**<HW4 Report>**

2013-12815 이동주

**1. Objectives:**

기본적인 정렬 알고리즘을 이해하고 구현함으로써 코드 작성시 의도한 동작 방식에 대해 기술한다. 정렬 알고리즘이 옳게 동작함을 테스트 코드를 통해 확인한 뒤, 다양한 Dataset에 대한 수행 속도를 비교함으로써 개선된 알고리즘이 실질적으로 얼마만큼의 수행 속도 차를 야기하는지 살펴보려 한다. 또한, 어떤 상황에도 최적인 정렬 알고리즘이 존재하지 않는다는 것을 아는 바, 테스트를 통해 어떤 상황에 특정 알고리즘이 대체로 적합한지 조사한다.

**2. Implementation:**

정렬 알고리즘의 구현은 수업 내용을 기반으로 하되 자세히 기술되지 않은 부분에 대해서는 대체로, 고민을 통해 직접 작성했다. 일부의 경우, 예컨대, Radix Sorting에서 자리 수의 빈도수를 통해 index를 참조할 수 있는 지에 대한 아이디어 등에 대해, 직관이 부족해 구현이 어려웠던 특정한 상황은 Michael T. Goodrich의 저서 <Data Structures & Algorithms>의 내용을 참조한 바 있다. 모든 경우에 대해서 코드를 직접적으로 참조한 부분은 없다. 이하에서 6개의 정렬 알고리즘에 대한 구현을 설명하겠다.

**1) Bubble Sort:**

인접한 두 data의 값을 비교해 더 큰 data가 우선하는 경우 그들간의 순서를 바꿈으로써 동작한다. 1번의 순회에서 가장 최대값인 data가 dataset의 끝(가장 오른쪽)까지 이동하게 된다. 즉, 길이가 n인 dataset에 대해 i번의 순회가 끝난 뒤에는 우측부터 크기가 큰 순서대로 i개의 data의 정렬이 완료되었음을 확신할 수 있다.

dataset[0, n-1]의 $0_{th}$ data부터 $(n-2)_{th}$ data까지 순회하면서 $i_{th}$ data와 $(i+1)_{th}$ data의 대소를 비교한다. 또한 우측에 정렬이 완료된 i개 원소는 제외하도록 했다. 따라서 [1, n-1]까지 순회하는 for문에 nesting된 [0, n-i-1]까지 순회하는 for문 내에서 $j_{th}$ data와 $(j+1)_{th}$ data를 비교하고 그 대소가 순서에 반대하는 경우 swap한다.

**2) Insertion Sort:**

정렬은 i번째 data를 그 전까지 정렬된 (i-1)개의 data들과 비교하여 옳은 자리에 삽입하는 방식으로 이루어진다. Dataset에 대한 i[1, n-1]번째 순회가 이루어질 때마다 i개의 data가 정렬되었다고 확신할 수 있으며, 이는 1번째 순회에 대해서 length=1인 subArr는 정렬된 것이므로 비약이 없다.

i번째 data가 정렬이 완료된 왼쪽의 subArr의 가장 큰 수인 (i-1)번째 data보다 작다면 옳은 자리를 찾아 탐색을 시작한다. 대소 비교를 통해 들어갈 자리를 먼저 찾은 뒤 해당하는 자리보다 우측에 있는 data들을 오른쪽으로 한 칸씩 이동시킨다. 이 때, i번째 data를 시작으로 옳은 자리까지 이동할 때까지 swap해주는 방식을 이용함으로써 연산을 가볍게 할 수 있었다.

또한, i번째 data가 그 전까지 정렬된 data의 rightmost data보다 작지 않다면, 옳은 자리에 있는 것이므로 i번째 data에 대한 정렬을 종료한다. 이상의 절차를 통해, [0, i]인 (i+1)개의 data가 정렬되었다고 확

신할 수 있으므로 i를 1증가시키고 다음 data의 정렬을 시작하는 방식으로 알고리즘을 설계했다.

