DS HW4 : Sorting Test

2022-18599
이기석

1. 서론
본 과제에서는 Bubble sort, Insertion sort, Merge sort, Heap sort, Quick sort, Radix sort 의 6 가지 정렬 방법을 이용해서 주어진 배열을 정렬한다. 또한, 이 과정에서 최대한 optimize 를 시키는 것을 목표로 한다. 마지막으로, 위의 6 가지 정렬 방법들 중 가장 효율적인 정렬 방법을 찾는 것을 O(n) 시간 내에 수행하는 Search 알고리즘을 제작한다.

2. 정렬 구현

2.1. Bubble Sort
Bubble Sort 는 chatGPT 를 이용하여 작성한 코드를 변형하지 않고 그대로 사용하였다. 이중 for 문으로 구성되어 있고, 항상 이 for 문을 모두 돌기 때문에 이는 Worst-case, best-case, average-case 모두 $O(n^2)$으로 동작한다.

2.2. Insertion Sort
Insertion Sort 역시 chatGPT 를 이용하여 구현하였고, for 문 하나와 while 문 하나로 구성되어 있다. 이는 Worst-case 와 average-case 에는 $O(n^2)$으로 동작하지만, 입력들이 거의 정렬되어 있는 상태에서는 O(n)과 유사하게 동작한다. 따라서, Search 함수를 구현하는 과정에서 이 성질을 사용할 것이다.

2.3. Merge Sort
Merge Sort 는 chatGPT 를 이용해서 구현하였고, divide 과정인 DoMergeSort()와 conquer 과정인 merge()로 구성되어 있다. Merge 함수에서는 result 배열에 left, right 을 옮겨적는 식으로 구현하였다. 따라서 merge sort 는 거의 uniform 하게 O(n logn)의 시간복잡도를 가지고 수행된다.

2.4. Heap Sort
Heap Sort 는 chatGPT 를 이용하여 구현했고, 먼저 makeHeap()을 통해 Heap 을 input 배열을 heap 으로 만들어준다. 이후에, makeHeap()을 계속 호출하여 구현하였다. 이는 교재의 percoalte 와 유사하다. 따라서, input 들이 모두 같을 때는 O(n)으로, 평소에는 O(n logn)으로 작동한다.

2.5. Quick Sort
Quick Sort 는 pivot 은 항상 오른쪽 끝 원소로 고정시켜 두었고, collision 에 의한 중복을 처리하기 위해서 값이 pivot 과 동일할 때, index 기준으로 index 가 짝수면 왼쪽에, 홀수면 오른쪽에 배치하였다. 이를 이용하여 collision 에 의한 시간복잡도를 평균적으로 O(nlogn) 정도로 줄일 수 있다. 다만, 최악의 경우 $O(n^2)$이 될 수 있다. 따라서, 이 부분을 Search 에서 처리해줘야 한다.

2.6. Radix Sort
Radix Sort 는 먼저 주어진 배열을 양수, 음수 배열로 나눈 뒤에 각각에 대하여 정렬을 수행한 것이다. LSD 기준으로 정렬을 진행하고, 하나씩 늘려서 진행했다. 또한, 음수인 경우에는 절댓값을 기준으로 정렬한 것이기에 역순으로 정렬이 되는데, 마지막에 이 부분을 처리했다. 이는 O(kn)으로 k 가 작을 때는 유의미하게 빨라야 한다.

3. Search 구현
3.1. Default algorithm
Search method 는 주어진 배열을 한번만 순회하여 O(n) 만에 최적의 정렬 알고리즘을 찾아주는 알고리즘이다. 일반적으로 랜덤하게 충분히 분산된 데이터가 주어졌을 때 QuickSort 가 가장 빠르기에, 모든 경우에 해당되지 않는 default case 에 해당하는 정렬알고리즘은 quicksort 로 하였다.

