HW4 보고서

신제민 2017-13717

1. 정렬 방법에 대한 설명

- Bubble Sort

기본적인 정렬 알고리즘 중 하나로, 배열의 처음부터 원소 두 개를 뽑아 비교를 한 후 값이 앞에 있는 원소가 더 크면 자리를 바꾸고 최대값을 마치 공기 방울이 바닥에서 떠오르듯 배열의 맨 뒤로 보낸다. 이후 배열의 크기를 1개씩 줄여서 다시 한번 진행하는 식으로 배열의 크기 만큼 반복한다. 정렬이 거의 되어 있는 경우 교환이 더 조금 일어나 수행시간이 감소한다.

- Insertion Sort

기본적인 정렬 알고리즘 중 하나로, Bubble Sort, Selection Sort와는 반대로 정렬이 된 부분을 배열의 앞부분에 둔다. 배열 앞에서부터 원소를 뽑아서 정렬된 부분의 원소와 비교를 통해 적절한 위치에 원소를 위치시킨다. 이 과정에서 shift가 큰 overhead를 발생시킬 수 있다. 따라서, 정렬이 거의 되어있는 상태에서 효율이 좋다. 마찬가지로 해당 과정을 배열의 크기만큼 반복해서 정렬을 수행한다.

- Heap Sort

주어진 배열을 buildHeap()매서드를 통해 힙 특성을 만족하는 Max Heap구조로 바꾸고, 배열의 가장 마지막에 있는 Min 원소를 Root의 Max 원소와 교환하고 Percolate Down()매서드를 수행하여 힙 특성을 유지하면서 차례로 정렬을 수행한다. Percolate Down()매서드는 재귀적으로 짤 수 있지만, 재귀적으로 수행하는 경우 프로시저 call, return에 따른 메모리 접근이 일어나므로 추가적인 overhead가 발생하여 수행 시간이 증가한다. 따라서 효율이 극대화 된 Heap Sort를 수행하기 위해, 비재귀형식으로 Percolate Down()을 수행했다. 한편, 어떤 배열이 주어지든 힙 구조로 변환 후, 정렬을 수행하기 때문에 Input 배열의 형태에 따른 편차가 크지 않은 장점이 있다. 동일한 원소가 많은 경우 percolate Down()에 따른 교환이 적게 일어나 수행 효율이 좋다.

- Merge Sort

주어진 배열을 절반 나누어 각각 정렬을 수행하고 각 정렬된 배열의 원소를 정렬하면서 합치는 수행을 재귀적으로 반복한다. 일반적으로 새로운 배열을 만들어서 정렬을 수행한 후 다시 원래 배열에 옮기는 식으로 수행하여 합치는 과정에서 overhead가 추가로 발생한다. 이를 방지하기 위해 input배열 크기의 새로운 배열 value2[] 하나를 만들어서, 배열의 절반 중 앞부분(p ~ r)은 value2 []배열에 정렬시키고, 배열의 절반 중 뒷부분(r+1 ~ q)은 input[]배열에 정렬시킨다. 그리고 value2[] 배열의 (p~r)부분 원소와 input[] 배열의 (r+1~q)부분 원소를 차례로 하나씩 비교하면서 작은 값부터 input[]배열의 맨 앞(p)부터 채운다. 합치는 과정이 완료되면 input[]배열의 원소들이 정렬된 모습으로 결과가 나타난다. 해당 방식을 재귀적으로 수행함으로써 새로운 배열에서 input[]배열에 옮기는 과정에 소요되는 overhead가 제거된 merge Sort를 수행할 수 있다. 한편, input[]의 형태와 관계없이 수행이 동일하게 일어나므로, input[] 형태에 따른 편차가 크지 않은 장점이 있다.

