2020-11782 김건희

1. 정렬 알고리즘 동작 방식

1-1. Bubble Sort

입력받는 배열의 길이 n에 대해서, I가 0부터 n-2까지 변하는 동안 각각의 I에 대해 j는 0부터 n-i-2까지 변하게 된다. 이때 배열의 j번째 원소와 j+1번째 원소를 비교하여 순서대로 되어있지 않으면 두 원소를 맞바꾼다. j가 n-i-2까지 변하는 이유는 각각의 i에 대해 동작이 실행될 경우 가장 큰 원소가 맨 뒤로 이동하기 때문에 그 다음 i에 대한 동작에서는 이전에 비교했던 배열에서 마지막 한 자리는 비교하지 않아도 되기 때문이다. 예를 들어 i=0일 때 0번째와 1번째 원소의 비교부터 n-2번째와 n-1번째 원소의 비교까지 이루어진다. 그리고 그 다음인 i=1일 때 0번째와 1번째 원소의 비교부터 n-3번째와 n-2번째 원소의 비교까지 이루어진다.

1-2. Insertion Sort

I는 1부터 n-1까지 변하면서 배열의 i번째 원소인 key 값이 들어갈 위치를 찾는 과정을 반복한다. 각 i에 대해 i-1번째 원소부터 앞쪽으로 이동하면서 key 값과 원소를 비교 후, 그 자리의 원소가 key보다 큰 경우 한 칸 뒤로 밀어낸다. 이를 더 이상 비교할 원소가 없거나 key보다 작거나 같은 원소가 등장할 때까지 반복한다. 반복이 끝나면 key가 들어갈 자리를 찾은 것이므로 그 자리에 key 값을 대입한다.

1-3. Heap Sort

힙 정렬을 구현하기 위해 우선 주어진 배열을 최대 힙으로 바꾸는 함수 buildMaxHeap을 구현해주었다. 이를 이용해 배열을 힙으로 바꾼 후 반복문을 통해 힙 정렬을 수행한다. 배열의 길이를 n이라 할 때, i는 n-1부터 1까지 변화시킨다. 각각의 i에 대해 힙에서 i번째 원소와 최댓값인 첫 원소를 바꿈으로써 배열에서 최댓값이 가장 뒤에 위치할 수 있도록 한다. 그리고 최댓값을 맨 뒤로 보냈기 때문에 이를 제외한 i-1번째 원소까지의 배열 일부분을 다시 heap으로 만들어주는 maxHeapify를 구현하고 적용했다. 이 과정을 반복하여 힙 정렬을 구현할 수 있었다. 추가로 배열의 두 원소의 자리를 바꿔주는 swap 함수도 구현해 사용하였다.

1-4. Merge Sort

병합 정렬의 경우 매번 병합할 때마다 서브배열을 따로 생성하면 처리 속도가 느려지기 때문에, 이를 개선하기 위해 주배열과 보조배열이 서로 그 역할을 계속 번갈아 가면서 바꾸는 switching merge 형태로 구현하였다. MergeSort 함수에서는 중간을 기준으로 배열 A의 왼쪽 부분과 오른쪽 부분을 각각 재귀적으로 MergeSort 함수로 그 결과를 B에 저장한다. 그리고 switching_merge 함수를 이용해 배열 B에 저장된 두 부분 배열을 병합하여 A에 저장한다. 이렇게 구현함으로써 A와 B가 주배열과 보조배열의 역할을 번갈아 가며 수행하면서 추가적인 배열을 생성하지 않아도 된다.

1-5. Quick Sort

퀵 정렬도 마찬가지로 재귀적으로 실행되는 알고리즘이기 때문에 quickSort라는 재귀 함수를 구현했다. 이 함수는 배열과 그 배열에서 정렬을 수행할 부분의 첫 원소의 인덱스(left)와 마지막 원소의 인덱스(right)를 입력으로 받는다. Left < right일 경우에만 함수를 실행하도록 설