3.2. Insertion sort 를 사용하는 경우

Insertion sort 는 2.2 절에서도 언급하였듯이 주어진 배열이 거의 정렬되었을 때 빠르다. 따라서, decrease_count 라는 변수를 만들었다. 이 변수는 배열의 v[i], v[i+1]에 대해서 처음으로 v[i] > v[i+1]인 i 를 저장하고, 이를 전체 배열의 크기에서 뺀다. 즉, decrease_count 는 배열의 시작부터 연속으로 증가하는 가장 긴 연속부분수열을 제외한 나머지의 크기이다. 이 값이 k 라고 하자. 이때, n-k 번째 항까지는 삽입정렬이 매우 빠르게 진행된다. 이유는, 이전의 모든 항들보다 이후 항들이 크기 때문에, while 문이 한 번 돌고 끝나기 때문이다. 이후 k 개의 항들에 대해서 고려하면, 이들은 최대 n-k 번의 연산을 거친다. 따라서, 총 시간복잡도는 (n-k)k 에 비례한다고 할 수 있다. 본 실험에서는 k < log_2(n)인 경우에는 insertion sort 가 빠르다는 사실을 확인하였다. 따라서 다음의 statement 를 이용해서 insertion sort 를 사용하는 경우를 제한할 수 있다.

```
if(decrease_count < (int)Math.log(value.length)) return 'I';
```

3.3. Radix sort 를 사용하는 경우

Radix sort 는 많은 경우에 다른 정렬들보다 빠르다. 하지만, 본 실습에서 사용한 radix sort 는 다른 정렬들에 비해서 ArrayCopy 를 다수 사용하여 그다지 빠르지 않는 모습을 보였다. 그러나, 두 자리 이하의 정렬에서는 유의미하게 빠르다는 사실을 확인할 수 있었다. 따라서, 아래의 코드를 이용해서 두 자리 이하의 정렬에서는 radix 를 사용하도록 하였다. 아래의 식에서 max_term 은 최댓값, min_term 은 최솟값이다.

```
if(max_term<100 && min_term> -100) return 'R';
```

3.4. Heap sort 를 사용하는 경우

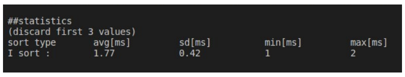
Heap sort 를 사용하는 경우는 Quick sort 를 사용하지 않는 경우와 완벽히 일치한다. Quick sort 가 느린 경우는 세 가지이다. 첫번째, 원래 배열이 거의 정렬되어 있는 경우, 두번째, 원래 배열이 역순으로 거의 정렬되어 있는 경우, 그리고 마지막으로 충돌 횟수, 즉 같은 값이 많은 경우이다. 세 가지 경우 모두 quicksort 가 $O(n^2)$을 보이면서 bubble sort 급으로 좋지 않은 효율을 보여준다. 처음의 경우는 3.2 에서 삽입정렬을 이용해서 처리하였다. 두번째 경우에도, 앞과 동일한 방법을 이용하여 increase_count 라는 변수를 정의하였고, 이 값이 너무 작으면 역순으로 거의 정렬되어 있다고 판단하였다. 따라서, 이 경우에는 heap sort 를 사용하였다.

마지막으로, collision 이 일어나는 경우들을 최대한 분산시키는 방법으로 quicksort 를 구현하였지만, 운이 나쁘면 testcase 에 따라 교묘하게 걸려들도록 할 수 있었다. 이를 처리하기 위해서 hash table 에 각 값이 몇 번 등장하는지를 저장하였고, 등장하는 횟수의 제곱의 합을 구하였다. 이 값의 최솟값은 $1^2+...+1^2=n$ 이고, 최댓값은 n^2 이다. 따라서, 이 값이 10nlog_2(n) 이상이면 heap sort 를 수행하도록 하였다. 이 threshold 는 여러 번의 시행착오 이후에 나온 결과이다.

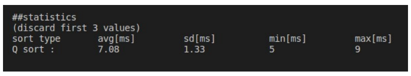
4. Appendix

4.1. Data for 3.2

Insertion sort 가 좋을 때를 비교한 것은 아래와 같다.



