알고리즘 실습(5주차)

201000287 일어일문학과 유다훈

1. 과제 설명 및 해결 방법

1. Counting Sort

Counting sort를 구현하고 실행하여 정렬되어있지 않은 배열을 정렬한다.

2. Radix Sort

주어진 데이터를 한자리 씩 나누어 각 자리마다 Counting sort를 실행한다.

입력받은 데이터는 32비트 정수로 표기하여 8비트씩 나누어서 정렬한다.

3. Randomized Select

배열에서 i번째로 큰 값을 찾는다.

Quick sort에서 쓰인 Randomized-Partition을 이용하여 범위를 줄여간다.

2. 주요 부분 코드 설명(알고리즘 부분)

1. Counting sort

```
void counting_sort(int array[], int arrayB[], int arrayC[], int max ) {
    for (int i =0; i < max; i++) {
        arrayC[i] =0;
    }
    for(int j =0; j  < 500; j++) {
        arrayC[ array[j] ] += 1;

    }
    for(int i= 1; i < max; i++) {
        arrayC[i] = arrayC[i] + arrayC[i-1];
    }
    for (int i = 499; i>=0; i--) {
        arrayB[ arrayC[ array[i] ] ] = array[i];
        arrayC[array[i]] -= 1;
    }



}
```

- counting sort를 하기 위해선 입력되는 데이터의 최대값이 필요하기 때문에 외부에서 데이터의 max값을 넣어준다.
- 배열 데이터의 갯수를 셀 배열 arrayC를 생성하여 초기값을 0으로 초기화한다.
- 입력 데이터 값에 해당하는 arrayC의 인덱스에 1씩 추가를 한다. 입력 데이터 값의 갯수를 카운터 해준다.
- 이후 arrayC[i]값보다 작거나 같은 데이터의 갯수를 카운팅한다.
- 결과를 입력할 arrayB에 arrayC [array[i]]에 해당하는 인덱스를 지정해주고, 해당 위치에 array[i]을 넣어준다.
- 갯수를 하나씩 빼준다.

2. Radix sort

- Radix sort

```
void radixsort(int array[],  int arrayLength) {


    for(int n=0; n<=1 ; n++) {
        int z = 0;
        if (n != 0) {
            z= n*8;
        }
        counting_sort(array, arrayLength, z );
    }

    }
```

- Radix sort는 주어진 데이터를 한 자리 씩 나누어서 정렬하는 방법이다.
- 본 과제에서는 32비트 정수를 8비트씩 나누어가며 정렬해야하고, 작은 단위부터 큰 자리 단위로 올라가기 때문에, for문을 이용하여 shift해줄 값을 counting sort에 전달한다.
- Radix sort는 결과물이 stable하게 정렬되므로 이때 counting sort를 이용하여 정렬한다.

- Counting sort

```
void counting_sort(int array[], int arrayLength, int n ) {
    int arrayB[500];
    int max = 0;
    int arrayDigit[500];


    for (int i=0; i <arrayLength; i++) {
        int a = array[i] >> n & 0xff;
            arrayDigit[i] = a;
    }
    for(int i = 0; i < arrayLength; i++) {
        if (arrayDigit[i] > max) {
            max = arrayDigit[i];
        }
    }
    int arrayC[max+2];

    for (int i =0; i < max+2; i++) {
        arrayC[i] =0;
        arrayB[i] =0;
    }
    for(int j =0; j  < 500; j++) {
        arrayC[ arrayDigit[j] ] += 1;

    }
    for(int i= 1; i < max+2; i++) {
        arrayC[i] = arrayC[i] + arrayC[i-1];
    }
    for (int i = 499; i>=0; i--) {
        arrayB[ arrayC[ arrayDigit[i] ] ] = array[i];
        arrayC[arrayDigit[i]] -= 1;
    }
    for (int i =0; i<500; i++) {
        array[i] = arrayB[i];
    }
}
```

- Counting sort는 위에서 서술한 counting sort와 크게 다른 점은 없다.
- 32비트 정수를 옆으로 n번 시프트한 값을 저장하는 배열 arrayDigit을 선언하고 이것을 counting sort를 이용하여 정렬하였다.
- 결과물이 삽입되는 arrayB에는 정렬된 순서대로 원래의 값을 넣기 위해 정렬 대상 데이터가 들어있는 array배열의 데이터를 삽입해준다.
- 정렬된 결과를 반영하기 위해 원래 데이터 배열 array에 다시 삽입해준다.

3. Randomized Select

```
int randomized_select(int array[], int pivot, int right, int index) {
    if (pivot == right)
        return array[pivot];
    int q = randomizedPartition(array, pivot, right);
    int k = q - pivot +1;

    if (index == k )
        return array[q];

    if (index < k)
        return randomized_select(array, pivot, q-1, index);
    else
        return randomized_select(array, q+1, right, index-k);
}
```

- i번째로 작은 값을 Recursive하게 찾는 방법.

- pivot값을 렌덤하게 뽑아 피봇 값 보다 i값의 대소여부에 따라 원래 데이터를 두 개로 쪼개어 한 쪽으로만 찾아간다.

3. 결과

- Counting sort
  - 데이터를 비교하는 작업 없이 Linear time으로 정렬할 수 있다.
  - 정수 값만 할 수 있다.
  - 데이터의 값이 커지면 커질수록 오래 걸린다.
- Radix sort
  - 비트시프트 연산에 능숙하지 않아서 구현하는 것이 어려웠다.
  - counting sort는 정렬대상 데이터 값의 최대값이 필요하므로 최대값을 구한 후, 최대값을 이용하여 임시배열을 선언하였으나, 왠지모르게 최대값만 가지고는 메모리 참조 오류가 발생하여, 최대값에 +2한 값으로 임시배열 arrayC를 선언하였다.
  - 실행 도중 어디선가 메모리 간섭 혹은 메모리 참조 오류가 일어나는 것 같지만 디버깅을 하여도 어디서 문제였는지 확인하기 어려웠다.
  -
- Randomized Select
  - i번째 작은 수를 잘 찾는다.