- Quick Sort

기준 원소를 잡고 input[] 배열을 기준 원소보다 작은 부분과 큰 부분으로 나누는 partition작업을 수행한다. 이후 각 부분에 대해서 재귀적으로 Quick sort를 수행한다. Quick Sort는 일반적으로 효율이 굉장히 좋지만 2가지 경우에 대해 비효율적으로 수행된다.

1. 정렬이 거의 되어있는 상태에서 맨 뒤 원소를 기준으로 뽑는 경우

맨 뒤 원소가 아닌 random으로 매번 기준 원소를 뽑아 정렬을 수행하도록 하여 해결하였다.

2. 동일한 원소가 많이 포함된 경우

기준 원소와의 비교 과정에서 동일한 원소가 나타나면 random()함수를 통해 1/2의 확률로 배열의 앞, 뒤 부분에 할당되도록 하여 확률적으로 절반씩 나뉘도록 하여 비효율을 제거하였다.

- Radix Sort

Input[]의 원소 값의 절대값 중 최대값의 자릿수를 기준으로 정렬을 진행할 횟수 k를 결정한다. 각 자릿수를 positive(cnt_pos[10]), negative bucket(cnt_neg[10])에 나누어 할당하고, negative에서는 index = 9 8 7 … 0 순으로 누적합(cum_neg)을 계산하고, 그 누적합(cum_neg[0])을 positive(cum_pos[0] = cnt_neg[0] + cum_neg[0])에서 이어받는다. positive에서는 0 1 2 … 9 순으로 누적합을 계산하여 전체 input[] 배열에 대한 해당 자릿수의 숫자의 각 위치를 할당한다. (Ex. Input[cum_pos[1]] = Input배열에서 양수이면서, 해당 자릿수가 1인 element가 할당될 위치) 한편, 이 과정을 input[]배열에 바로 할당하게 되면 값이 왜곡되기 때문에 input[]배열과 같은 크기를 같은 tmp[] 배열을 새로 할당하여 그곳에 정렬을 수행한 후 최종적으로 다시 input[]으로 이동시키는 overhead가 발생한다. 한편, 최대값의 자릿수 k가 정렬 횟수를 결정하기 때문에 숫자를 뽑는 범위가 radix 정렬의 수행 시간에 영향을 미칠 수 있다.

2. 실험 방법

Case 1~4에 대해서는 1000개, 10000개, 100000개, 1000000개의 원소를 겹치는 숫자가 거의 없도록 범위를 설정하여 무작위로 뽑는다. Case 뽑힌 원소의 배열을 6가지 방법의 정렬 방법을 적용하여 'ms' 단위로 수행시간을 측정한다. 해당 실험은 총 5회씩 반복하여 각 방법론에 대해 평균, 최솟값, 최댓값, 표준편차를 구한다. 총 10회의 실험은 컴파일 특성상 같은 배열에 대해 같은 정렬 방법을 연속적으로 적용하는 경우 성능이 달라지는 문제를 피하기 위해 하나의 배열당 정렬 방법론을 한번씩만 적용하고 다시 배열을 뽑아 수행한다.

Case 5~9에 대해서는 10000개의 원소를 뽑되, 뽑는 범위를 다르게 설정하여 범위에 따른 각 정렬 방법의 수행 효율을 비교한다. 뽑는 범위를 좁히는 것은 크게 두가지 측면에서 해석할 수 있다.

1) 동일 원소수의 증가

뽑는 범위가 줄어든다면 동일한 원소를 뽑을 확률이 증가된다.

2) 원소 자릿수의 감소

뽑는 범위를 0에 가깝게 줄인다면 뽑히는 원소의 최댓값이 줄어들어 radix sort에 영향을 줄 수 있는 최대 자릿수의 크기가 감소한다.