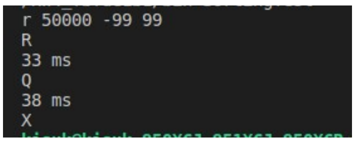
```
##statistics
(discard first 3 values)
sort type       avg[ms]         sd[ms]          min[ms]         max[ms]
I sort :        1.77            0.42            1               2
```



```
##statistics
(discard first 3 values)
sort type       avg[ms]         sd[ms]          min[ms]         max[ms]
Q sort :        7.08            1.33            5               9
```

위는 정확히 val[i] = i for i<=n-log_2(n), val[i] = -i for i>n – log_2(n)으로 잡은 것이다. 즉, 위에서 insertion sort 를 고르는 threshold 값이다. 이때는 insertion sort 가 세 배 이상 빠르다는 것을 확인할 수 있다. data 는 20000 개로 잡았다. 이유는, 더 크게 하면 quick sort 에서 stack overflow 가 일어나기 때문이다.

4.2. Data for 3.3



```
r 50000 -99 99
R
33 ms
Q
38 ms
X
```

위는 50000 개의 데이터를 [-99,99] 범위에서 실행했을 때의 radix sort, quick sort 의 결과이다. 위의 과정을 여러 번 시행했을 때, radix sort 가 quick sort 에 비해 근소적인 차이로 더 바름을 확인하였다. 또한, 수가 많아질 때 이 현상은 극대화된다. 실제로 데이터가 50 만개 이상일 경우, 구간이 좁으면 radix sort 가 5 배 이상 빨라지는 현상을 확인하였다. 따라서 threshold 를 100 보다 data 의 절댓값의 최댓값이 작은 경우로 설정한 것이다. 자릿수가 많아지면 많아질수록, bucket 을 형성하는데 비용이 많이 들기 때문에 매우 큰 폭으로 증가함을 확인하였다.

4.3. Data for 3.4

개선된 quicksort 에 대하여, 구간이 좁지만 데이터의 수가 많아 collision 이 많이 발생하는 랜덤 데이터셋에서 quicksort 를 수행한 결과와 heapsort 를 수행한 결과는 크게 차이가 나지 않는다. 또한, quicksort 를 수행하는 과정에서 비교 연산에 관하여 비용이 조금 더 들어가는 것은 데이터 상 유의미한 지표로 나타나지는 않는 것으로 확인하였다. 하지만, 만약의 상황을 대비하여 collsion 이 너무 많은 경우에는 quicksort 대신 heapsort 를 사용하도록 하였다.

이제, 원래의 배열이 역순으로 정렬되어 있었을 경우에 대한 데이터셋은 아래와 같다.

```
##statistics
(discard first 3 values)
sort type       avg[ms]         sd[ms]          min[ms]         max[ms]
H sort :        2.15            0.53            1               3
```

```
##statistics
(discard first 3 values)
sort type       avg[ms]         sd[ms]          min[ms]         max[ms]
Q sort :        49.92           4.36            45              62
```

위는 거의 역순 정렬되어 있는 데이터에 대한 testset 이다. 10000 개의 데이터를 가지고 있고, 각각은 역순 정렬되어 있다. 이때, heap 이 엄청난 차이를 보이면서 빠르다는 사실을 확인할 수 있다.

4.4. Discussion for number of increasing consecutive pairs and further improvements

과제 spec 에 명시된 대로, insertion sort 를 사용하는 것을 증가하는 연속된 순서쌍의 수를 이용해서 정의할 수도 있다. 하지만, 이에는 치명적인 반례가 존재한다. 바로 아래의 수열이다.

val[i] = i for i<n/2, val[i] = i-INF for i>=n/2

위의 수열은 증가하는 연속된 순서쌍이 n-2 개나 되지만, 삽입 정렬을 수행했을 때 시간이 O(n^2)로 나타난다. 따라서 이러한 경우를 처리하지 못하기 때문에, 처음부터 증가하는 수열을 잡는 것으로 대체하였다.

또한, LIS 가 n-O(logn)인 경우에도 위와 같은 논의로 삽입정렬을 쉽게 처리할 수 있다. 하지만, LIS 를 구하는 가장 잘 알려진 빠른 알고리즘은 이진탐색을 이용하여 log n scale 로 작동하고, 이는 Search 를 O(n)에 구현해야 한다는 과제의 취지에 맞지 않기 때문에 사용하지 않았다.