**자료구조 HW4 Report** (양기창 2015-13078)

**1. 알고리즘 동작 방식**

Time complexity는 average case를 기준으로 작성하였다.

1.1. Bubble Sort:

매 iteration마다, 배열의 관심 영역에서 가장 큰 원소를 해당 영역의 가장 뒤쪽으로 보내고, 해당 원소를 제외한 부분으로 관심 영역을 줄여나가는 정렬 방법이다. 배열의 관심 영역은 처음에는 전체 배열에서 시작하여, 점점 작아져 가장 앞 쪽 원소만을 포함한 영역으로 축소된다. 그에 따라 배열은 맨 뒤부터 점점 정렬된다.

관심 영역에서 가장 큰 원소를 해당 영역의 가장 뒤쪽으로 보내는 방법은 다음과 같다. 관심 영역의 가장 앞부터 시작하여 관심 영역의 가장 뒤까지 진행하며, 연속된 두 원소를 비교하여 크기 순에 맞지 않다면 swap한다. 그러면 찾고자 하는 가장 큰 원소는 비교 과정에서 계속 살아남아 가장 뒤까지 옮겨지게 된다.

1.2. Insertion Sort:

i번째 iteration에서, 앞 i개의 원소가 정렬된 상태로 시작하여, i+1번째 원소가 들어가야 할 위치를 찾아 그 위치에 위치시킨다. 즉, 배열의 맨 앞쪽부터 정렬되기 시작한다.

i+1번째 원소가 들어가야 할 위치를 찾을 때는, 첫 번째 원소부터 i번째 원소까지의 배열에 binary search를 이용한다. 위치를 찾았으면, i번째 원소부터 해당 위치의 원소까지를 뒤쪽으로 한 칸씩 shift한 후 i+1번째 원소를 해당 위치에 삽입한다.

1.3. Heap Sort:

먼저, 마지막 원소의 parent부터 시작하여 root까지 각각 percolate down함으로써, 주어진 배열이 max heap이 되도록 만든다. 마지막 원소의 parent가 heap property를 위배할 가능성이 있는 첫 번째 원소이기 때문에 위 과정을 통해 heap property를 만족시키는 영역이 점점 확장되어, 결국 배열 전체가 heap property를 만족시키게 된다.

Max heap을 만든 후에는, 매 iteration마다 root와 관심 영역의 마지막 원소를 swap한 후, 관심 영역에서 마지막 원소(기존의 root)를 제외시키고, root(기존의 마지막 원소)를 관심 영역 내에서 percolate down한다. 그러면 관심 영역이 점점 줄어들고, 관심 영역 밖(배열의 뒤쪽)에는 정렬된 원소들이 위치하게 된다.

Percolate down은 해당 원소가 heap property를 만족시키도록 heap 내에 위치시키는 operation이다. 해당 원소를 해당 원소의 두 children 중 큰 child와 swap하고, child가 된 해당 원소를 다시 percolate down하는 방식으로 진행된다.

1.4. Merge Sort:

주어진 배열을 동일한 길이의 두 영역으로 나누어 각각을 merge sort한 후, 둘을 merge하는 recursive한 방식으로 진행된다. Recursive call을 반복하여 나누어진 영역의 길이가 1이 되면, merge를 시작하게 된다.

두 영역의 merge는 다음과 같이 진행된다. 결과를 담을 배열을 새로 생성한 후, 각 영역의 첫 번째 원소들부터 비교하기 시작하여 작은 원소를 결과 배열에 담는다. 이후, 작은 원소를 가지고 있던 영역의 다음 원소를 비교 대상으로 삼는다. 이와 같은 과정을 반복하여 두 영역의 원소들을 모두 결과 배열에 담은 후, 결과 배열을 기존의 배열에 복사한다.

1.5. Quick Sort:

주어진 배열의 원소 중 하나를 pivot으로 삼아, pivot 이전에는 pivot보다 작은 원소들이, pivot 이후에는 pivot보다 큰 원소들이 위치하도록 만드는 ‘partition’을 진행한다. 이후, pivot의 위치 이전/이후 영역에 대해서 각각 quick sort를 recursive하게 진행한다.