정하였는데, left=right인 경우에는 정렬할 필요가 없기 때문에 이를 제외하기 위한 것이다. 비교했을 때 left<right인 일반적인 경우라면 pivot을 설정함과 동시에 그 pivot을 기준으로 작은 원소들은 왼쪽, 큰 원소들은 오른쪽에 위치시켜야 한다. 이를 위해 partition 함수를 구현하고 실행했다. 여기서 고려한 점은 중복된 원소가 있을 때 quicksort를 더 빨리 수행시키기 위해 partition 함수를 3-way partition으로 구현한 것이다. partition 함수에서는 입력 받은 부분배열의 가장 왼쪽 원소를 pivot으로 설정하고 이보다 작은 원소는 왼쪽, 큰 원소는 오른쪽에 위치시킨다. 이 과정에서 왼쪽에 가장 최근에 위치시킨 인덱스의 오른쪽 인덱스(lt)와 오른쪽에 가장 최근에 위치시킨 왼쪽 인덱스(gt)의 값을 업데이트한다. 부분배열의 모든 원소에 대해 이 과정이 이루어지면 lt부터 gt의 인덱스를 가지는 원소들은 전부 pivot과 같은 값을 지니는 원소들로 이루어진다. 따라서 partition을 수행하면 lt와 gt를 반환해준다. 이 반환된 값을 이용해 quickSort에서 중복된 원소들의 구간을 제외한 양쪽 부분배열에 대해 다시 quickSort를 재귀적으로 실행한다.

1-6. Radix Sort

기수 정렬을 하기 위해 우선 입력 받은 배열에서 양수 부분과 음수 부분을 따로 저장할 배열 positive와 negative를 선언했다. 그리고 선언한 두 배열에 양수와 음수를 각각 저장했다. 이 과정에서 양수는 그대로 저장했으나, 음수의 경우 이를 양수로 저장하기 위해서 1을 먼저 더하고 -1을 곱하는 변환 과정을 거쳤다. 이렇게 처리한 이유는 음수에 -를 그대로 붙여 양수로 바꾸게 되면 int 범위에서 가장 작은 수가 가장 큰 수보다 절댓값이 크므로 오버플로우가 발생할 수 있기 때문이다. 따라서 이를 방지하기 위해 위 같은 변환 과정을 거쳤다. 다음으로 입력배열에 양수가 있었다면 positive에 대해 radixSort함수를 실행시키고, 동일하게 음수가 있었다면 negative에 대해 radixSort함수를 실행시킨다. 양수 혹은 음수가 있었는지 확인하는 이유는 입력 배열이 양수만 들어오거나 음수만 들어오는 경우가 있을 수 있는데, 이를 확인하지 않고 radixSort함수를 양수 배열과 음수 배열에 대해 실행시킬 경우, 빈 배열에 대해 실행시키는 꼴이 되어 에러가 날 수 있기 때문이다. 이때 radixSort함수의 구현 내용을 살펴보면 입력 원소들 중 최대 자리수를 가지는 값을 max에 저장해 이 수의 자리수를 기준으로 countSort를 실행시킨다. 이때 실행되는 countSort는 정해진 자리의 숫자들에 대해 그 숫자를 가지고 있는 원소들의 개수를 카운트하여 count배열에 저장하고, 누적 합을 계산한다. 그리고 각 원소에 대하여 그 원소가 임시 배열인 output에 들어가야 할 인덱스를 누적합 배열에서 찾고, 찾은 누적합 배열의 원소 값을 -1 하여 이후 같은 누적합 배열의 원소를 사용하는 원소가 있을 경우 output에서 이전에 들어온 원소 그 앞에 그 원소를 위치시킬 수 있도록 한다. 이렇게 생성한 임시배열 값을 원래 배열에 다시 복사해주는 것이 countSort의 역할이다. 이 countSort를 각 자리에 대해 반복하면 radixSort 함수의 역할은 마무리된다. 다시 기수정렬로 돌아와서 이제는 양수 배열과 음수배열이 각각 정렬되었기 때문에 이를 다시 합쳐주는 작업이 필요하다. 우선 앞은 음수 배열이 차지해야 하는데 이전에 변환 과정을 고려하려 각 원소는 +1을 해준 후 -1을 곱해주는 작업이 필요하다 이때 음수를 양수로 바꾼 후 이에 대해 radixSort를 했기 때문에 순서가 반대되므로 이를 다시 뒤집어 주는 과정이 필요하다. 음수 배열을 이렇게 배열했다면 그 뒤로는 양수 배열을 그대로 넣어주면 기수 정렬이 마무리된다.

2. 정렬 알고리즘 동작 시간 분석

정렬 알고리즘/ 입력 데이터 수	100	1000	10000	100000	1000000
Bubble	0	9	93	17036	-
Insertion	0	4	25	730	72353
Heap	0	1	5	19	163
Merge	0	1	4	17	129
Quick	0	2	5	24	119
Radix	0	1	4	20	84

표1. 정렬 알고리즘 동작 시간 - 입력 데이터 범위 : -10^9 ~ 10^9, 동작 시간 단위 : ms, 10회 반복 실험한 평균값 반올림

위 표에서 알 수 있듯이 평균적인 경우에서 가장 느린 알고리즘은 bubble sort이다. 가장 빠른 알고리즘은 입력 데이터의 수에 따라 다르게 나타난다. 10000개의 데이터가 입력될 때 까지는 merge sort와 radix sort가 가장 빠르지만 100000개의 데이터에서는 merge sort가, 1000000개의 데이터부터는 radix sort가 가장 좋은 성능을 보이고 있음을 확인할 수 있다.

