자료구조

# HW 4 Report

자유전공학부 김지원
2019-11563

1. **정렬 알고리즘의 동작 방식** (예. input array : A[0, ..., n-1] 일 때)
    a. Bubble Sort

    Bubble Sort 는 정렬의 앞(A[0]) 에서부터 A[j]까지 훑는데, 이 j 는 for 문이 한 번씩 돌 때마다 크기가 하나씩 작아진다. 두 개의 원소를 비교하고, 이 두 개의 원소가 순서가 바뀌어 있으면 (정렬되지 않았으면) 두 원소를 swap 하여 정렬해준다. 또한, 추가적으로 'swapped'라는 변수를 추가하였다. 만일 처음 배열을 전체 iteration 할 때 한 번도 swap 이 일어나지 않았으면, 이 배열은 이미 정렬된 배열이므로 즉시 break 를 하여 시간 단축을 하도록 구현하였다.

    b. Insertion Sort

    0 부터 n-1 까지 전체 배열을 훑으면서 현재 원소 A[i]가 올바른 위치에 가게 하도록 A[0]-A[i-1] 사이에 끼워 넣는다. 즉, A[i]가 바로 앞의 원소인 A[j]보다 작다면, 정렬이 되지 않은 상황이므로 j 를 하나씩 줄이면서 원소 A[i]와 비교하고, A[j]는 한 칸씩 뒤로 밀어주어 알맞은 자리에 A[i]를 삽입할 수 있도록 한다.

    c. Heap Sort

    주어진 배열을 heapify 하여 heap 자료구조 형태로 만들어준다. Heap 자료구조 형태에 따라 가장 큰 max value 가 배열의 맨 앞에 위치하게 된다. 배열의 맨 앞과 맨 뒤(A[i])를 바꿔준 뒤, i 의 크기를 점점 줄이며 시행을 반복한다.

    d. Merge Sort

    먼저 배열을 원소가 1 개일 때까지 이등분한 뒤 각각을 정렬하여 merge 하는 작업을 원래 배열의 크기가 될 때까지 반복한다. Merge 함수는 이등분된 두 배열을 각각 left, right 로 두고, left 의 첫번째 원소와 right 의 첫번째 원소부터 비교를 수행하며 iterate 한다. Merge 함수에 의해 이미 정렬된 left, right 배열은 새로운 mergedArray 에 저장한다.

    e. Quick Sort

    Quick Sort 는 배열의 맨 마지막 원소를 pivot 으로 잡은 뒤 이 pivot 을 기준으로 partition 함수를 수행한 뒤, 다시 pivot 의 왼쪽 배열과 오른쪽 배열을 quick sort 하는 형식으로 recursive 하게 진행되고, 이를 수행하는 함수가 quickSortRecursive 이다. 이렇게 정렬된 왼쪽, 오른쪽 함수를 합쳐주면 전체 정렬된 배열을 구할 수 있다.

    f. Radix Sort

    음수가 포함된 인풋이 들어올 수 있으니 수행 전 먼저 음수와 양수를 나눠 구분한 뒤 따로따로 radix sort 를 적용한다. 단 음수의 경우 절댓값에 대해 ascending order 가 아닌 descending order 로 정렬을 하여야 정확하기 때문에 radix sort 를 진행한 뒤 순서를 바꾸는 reverseArray 함수를 수행하여 정렬하도록 했다. 음수와 양수 각각 집합에 대해서

radixSortHelper 함수를 적용한다. 이 함수는 가장 큰 max value 를 구한 뒤(max value 의 자리수만큼 recursion 을 진행하기 위하여), 각 자리수에 대하여 countingSort 를 수행한다. Counting sort 는 배열 숫자들의 least significant digit (1 의 자리수)부터 진행한다. 0-9 까지 횟수와 누적합을 구한 뒤, 이에 따라 해당 digit 에 대해 정렬된 output array 를 구하고, 이를 모든 digit 에 대해 recursive 하게 적용한다.

2. **동작 시간 분석** (정렬에 걸린 시간을 측정하고 이를 분석합니다. 특히, data 의 개수에 따른 분석은 반드시 포함하도록 합니다.)

