정렬의 평균 시간 복잡도가 $O(n^2)$이기 때문에 속도가 오히려 증가하는 경우가 발생

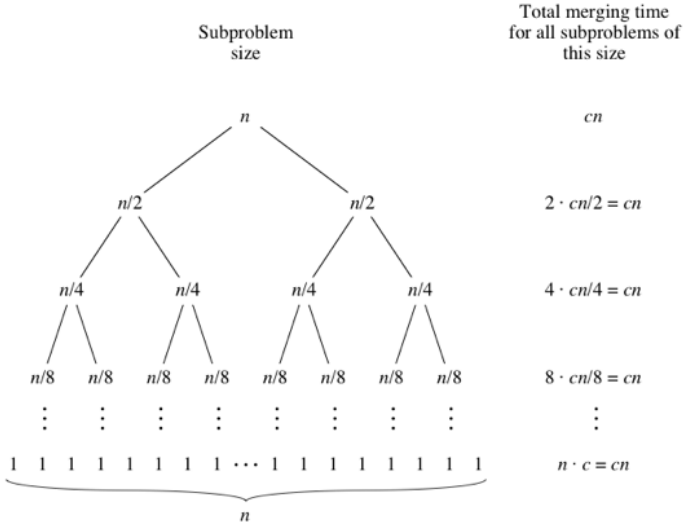
9. 결과 비교

- 시간 복잡도 비교

정렬 알고리즘	최선	평균	최악
병합	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
병합 + 삽입 (X = THRESHOLD)	$O(nx + n\log\frac{n}{x})$	$O(nx + n\log\frac{n}{x})$	$O\left(n + n\log\frac{n}{x}\right)$
퀵	$O(n\log n)$	$O(n\log n)$	$O(n^2)$
랜덤 퀵	$O(n\log n)$	$O(n\log n)$	$O(n^2)$

- 병합 정렬 알고리즘

$$T(n) = \begin{cases} cn & , \ \mathrm{n} = 1 \\ T(\lceil\frac{n}{2}\rceil) + T\left(\lfloor\frac{n}{2}\rfloor\right) + n, & \text{otherwise} \end{cases}$$

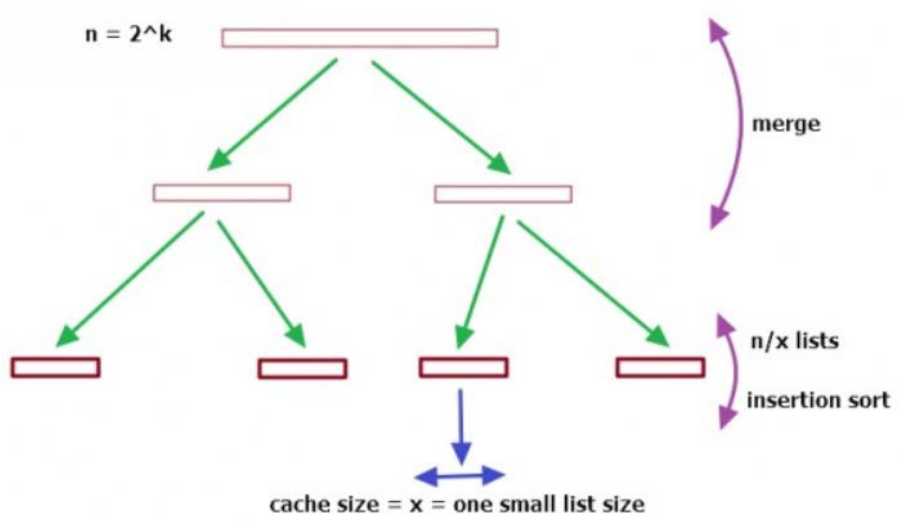


트리의 높이가 h 이면 총 병합 시간은 $\mathrm{h} * \mathrm{cn}$이고 h는 n이 1 이 될 때까지 분할 하는 과정에서 $\log n + 1$이 되기 때문에 $\mathrm{cn}(\log n + 1)$의 시간이 걸린다.

복잡도는 $O(n\log n)$가 된다.

N = 10000 의 경우를 test 했을 때 Merge_sort 호출 수는 9999 회 호출한 것을 확인할 수 있다.

- 병합 + 삽입 정렬 알고리즘



Threshold (= x)보다 작은 분할로 나뉘어 지면 해당 분할 배열은 더 이상 분할하지 않고 삽입 정렬을 하는 방식이다.

병합 정렬에서 x 부분 배열까지 복잡도 $O(\log\frac{n}{x})$를 가지기 때문에 해당 부분에 대한 복잡도는 $O(n\log\frac{n}{x})$를 가진다.

X 에서의 삽입 정렬에 대한 복잡도가 추가되어야 하므로 $\frac{n}{x}$ 에 대한 Insertion(x) 복잡도를 추가하여 $O(\frac{n}{x}Insertion(x)+n\log\frac{n}{x})$가 된다.

$Insertion(x)$의 Best-case 에 따라 최선의 경우 $O\left(n+n\log\frac{n}{x}\right)$가 되며 최악과 평균의 경우 $O(nx+n\log\frac{n}{x})$가 된다.

N = 10000 개의 test-case 로 실험하면서 $Threshold$를 10, 20, 50, 100, 500, 1000 으로 변경해보며 실행해 보았을 때 500 까지는 계속하여 줄어들었지만 1000 까지 키웠을 경우 다시 측정 속도가 늘어난 것을 확인할 수 있었다.

그러나, test-case 의 특수성에 따라 해당 결과가 나온 것일 수도 있기 때문에 한계점(Threshold)를 너무 높이는 것은 좋은 결과를 낼 수 없을 수도 있다.

- 퀵 정렬 알고리즘

$T(n) = 2 * T(n/2) + n = n^2T(n/n^2) + 2 * n = ... = h * n$

수식은 병합 정렬과 유사한 방식을 따르기 때문에 복잡도는 $O(n\log n)$이다.

하지만 pivot 의 선정을 마지막 수를 계속해서 선정하게 된다면 최악의 경우 $O(n^2)$가 된다.

N=10000 개의 테스트 케이스로 실행해 보면 함수 호출 횟수가 6661 회로 Merge Sort 보다 호출 횟수가 적은데 그 이유는 Pivot 에 따라 분할되는 기준이 다르기 때문이다.

또한, 퀵 정렬의 경우 대부분의 컴퓨터 구조에서 효율적으로 작동하도록 설계되어 있으며 제곱 시간이 걸릴 확률이 거의 없도록 설계가 가능하기 때문에 일반적인 다른 $O(n\log n)$의 복잡도를 가지는 알고리즘에 비해 훨씬 빠르게 동작하기 때문에 병합 정렬보다 빠른 속력이 나온 것을 확인할 수 있다.

- 랜덤 퀵 정렬 알고리즘

  퀵 정렬과 같은 복잡도를 가지지만 Randomize 를 하는 이유는 pivot 을 범위 내의 임의의 수를 선정하도록 하여 최악의 경우를 대비하기 위함이다.

  비록 random 함수 호출에 따른 수행 시간은 늘었지만 Quick sort 호출 횟수가 늘어나기도, 줄어들기도 한다.