알고리즘 실습(4주차)

201000287 일어일문학과 유다훈

1. 과제 설명 및 해결 방법

1. Quick Sort

Quick sort를 이용하여 무작위 데이터가 제대로 정렬되는지 확인한다.

2. Randomized Quick Sort

Quick sort는 평균적으로 O(n log n)의 속도를 내지만, 이미 정렬이 되어있는 자료가 입력되는 경우, 최악의 속도로 O(n^2)의 속도가 나게 된다. 이 것을 방지하기 위해 피봇값을 랜덤하게 뽑아서 자리교체를 한 후 정렬을 하게되면 O(n log n)의 속도와 비슷하게 나오게 된다.

3. Insertion sort & Quick sort 구현 및 문제 풀이

데이터가 거의 정렬 되었을때 O(n)의 빠른 속도를 보여주는 Insertion sort를 이용하여, Quick sort를 사용하여 데이터가 어느정도 정렬이 되었다면, 그 때 Quick sort보다 insertion sort를 이용하여 정렬하는 것이 속도를 더욱 빠르게 할 수 있다. 이 때 어떠한 원소의 개수 K인 경우 Quick sort 보다 insertion sort가 더 빠른지 측정한다.

2. 주요 부분 코드 설명(알고리즘 부분)

1. Quick Sort

- partition, QuickSort

```
void swap(int array[], int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

int partition(int array[], int pivot, int right) {
    int x = array[right];
    int i = pivot-1;

    for (int j = pivot; j <= right-1 ; j++) {
        if (array[j] < x ) {
            i++;
            swap(array, i, j);
        }
    }
    swap(array, i+1, right);
    return i+1;
}

void quicksort(int array[], int pivot, int right){
    if (pivot < right) {
        int q = partition(array, pivot, right);
        quicksort(array, pivot, q-1);
        quicksort(array, q+1, right);

    }
}
```

  - 파티션의 기준을 잡을 피봇값이 정렬 대상 데이터의 제일 끝값보다 작은 동안 데이터 크기를 1/2로 줄여나가며 재귀적으로 호출하는 quick sort.
  - 정렬 대상의 제일 끝값과 피봇값을 비교하여, 오름차순으로 정렬해나가며, 피봇값보다 작은 값들은 앞으로, 큰 값들은 뒤로 보내는 partition.

2. Randomized Quick Sort

- randomizedPartition

```
int randomizedPartition(int array[], int pivot, int right) {


    int i = (int)(( rand() % (pivot-right)+pivot ));
    swap(array, right, i);
    //printf("%d", array[pivot]);
    return  partition(array, i, right);
}
```

- 구현방식은 보통의 퀵정렬과 동일하나, 파티션을 구해줄 때 랜덤하게 구해주어야 한다.
- 기준이 되는 값에서 정렬 대상 자료의 맨 끝값을 빼주고, 다시 기준값을 더해준다. 발생시킨 난수에 해당 값으로 나누어 나온 나머지값을 파티션의 기준값으로 한다.
- 위와 같은 방식으로 한다면 재귀호출을 이용하여 범위가 n/2로 나뉘어져도 기준점을 정렬데이터 범위만큼만 구할 수 있다.

3. Quick Sort & Insertion Sort

- Insertion Sort

```
void insertionSort(int array[], int pivot, int right) {
    int i, j, key;

    for(j= pivot+1; j<right; j++) {
        key = array[j];
        for(i = j-1; i >= pivot && array[i] > key; i--) {
            array[i+1] = array[i];
        }
        array[i+1] = key;
    }
}
```

- 삽입 정렬용 함수

- Quicksort and Insertionsort

```
void quickSortAndInsertionSort(int array[], int pivot, int right, int k) {
    testingQuickSort(array, pivot, right, k);
    insertionSort(array, pivot, right);

}
```

- 어떠한 값 k만큼 실행하고 그 후에는 삽입 정렬을 실행한다.

- testtingQuicksort

```
void testingQuickSort(int array[], int pivot, int right, int k) {
    if (right - pivot > k) {
        int q = partition(array, pivot, right);
        testingQuickSort(array, pivot, q-1, k);
        testingQuickSort(array, q+1, right, k);

    }
}
```

■ 정렬할 대상의 범위가 K보다 큰 동안에는 재귀적으로 퀵정렬을 호출한다.

3. 결과

- Quick Sort, Randomized Quick Sort
  - 매우 빠른 속도로 정렬이 잘 된다.
  - 이미 정렬되어 있는 데이터를 입력 값으로 주어도 그 속도는 정렬되지 않은 데이터를 정렬 할때와 비슷한 속도이다.
- Quick Sort & Insertion Sort
  - 퀵정렬의 재귀호출은 파티션으로 인하여 생기는 두 개씩의 트리의 깊이(높이) i가 log(n/k)일때 멈춘다. 이 때 정렬의 결과로 생긴 배열은 k개의 사이즈를 가진 n/k의 부분배열로 나뉘어진다. 이 때 삽입정렬을 한다면 여기서 걸리는 시간은 n/k * O(k^2) 이므로 O(nk)가 된다. 보통 이 값은 O(n log n)보다 빠르긴 하나, k값이 많이 커지면 느려지게 된다.

```
start2 = clock();
quickSortAndInsertionSort(array, 0, 49999, 49955);
end2 = clock();
sortingTime = (double)(end2 - start2) / CLOCKS_PER_SEC; //시간측정
printf("QuickSort and InsertionSort");
printf("%f\n", sortingTime);

return 0;
```

```
Hello, World!
QuickSort0.007619
QuickSort and InsertionSort0.007384
Program ended with exit code: 0
```

  - 5만개의 데이터로 실험결과 약 49955의 k값을 주었을때부터 일반 퀵정렬보다 빨라지며 그 이상의 값을 주면 더욱 빨라지는 것을 확인할 수 있었다.
  - 이것은 입력데이터값이 50개 이하의 작은 데이터값을때 매우 빠른 속도를 보이는 삽입정렬의 특성을 확인할 수 있었다.
  - 결론으로, k는 전체 데이터값 n개에서 -4~50을 했을 때의 값일 때부터 빨리지는 것으로 확인할 수 있었다.