Partition 방식은 다음과 같다. 주어진 부분의 마지막 원소를 pivot으로 정하고, 배열을 순차적으로 살펴보며 배열의 앞쪽에 pivot보다 작은 원소들을 쌓아나간다. 이 때 ‘pivot보다 작은 원소들 중 가장 뒤쪽 원소’의 위치를 tracking한다. 살펴보는 새 원소가 만약 pivot보다 작다면, tracking하는 위치를 1 늘리고, 해당 위치의 원소와 새 원소를 swap한다. 그러면 tracking하는 위치는 계속해서 ‘pivot보다 작은 마지막 원소’가 될 것이고, 해당 위치 이전 원소들은 모두 pivot보다 작을 것이다. 이를 반복한 후, 마지막으로 tracking한 위치의 원소와 pivot을 swap한다.

1.6. Radix Sort:

숫자들의 가장 낮은 자릿수부터 기준으로 삼아 정렬(stable sort)한 후, 점차 높은 자릿수를 기준으로 정렬한다. 본 과제에서는 binary representation의 1 byte(8 bit)를 하나의 자릿수로 보는 경우와 decimal representation의 하나의 digit을 하나의 자릿수로 보는 경우를 모두 구현해놓았다. 여기에서는 binary representation 기준을 default로 삼고 이에 대한 설명을 하였다.

하나의 byte에서는 256가지의 bit 패턴이 나온다. 따라서 숫자들을 살피며 각 패턴 별 등장 횟수를 센다. 횟수들을 크기 순으로 누적시키면, 각 패턴을 가진 숫자들이 새로운 배열의 어느 위치에 들어가야 할지 알 수 있다. (다만 most significant byte의 경우, 2’s complement의 음수 표현을 고려하여 0x00~0x7f가 0x80~0xff보다 큰 쪽에 위치하도록 해준다.) 이 위치들을 이용하여 해당 byte를 기준으로 정렬된 새로운 배열을 얻는다. 이를 각 byte에 대해 반복하다.

**2. 알고리즘 동작 시간**

실험 시 정확한 시간 측정을 위해 뼈대 코드의 System.currentTimeMillis() 대신 System.nanoTime()을 이용하여 나노초 단위로 구한 후, 밀리초 단위로 환산하였다.

먼저, 의 정수를 생성하도록 고정해놓고, 데이터의 개수를 4000~40000개까지 4000단위로 증가시키며, 각각에 대해 6개의 알고리즘을 50회씩 실행하였다. 결과는 다음 표와 같다. (CV는 Coefficient of Variation으로, Sample mean / Sample standard deviation이다.)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data 수 | 4000 | 8000 | 12000 | 16000 | 20000 | 24000 | 28000 | 32000 | 36000 | 40000 |
| Bubble |  |  |  |  |  |  |  |  |  |  |
| Mean | 20.33 | 89.39 | 253.25 | 417.07 | 683.56 | 1005.17 | 1295.79 | 1993.78 | 2303.64 | 3194.31 |
| CV | 0.40 | 0.15 | 0.29 | 0.10 | 0.08 | 0.08 | 0.05 | 0.23 | 0.06 | 0.20 |
| Insertion |  |  |  |  |  |  |  |  |  |  |
| Mean | 2.32 | 8.26 | 25.44 | 32.22 | 55.80 | 77.42 | 92.50 | 131.56 | 157.98 | 246.08 |
| CV | 0.10 | 0.27 | 0.43 | 0.20 | 0.29 | 0.25 | 0.15 | 0.16 | 0.10 | 0.26 |
| Heap |  |  |  |  |  |  |  |  |  |  |
| Mean | 0.44 | 1.20 | 1.91 | 1.89 | 2.38 | 2.89 | 3.95 | 4.20 | 4.76 | 5.14 |
| CV | 0.30 | 0.59 | 0.91 | 0.29 | 0.12 | 0.21 | 0.41 | 0.29 | 0.26 | 0.15 |
| Merge |  |  |  |  |  |  |  |  |  |  |
| Mean | 3.14 | 3.83 | 4.58 | 5.15 | 5.74 | 5.60 | 5.85 | 6.42 | 6.74 | 7.72 |
| CV | 1.10 | 2.35 | 1.54 | 1.25 | 1.41 | 0.80 | 0.51 | 0.33 | 0.31 | 0.16 |
| Quick |  |  |  |  |  |  |  |  |  |  |
| Mean | 0.43 | 1.00 | 0.93 | 1.27 | 1.78 | 1.93 | 2.19 | 2.53 | 3.08 | 4.07 |
| CV | 2.19 | 0.64 | 0.04 | 0.09 | 0.30 | 0.11 | 0.11 | 0.12 | 0.11 | 0.22 |
| Radix |  |  |  |  |  |  |  |  |  |  |
| Mean | 0.25 | 0.50 | 0.91 | 0.92 | 1.11 | 1.45 | 1.28 | 1.50 | 1.40 | 1.12 |
| CV | 0.07 | 0.52 | 1.27 | 0.26 | 0.39 | 1.94 | 0.40 | 0.59 | 0.82 | 0.69 |