**3) Heap Sort:**

별도의 클래스로 독립시킬 필요가 있다고 여겨 생성한 HeapSort클래스를 내에 sort를 실행하는 public static method로 dataset의 length가 2 이상인 경우에 대해서만 dataset을 인자로 주었다. length가 0이거나 1인 경우엔 정렬이 필요하지 않으므로 dataset을 그대로 리턴함으로써 정렬을 종료한다.

정렬이 시작되면 우선 기존의 dataset을 percolateDown하며 maxHeap으로 바꾼다. 개념적 동작방식에 따라 트리 구조의 도입을 고민했으나 필요한 연산이 percolateDown 밖에 없음을 이해하고 인자로 주어진 배열 dataset을 그대로 maxHeap으로 변환해 사용했다.

maxHeap을 만족시키고 나면, 최대값이 보장된 root를 맨 뒤로 보냄으로써 maxHeap에서 배제하고 (Deletion하는 결과와 같다) size를 1 줄인 나머지를 다시 percolateDown함으로써 maxHeap을 만족시킨다. 이를 (size - 1)번 반복함으로써 dataset의 오른쪽에 크기가 줄어드는 maxHeap에 대한 max data를 차곡차곡 쌓을 수 있고 남은 1개의 원소는 당연히 min data이므로 정렬이 완료된다.

**4) Merge Sort:**

Merge Sort는 원리상 재귀적으로 동작하며 divide-and-conquer 방식에 의한다. 따라서, 1. Divide, 2. Recursive call, 3. Conquer(Merge)의 절차에 따라 알고리즘을 작성했다.

첫째, Divide에 대하여, dataset의 크기가 <= 1인 경우에는 더 이상 나눌 수 없으므로 Divide를 실시하지 않는다. dataset의 크기가 1일 때 (leftmostIndex == rightmostIndex)인 점을 판단 기준으로 했다. 더이상 나눌 수 없는 subArr는 2. Recursive, 3. Merge 할 필요가 없으므로 현재 function call에서 후행하는 단계로 진행하지 않고 return함으로써 이전에 재귀적으로 function call했던 위치로 반환된다.

둘째, Recursive call에 대하여, 현재 정렬의 대상이 되는 subArr에 대해 [leftmostIndex, mid]와 [mid+1, rightmostIndex]의 두 subArr로 나누어 각각 recursively call한다. 이로써 1~2 단계에 의해 크기가 1이거나 0이 될 때까지 재귀적으로 함수 호출되게 되며 반환된 것들은 모두 크기가 1이거나 0이라고 확신할 수 있다.

셋째, Conquer(Merge)에 대하여, 이 곳에 최초로 도달한 merge의 대상이 되는 subArr는 각각 크기가 <= 1이다. 최초의 Merge 대상이 된 두 subArr는 또한 0 또는  1인 크기에 의해 각각 이미 정렬되어 있다. 또한, 이 두 subArr는 이 곳에서 합쳐지며 크기가 두배인 정렬된 subArr가 되므로 병합하게 되는 두 subArr는 모든 경우에 대해 각각 정렬되어 있다고 확신할 수 있다. 각각의 subArr를의 index를 두고 그들이 가리키는data중 작은 원소를 먼저 채우고 옮겨진 subArr의 인덱스를 한 칸 뒤로 이동하는 것을 반복한다. 이를 두 subArr중 하나가 모두 옮겨질 때까지 반복한 뒤, 각각의 index가 가리키는 대상 중 크기가 작은 것을 모두 오른쪽에 차곡차곡 채워 넣는다. 이 때, 둘 모두 비어있는 일은 없기 때문에 left/right subArr가 비어 있는지 검사하는 순서는 상관이 없다.