   1) 데이터 개수에 따른 시간 측정 및 비교 (unit : ms)
   ( Range: 최소 -100,000~ 최대 100,000 고정)

| | 10,000 | 50,000 | 100,000 | 1,000,000 |
|---|---|---|---|---|
| Bubble | 111.09 | 4380.28 | - | - |
| Insertion | 14.12 | 201.02 | 800.32 | - |
| Heap | 3.12 | 13.23 | 24.38 | 185.32 |
| Merge | 3.01 | 14.39 | 27.38 | 193.47 |
| Quick | 1.56 | 8.93 | 15.32 | 108.23 |
| Radix | 3.21 | 7.12 | 10.98 | 71.09 |

실험은 랜덤으로 난수를 생성하여 각 데이터 개수에 따른 시간을 측정하였다.각 정렬 방법 당 24 번씩 반복하여 실험을 진행하고, 앞의 4 번의 실험결과는 컴파일러로 인해 왜곡된 시간일 수 있으므로 제외하고, 뒤 20 번 실험의 평균으로 계산하였다.

**Bubble sort** 의 예상 수행시간은 O(n^2)이므로 예상과 같이 수행시간이 가장 오래 걸리며, 배열 크기가 증가할 때마다 기하급수적으로 시간이 증가함을 알 수 있다. **Insertion sort** 역시 bubble sort 보다는 시간이 적게 걸리지만, 배열의 크기에 따라 기하급수적으로 증가한다. **Heap sort** 의 경우 데이터가 10 배씩 증가함에 따라 약 13 배, 14 배 증가하며 시간 복잡도가 O(nlogn)임을 확인할 수 있다. **Heap, merge, quick** 중 대체적으로 quick sort 가 가장 빠른 것을 볼 수 있다. 그러나 세 정렬 방식은 모두 O(nlogn)으로 자료 개수가 증가할수록 비슷한 시간대에 수렴함을 알 수 있다. **Radix sort** 같은 경우 자료 개수가 적을 때는 기타 추가 작업으로 인해 quicksort 보다 시간이 오래 걸리지만, 자료 개수가 증가할수록 O(n) 시간에 비례하여 가장 빠른 소팅 알고리즘이다. 특히 이는 실험 조건에서 range 를 -100,000, 100,000 으로 고정하였기 때문으로 보인다.

3. **Search 동작 방식** (어떤 아이디어들을 어떻게 구현했는지, 하이퍼 파라미터는 어떻게 설정했는지 설명하도록 합니다.)

아래 모든 실험은 각 경우 당 첫 4 번의 수행을 제외하고, 5 번째 수행부터 20 번 연속 반복하여 평균을 산출한 값들로 비교를 진행하였다. 또한, 모든 숫자의 단위는 ms 이다.

a. 입력의 **최대 자릿수가 9 이하**면 **radix sort** 사용 ( 배열 크기 1,000,000 고정)

| | 9 자리 | 10 자리 | 15 자리 (long) |
|---|---|---|---|
| Quick | 113.23 | 115.63 | 115.49 |
| Radix | 89.49 | 118.21 | 119.27 |

위 실험은 배열의 크기를 1,000,000 으로 고정한 채 range 를 바꾸며 자릿수 별로 시간이 달라짐을 측정하였다 (음수 포함). 2 번의 실험과 동일하게 처음 4 번 수행 이후 20 번 수행 값의 평균으로 계산하였다. 또한, 배열의 개수가 1,000,000 일 때 수행시간이 가장 적은 quick sort 와 비교를 하여 k 를 측정하였다.

9 자릿수까지는 radix sort 가 다른 정렬 방법 중 가장 빠른 quick sort 보다 빨리 작동하는 것을 볼 수 있다. 그러나 10 자릿수가 되면 quick sort 가 더 빨라지며, 자릿수가 더 커질수록 radix sort 의 값은 약간 증가하는 것을 관찰할 수 있다. 따라서 입력의 최대 자릿수가 9 자리 이하인 경우에만 radix sort 는 사용하고, 이외의 경우에는 quick sort 를 사용하는게 더 빠름을 알 수 있다.

단, **자료 개수가 50,000 개 미만인 경우** radix sort 보다 quick sort 가 대체적으로 더 빠른 것을 관찰하였는데 (위 2 번의 실험) 이는 radix sort 를 하기 위해 부가적으로 걸리는 작업 시간 때문이라 판단하였다. 따라서 Search 알고리즘에서 자료 개수가 50,000 개 이상이고, 자릿수가 9 이하인 경우에 'R'을 리턴하도록 구현하였다.

b. Hash table 에 입력을 넣어서 **충돌 발생률이 0.998 이상**이면 **merge sort** 사용 ( 배열 크기 100,000 고정)

| | 0.995 | 0.998 (nextInt(size/500)) | 0.999 (nextInt(size/1000)) |
|---|---|---|---|
| Merge | 25.12 | 22.63 | 20.38 |
| Quick | 14.83 | 25.71 | 37.24 |