3. 실험 결과

	input(N)	range	Bubble	Insertion	Heap	Merge	Quick	Radix

mean	1,000	-30,000~30,000	24.4	11.6	1.2	2.6	4.6	4.6
min			21	10	1	2	3	4
max			29	14	2	3	8	6
std			3.578	1.517	0.447	0.548	1.949	0.894
mean	10,000	-300,000~300,000	336.4	40.4	5.8	5.4	9.2	39
min			248	29	4	3	6	29
max			440	48	7	8	13	49
std			91.002	8.385	1.304	1.949	3.114	9.138
mean	100,000	-3,000,000~3,000,000	23808.4	1501.8	27.4	25.6	42.2	124.4
min			23440	1471	25	24	35	121
max			24189	1525	28	28	66	128
std			352.039	20.278	1.342	1.517	13.405	2.608
mean	1,000,000	-30,000,000~30,000,000	-	152549	217	221	244	592
min			-	149096	213	220	219	587
max			-	155262	219	222	286	600
std			-	3148.903	3.215	1.000	36.828	7.234
mean	10,000	-30,000~30,000	241.8	27.6	4.6	4	7.8	25.6
min			234	24	4	4	5	23
max			248	31	6	4	10	29
std			5.02	3.05	0.894	0	1.924	2.702
mean	10,000	-3,000~3,000	284	31.8	4.6	4.8	7.4	25.6
min			239	24	4	4	5	21
max			381	41	7	7	12	43
std			62.234	6.87	1.342	1.304	2.793	9.737
mean	10,000	-300~300	280.2	25.2	4.4	4	7.8	20.2

min			242	23	4	4	6	18
max			427	29	5	4	10	24
std			82.068	2.49	0.548	0	1.643	2.387
mean	10,000	-30~30	239.2	24.4	4	3.6	8.8	15
min			237	22	4	3	7	14
max			243	26	4	4	11	18
std			3.033	1.517	0	0.548	1.789	1.732
mean	10,000	-3~3	237.8	23.2	4	3.2	10.4	7.6
min			230	21	4	3	9	7
max			243	26	4	4	12	8
std			5.586	2.168	0	0.447	1.14	0.548

4. 원소 수 N에 따른 효율성 비교(Case 1, 2, 3, 4)

case1을 제외하고 평균적인 수행 속도는 Heap = Merge > Quick > Radix >> Insertion >> Bubble 순으로 나타났다. 정렬의 시간 편차는 case 1,2,3의 경우 Merge = Heap < Radix <= Quick < Insertion < Bubble 순으로 나타났다. case 4의 경우 Merge < Heap < Radix < Quick < Insertion < Bubble 순으로 나타났다. 한편, Bubble sort의 경우 N = 100만일 때, 수행 시간이 200초 이상 걸려서 수행 시간을 null값으로 표현하였다. 따라서, 10만 이하의 원소 수에 대해서는 Merge와 Heap이 가장 효율이 좋았고, N = 100만에서는 Merge의 수행이 가장 효율이 좋았다.

5. 범위에 따른 효율성 비교(Case 5, 6, 7, 8)

Heap, Merge는 범위에 따라 크게 성능의 차이를 보이지 않았다. Quick의 경우 기존의 단점이 동일한 원소를 많이 포함하면 성능이 나빠지는 것이었는데, 결과도 마찬가지로 동일한 원소가 늘어날수록 수행 시간이 증가하였다. 그러나, 표준 편차는 오히려 뽑는 범위가 줄어듦으로써 대체로 감소하는 경향을 보였다. 이는 뽑는 범위가 감소하면서 표본 추출의 무작위성이 줄었기 때문일 가능성이 있다. 한편, Radix sort는 표본 추출의 범위가 줄어듦과 함께 수행 시간 및 표준 편차가 줄었다. 이는 최대 자릿수 k가 감소함에 따라 수행 횟수가 줄어든 요인이 크게 작용했을 것이다. Insertion sort와 Bubble sort도 마찬가지로 범위가 줄어듦과 함께 수행 시간 및 표준 편차가 줄었다. 이는 두 경우 모두 동일한 원소의 증가로 배열의 정렬의 정도가 높아짐으로써 나타난 효과로 해석된다.