3. Search 알고리즘 동작 방식

먼저 getSortedRate 함수를 이용해 배열의 인접한 원소끼리 정렬된 비율을 계산해 이것이 sortedThreshold보다 크면 insertion sort가 가장 빠르다고 할 수 있다. 이때 sortedThreshold의 값을 0.999인 극단적인 값으로 주었는데, 이는 여러 번 실험해본 결과 극단적으로 잘 정렬되어 있는 경우가 아닐 경우 insertion sort가 가장 빠른 정렬 방법이 아니었기 때문이다. 그래서 이를 조건문들 중 가장 앞에 배치했다.

다음으로 getMaxDigit함수를 이용해 입력된 배열에서 가장 큰 자리수를 찾아낸다. 이 값이 radixMaxDigits의 값보다 작거나 같으면 그 안에 조건문을 통해 판별하게 된다. 다수의 실험을 진행한 결과 대부분의 경우 quick sort가 가장 빠른 알고리즘으로 나타났으나, 특정한 배열의 길이의 범위에서 radix sort가 빠른 구간이 있어서 이를 분리하기 위해 조건을 추가했다.

우선 getCollisionRate 함수를 이용해 구한 중복 원소의 비율이 duplicateThreshold보다 크면 quicksort를 선택한다. 자리수나 배열의 길이와 상관없이 높은 수준의 중복 원소 비율을 보이면 quick sort가 빠른 경향을 보이기 때문이다. 다음으로 배열의 길이가 5000보다 크면 radix sort를 선택한다. 이 구간에서만 유독 radix sort가 강세를 보였기 때문이다. 아마 이보다 배열의 길이가 작을 때는 최대 자리수(O(kn))에 비해 배열의 길이(O(nlogn))가 작아지기 때문으로 추측된다. 이 외의 경우에 대해서는 quick sort를 선택한다.

이 알고리즘을 결정하기 위해 실험했던 내용의 일부를 정리하면 마지막에 배치한 표와 같다. 각 케이스에 대해 실험을 10번씩 진행했으며 평균을 구한 후 반올림하였다. 각 범위에 대해 난수를 생성하여 실험했다

4. Search 알고리즘 동작 시간 분석

평균적인 경우를 가정해 -10000 ~ 10000 범위에서 10000개의 random한 수들을 정렬하는 경우를 계산했다. 총 10번의 실행을 거쳐 경과 시간의 평균을 구한 결과 Search를 한 후 해당되

는 정렬을 돌렸을 때는 9ms가 걸렸지만, 모든 정렬을 돌리는 데에는 128ms가 걸려 확연한 차이를 확인할 수 있었다.

범위	Radix	Merge	Quick	Fastest
[-100, 100]	1	1	0	Quick Sort
[-500, 500]	1	1	0	Quick Sort
[-1000, 1000]	1	1	0	Quick Sort
[-5000, 5000]	1	1	0	Quick Sort
[-10000, 10000]	1	1	0	Quick Sort

표2. 입력 데이터 개수 1000일 때 정렬 알고리즘 비교 : 각 범위마다 난수 생성 후 실험 10회 반복하고 평균값 계산, 단위는 ms

범위	Radix	Merge	Quick	Fastest
[-100, 100]	1	1	2	Quick Sort
[-500, 500]	1	2	2	Quick Sort
[-1000, 1000]	1	2	2	Quick Sort
[-5000, 5000]	2	2	1	Quick Sort
[-10000, 10000]	2	3	2	Quick Sort

표3. 입력 데이터 개수 5000일 때 정렬 알고리즘 비교 : 각 범위마다 난수 생성 후 실험 10회 반복하고 평균값 계산, 단위는 ms

범위	Radix	Merge	Quick	Fastest
[-100, 100]	2	3	3	Quick Sort
[-500, 500]	2	4	3	Quick Sort
[-1000, 1000]	2	4	3	Quick Sort
[-5000, 5000]	3	3	2	Quick Sort
[-10000, 10000]	3	4	2	Quick Sort

표4. 입력 데이터 개수 10000일 때 정렬 알고리즘 비교 : 각 범위마다 난수 생성 후 실험 10회 반복하고 평균값 계산, 단위는 ms