위 실험은 배열의 크기를 100,000 으로 고정한 채 인풋 배열의 중복 발생률을 조정하며 각 정렬 방식의 시간을 측정하였다. 배열의 크기는 고정한 채 랜덤 숫자의 range 를 조정하여 중복 발생률을 조정하였다. 중복 발생률은 java 의 HashSet 을 import 하여 인풋을 하나씩 집어 넣고, 만일 동일 원소가 존재하면 duplicateCount 를 하나씩 증가시켜 마지막에는 duplicateCount/array.length 를 return 하는 방식으로 구현하였다.

실험 결과 중복 발생률이 0.998 이상일 때부터 quick sort 의 시간이 급격하게 느려졌다. 또한, 다른 정렬 방식들을 실험해보았는데 quick 을 제외한 방식 중 merge sort 가 가장 빠른 것으로 나와 merge, quick 이 두가지를 집중적으로 비교하였다. 실험 결과 Search function 에서는 인풋 배열의 중복 발생률이 0.998 이상이면 'M'을 리턴한다.

c. 입력의 **정렬 비율이 0.993 이상**이면 **insertion sort** 사용

| | 0.990 | 0.993 | 0.995 |
|---|---|---|---|
| Insertion | 26.26 | 22.74 | 15.80 |
| Quick | 21.42 | 30.53 | 38.96 |

마지막으로는 이미 정렬된 A[i]와 A[i+1] 쌍의 개수 및 비율에 대해 정렬 시간을 알아보는 실험을 진행하였다. getSortedRatio 함수를 통해 먼저 배열을 한 번 iteration 하며 inversion 된 (A[i] > A[i+1]) 쌍의 개수를 구한 뒤, 1 - (number of inversions) / (number of pairs) 를 return 하여 정렬 비율을 구한다. 입력의 정렬 비율이 높아질수록 Insertion sort 의 시간이 가장 빠른 것을 발견하였고, 나머지 sorting method 중 두 번째로 빠른 quick sort 와의 집중 비교를 하였다. 실험 결과 정의한 정렬 비율이 0.993 이상이면 insertion sort 를 사용하는 것이 더 빠른 것을 발견하여 Search 함수에서 'I'를 return 하도록 구현하였다.

4. **Search 동작 시간 분석** (모든 정렬을 일일이 수행하는 것과 Search 를 동작시키고 출력되는 정렬을 한번만 실행시킨 것의 시간을 비교해봅니다.)

Search 함수는 위 a, b, c 실험에 필요한 파라미터들을 한 번의 배열 전체 iteration 을 통해 구할 수 있도록 getRatios 함수를 구현하였다. 한 번의 iteration 을 통해 배열의 max 자릿수, duplicate ratio, sorted ratio 를 구할 수 있고, 이를 double[]로 return 하도록 구현하여 총 O(N) 시간이 걸린다. 하이퍼 파라미터들이 들어있는 배열을 받은 search 함수는 위 실험 결과에 따라 알맞은 정렬 방법을 출력한다. 만일 위 세 가지에 해당하는 것이 아니라면, 모든 정렬 방법 중 평균적으로 가장 빠른 quick sort 를 return 하도록 구현하였다.

| | 평균적인 경우 | 최대 자릿수 <= 9 | 중복 발생률 >= 0.998 | 정렬 비율 >= 0.993 |
|---|---|---|---|---|
| 모든 정렬 방식 수행 | 11807.32<br>(80.38) | 11798.53<br>(73.64) | 10734.23<br>(69.42) | 1750.32<br>(80.42) |
| **Search 수행** | **33.58** | **36.58** | **23.83** | **27.48** |

실험 방식은 각 경우에 대해 4 번째 이후 시행부터의 수행 시간을 합하여 모든 정렬 방식에 해당하는 시간을 구했고, 20 번의 수행 시간의 평균으로 계산하였다. 배열 크기는 100,000 으로 고정하였다. 또한, 괄호 안의 숫자는 가장 시간이 오래 걸리는 bubble, insertion sort 의 수행 시간을 제외한 시간 합의 평균을 적어 더 객관적인 비교가 가능하도록 하였다.
실험 결과, 모든 정렬 방식을 수행하여 가장 빠른 시간을 구하는 것보다 search 함수를 수행한 뒤, 이 결과로 출력된 정렬 방식으로 정렬하는 시간이 더 빠른 것을 확인할 수 있다.