데이터가 수가 적을 때는 Heap, Merge, Quick, Radix 간의 차이를 확인하기 힘들다. 따라서 이 4개의 알고리즘에 대해서만 데이터 수를 늘려, 100000~1900000개까지 200000 단위로 늘려가며 추가적인 실험을 진행하였다. 아래 표는 평균 동작 시간만을 나타낸 표이다.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data 수 | 0.1M | 0.3M | 0.5M | 0.7M | 0.9M | 1.1M | 1.3M | 1.5M | 1.7M | 1.9M |
| Heap | 16 | 63 | 121 | 161 | 227 | 298 | 356 | 419 | 531 | 698 |
| Merge | 20 | 77 | 132 | 163 | 199 | 280 | 302 | 362 | 502 | 630 |
| Quick | 12 | 43 | 59 | 88 | 101 | 133 | 156 | 186 | 200 | 284 |
| Radix | 2 | 6 | 11 | 19 | 20 | 23 | 32 | 33 | 41 | 53 |

위 표와 그래프들에서 보이듯이, 성능이 좋고 데이터 개수 증가에 대한 동작 시간 증가 비율이 작은 알고리즘부터 나열하면 순서이다.

이는 대체로 이론적인 time complexity와 일치하는 결과이다. Bubble sort와 Insertion sort의 경우 그래프의 꼴이 이차 함수 모양이므로 average time complexity 을 확인할 수 있다. 다만 Insertion sort 구현 시 linear search가 아닌 binary search를 이용하였기 때문에 bubble sort에 비해 좋은 성능을 보였다. Radix sort는 이기 때문에 오른쪽 그래프에서 거의 linear하게 증가하는 모습을 볼 수 있다. Heap, Merge, Quick sort의 경우 언뜻 보기에는 Radix sort와 비슷하게 linear한 모양을 띄는 것처럼 보이지만, 데이터 수가 늘어남에 따라 기울기가 조금씩 증가하는 것을 볼 수 있다. 이는 임을 암시한다.

다음으로, Radix sort를 decimal representation 기준으로 구현한 경우와 binary representation 기준으로 구현한 경우를 비교해보았다. 사이의 데이터를 생성하여 동작 시간을 비교하였다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Size | 1E+2 | 1E+3 | 1E+4 | 1E+5 | 1E+6 | 1E+7 |
| Decimal | 0.37 | 1.87 | 10.64 | 105.77 | 1152.06 | 11949.50 |
| Binary | 0.10 | 0.38 | 2.02 | 17.63 | 145.63 | 1939.68 |

전반적으로 Binary representation을 이용할 때가 성능이 훨씬 좋은 것을 확인할 수 있다. 이를 통해 컴퓨터는 bit 단위(혹은 byte 단위) 처리에 최적화 되어있음을 확인할 수 있다. 이는 십진수의 곱셈/나눗셈과 달리, 이진수의 곱셈/나눗셈은 shift operation으로 처리할 수 있기 때문으로 추정된다.