**5) Quick Sort:**

이 정렬 알고리즘 역시 반복적으로 pivot을 설정하면서 재귀적으로 동작하기 때문에 base case와 partitioning 부분을 나누어 구현했다. 재귀 선언이 종결되는 조건은 (leftmostIndex == rightmostIndex)

로 이전의 논의와 동일하다. Partitioning에 필요한 pivot은 첫 data와 마지막 data가 대표성이 없을 수 있으므로 두 데이터의 중간값으로 설정했다. 탐색의 Index에 대하여 leftIndex는 leftmostIndex에서부터 출발해 오른쪽으로 진행하고, rightIndex는 rightmostIndex에서 출발해 왼쪽으로 진행한다. Recursive call의 종결 조건에 걸리지 않은 경우 Partitioning을 진행하고, Partitioning의 종결조건은 leftIndex와 rightIndex가 교차하는 것으로 한다.

본격적인 정렬의 원리는 다음과 같다. dataset의 왼편에서 leftIndex가 pivot보다 작지 않은 data를 찾으면 그곳을 가리킨 채로 멈춘다. dataset의 오른편에서 출발한 rightIndex 또한 pivot보다 크지 않은 data를 찾으면 그곳을 가리킨 채로 멈춘다. 만약 두 인덱스가 아직 교차하지 않았다면, 이는 잘못된 partition에 위치한 두 data를 가리키고 있는 것을 의미하므로 두 data를 swap함으로 partitioning했다.

**6) Radix sort:**

양수와 음수를 처리하는 과정에 있어 나머지 연산에서 음수가 되는 것을 발견했다. 다양한 방법이 도입될 수 있겠지만, 정렬의 전처리 과정에서 양수와 음수를 따로 구분해서 모두 양수로써 절대값에 따라 정렬한 뒤 다시 음수로 바꾸어 주며 역순으로 모았다.

RadixSort클래스의 sort함수에서는 양수/음수 각각에 대해 같은 원리로 정렬이 수행된다. 우선 최대 자릿수를 구한 뒤 각 자릿수에 대한 순회를 실시한다. 자리 수를 이하에서 digit이라고 칭하겠다. 10가지 digit에 대한 0~9의 bucket과 data의 digit만 저장하는 remainderArr를 선언해두고 각 자릿수에 대한 정렬이 완료되면 sortedArr에 매번 결과값을 재배열한다.

자릿수를 초점하기 위해 divideOperand를 1에서 시작해 10씩 곱해주며 각 data를 나누어 조사하는 자릿 수의 digit을 1의자리로 낮춘 뒤 10을 나머지연산해 digit을 얻어낸다. 이를 remainderArr에 모두 모으면서 자릿수의 빈도수를 bucket의 원소를 증가시키며 조사한다. 모든 data에 대한 digit 추출이 끝나면 bucket에는 0~9 자릿수의 빈도수가 조사되어 있고 remainderArr에는 각 data의 digit만이 저장되어 있다. 자릿수를 기준으로 정렬을 할 때에 별도의 정렬 알고리즘을 도입하지 않기 위해서 빈도수만으로 이전 자릿수 정렬의 순서를 기준으로 Stable sorting하기 위해 bucket에 대해 빈도수를 누적해서 0~9까지 어떻게 누적되고 있는지를 저장한다. 이전 단계의 data 정렬을 담고 있는 sortedArr에서 digit만이 추출된 remainedArr 가장 오른쪽에서부터 각 digit의 빈도수를 참조하면 가장 마지막의 오는 해당 digit을 가지는 원소가 몇번째로 등장하는지를 알 수 있다. 따라서 sortedArr의 현재 자릿수에 대한 정렬을 우측에서부터 좌측으로 차곡차곡 쌓아나갈 수 있다. 이는 기존의 순서가 유지되어야 한다는 Radix Sort에 적합한 방식이라고 생각한다.

**3. Results:**

제공된 Testcase에 대하여 1~40번 case에 대해 1초 미만, 41번부터 50번까지는 7-8초 정도의 수행시간이 걸렸으며 제한된 기준에 넉넉하게 부합했다. 추가적으로 정렬 알고리즘에 대한 수행을 비교해보려 한다. 각각의 경우에 대해서 10개씩의 test set에 대해 동일하게 내린 첫번째 명령을 수합했다.

| | | | | |
|---|---|---|---|---|
| **1. Size of data** | $-10^2<n<10^2$, $-10^3<n<10^3$, $-10^4<n<10^4$, $-10^5<n<10^5$ | | | |
| **2. Distribution of data** : | 1) **Density** : | Sparse(25%) | <---Mid(50%)---> | Dense(75%) |
| (수행하지 못했습니다) | 2) **Order** : | Sorted | <----Random----> | Reverse-sorted |

1. Size of data에 대하여

| | $-10^2<n<10^2$ | | $-10^3<n<10^3$ | | $-10^4<n<10^4$ | | $-10^5<n<10^5$ | |
|---|---|---|---|---|---|---|---|---|
| | 평균 | 편차 | 평균 | 편차 | 평균 | 편차 | 평균 | 표준편차 |
| **1) Bubble Sort** | 0.5 | 0.52 | 9.9 | 1.10 | 133 | 2 | 16936.8 | 195. |
| **2) Insertion Sort** | 0.5 | 0.52 | 8.6 | 0.51 | 78.2 | 6.34 | 5968.6 | 20.6 |
| **3) Heap Sort** | 1.1 | 0.31 | 2.3 | 0.67 | 6.2 | 1.78 | 25.2 | 1.64 |
| **4) Merge Sort** | 0.9 | 0.56 | 2.9 | 0.31 | 7.8 | 1.30 | 41.4 | 2.19 |
| **5) Quick Sort** | 0.9 | 0.31 | 1.9 | 0.31 | 5.8 | 0.44 | 34.2 | 9.31 |
| **6) Radix Sort** | 1.2 | 0.63 | 1.9 | 0.56 | 9.8 | 0.44 | 25.6 | 0.54 |

* 2. 1) density 는 Mid(50%)로 고정, 2.2) Order은 Random으로 고정

2. Distribution of data 중 1) Density에 대하여 (수행하지 못했습니다)

| | $-10^2<n<10^2$ | $-10^3<n<10^3$ | $-10^4<n<10^4$ | $-10^5<n<10^5$ | $-10^6<n<10^6$ |
|---|---|---|---|---|---|
| **1) Bubble Sort** | | | | | |
| **2) Insertion Sort** | | | | | |
| **3) Heap Sort** | | | | | |
| **4) Merge Sort** | | | | | |
| **5) Quick Sort** | | | | | |
| **6) Radix Sort** | | | | | |

* 2. 1) density 는 Mid(50%)로 고정, 2.2) Order은 Random으로 고정

2. Distribution of data 중 2) Order에 대하여 (수행하지 못했습니다)

| | $-10^2<n<10^2$ | $-10^3<n<10^3$ | $-10^4<n<10^4$ | $-10^5<n<10^5$ | $-10^6<n<10^6$ |
|---|---|---|---|---|---|
| **1) Bubble Sort** | | | | | |
| **2) Insertion Sort** | | | | | |
| **3) Heap Sort** | | | | | |
| **4) Merge Sort** | | | | | |
| **5) Quick Sort** | | | | | |
| **6) Radix Sort** | | | | | |

* 2. 1) density 는 Mid(50%)로 고정, 2.2) Order은 Random으로 고정

**4. Summary & Limitations**

생각보다 구현에 걸린 시간이 길어 예정한 테스트를 모두 진행하지 못하고 과제를 제출하게 되어 아쉽다. 예상되는 결과로써 Distribution에 따라 비효율성을 초래하는 정렬 알고리즘에 대해 분석해보지 못해 아쉽다.