자료구조 HW4 보고서

2015-15356 이준희

1. 정렬 알고리즘 동작 방식 ---

① Bubble Sort

Bubble Sort는 크게 이중 for loop으로 작동한다. 바깥쪽 루프는 아직 정렬되지 않는 배열의 크기를 하나씩 줄여주는 역할을, 안쪽 루프는 왼쪽부터 인접한 두 요소의 크기를 비교해, (현재 정렬되지 않은 부분에서) 가장 큰 수를 가장 오른쪽으로 이동시키는 역할을 한다. 배열크기를 n이라고 했을 때, 바깥쪽 루프는 총 (n-1)만큼 돌고, 안쪽 루프는 바깥쪽 루프가 돌때마다 매번 크기가 하나씩 줄어들어 (n-1), (n-2), ... , (n-(n-1))번 돈다. 안쪽루프를 기준으로 하여 본 알고리즘의 수행시간을 구해보면, (n-1)+(n-2)+ ... + 2 + 1 = n(n-1)/2에 의해 최종적으로 θ(n^2)가 됨을 알 수 있다. 배열 내부에서 swap이 이루어지기 때문에 in-place이며, 안쪽루프에서는 인접한 두 수 중 왼쪽에 위치한 값이 더 클 경우 swap을 수행하기 때문에 stable하다. -> [개선 가능] : 기존 bubble sort는 중간에 sorting이 완료되었다 할지라도 뒤쪽까지 무의미한 비교연산을 진행하여 항상 θ(n^2)만큼의 시간을 소요한다. 이러한 비효율을 해결하기 위하여 sorted 라는 boolean 값을 바깥루프와 안쪽 루프 사이에 위치시켜줄 수 있다. sorted는 안쪽루프에서 한번도 swap이 이루어지지 않았을 경우 true가 되고, 안쪽 for loop 말미에 sorted가 true일 경우 전제 sorting 함수에 대하여 return을 해주는 코드를 추가 해준다. 이렇게 하면 최선의 경우 θ(n)의 시간이 걸린다. (참고문헌-쉽게 배우는 알고리즘)

② Insertion Sort

Insertion Sort는 바깥쪽 for loop 하나(a), 그 안에 while loop 하나(b)를 지니고 있다. 바깥쪽 for loop(a)은 배열의 왼쪽부터 현재 정렬된 부분의 크기를 하나씩 늘려나가는 역할을 하고, 안쪽 while loop에서는 실제 정렬이 수행된다. (b) 현재 요소가 정렬된 부분의 가장 큰 요소보다 작다면, 자신보다 작거나 같은 요소를 만날 때 까지 왼쪽으로 한 칸 씩 훑는다(훑어진 요소들은 왼쪽으로 한 칸씩 shift된다). 자신보다 작거나 같은 요소를 만나면 while loop(b)을 빠져나와 해당 요소의 바로 왼쪽에 현재 요소값을 복사해 넣는다(삽입). (a)는 배열의 두 번째 요소에서부터 시작해 총 (n-1)번 돌며, (b)는 최선의 경우(이미 정렬이 되어있는 배열의 경우)0번, 최악의 경우(정렬이 반대로 되어있는 배열의 경우) 1 + 2 + 3 + ... + (n-1) = n(n-1)/2번 순환하게 된다. 따라서 본 알고리즘은 최선의 경우 (a)만 n-1번 수행되어 총 θ(n)가 되고, 최악의 경우 (b)가 총 n(n-1)/2번 순환하여 θ(n^2)의 수행시간을 필요로 하게 된다. 자신과 같은 크기의 요소와는 자리를 바꾸지 않으므로 stable하고, 배열 내부에서 삽입 및 정렬작업이 모두 이루어지므로 in-place이다.

③ Heap Sort

Heap sort는 (1) heapify와 (2)sorting, 이렇게 크게 두 단계로 이루어진다. (1) heapify는 leaf의 부모노드(인덱스 값 value.length/2)에서부터 시작해 전체 heap의 부모노드에 이르기까지 한 단계씩 percolate down작업을 수행함으로써 이루어진다. percolate down은 해당 노드와 자식노드의 크기관계를 heap property에 맞게 fix하는 역할을 하며, 부모-자식이

heap property를 잘 유지하고 있거나(종료 요건 1), heap의 leaf노드에 도달(종료 요건 2)할 때까지 재귀호출 된다. (2) sorting은 현재 부모노드(A)와 sorted되지 않은 부분 중 가장 끝에 있는 요소(B)를 서로 swap한 후, (A)를 sorted된 뒷부분에 포함시켜 관심의 대상으로부터 제외시키고, swap으로 인해 heap property가 깨졌을 경우를 처리하기 위해 새로운 부모노드 (B)부터 시작해 또다시 percolate down 작업을 진행한다. 배열의 총 요소 수를 n이라고 했을 때 (1) heapify 작업은 O(nlogn)의 수행시간(정확하게는 θ(n))을 필요로 한다. 이는 요소수가 n인 heap에서 그 어떤 subtree도 높이가 $\log_2 n$ 을 넘을 수 없어 한 단계씩 percolate down 해주는데 O(logn)이 걸리고, leaf의 부모노드로부터 heap 전체 부모노드까지 총 floor(n/2)번의 단계에서 이 과정을 반복하기 때문이다. (2) sorting은 총 n-1번의 for loop을 돌고, for loop 한번마다 percolate down이 호출되므로 총 O(nlogn)의 수행시간이 소요됨을 알 수 있다. 작성한 코드에서는 주어진 배열 내에서 원소간의 재배치가 이루어지므로 in-place이고, percolate down을 진행할 때 부모노드와 자식노드의 크기관계 때문에 동일한 값을 지닌 두 요소의 순서가 바뀔 수 있으므로 un-stable하다.

④ **Merge Sort**

merge_sort 메소드에서는 인풋 배열을 반을 나눠 재귀호출(base case-배열의 원소가 1개일 때)을 통해 각각에 대한 merge_sort를 수행(a)하고, 그리고 그 둘을 merge(b)하는 구조로 배열을 정렬한다. 실제로 정렬이 이루어지는 건 merge메소드에서인데, 인자로 받아온 두 배열(이 두 배열은 이미 정렬이 완료된 상태이다)을 왼쪽->오른쪽 순으로 크기를 비교하여 작은 요소를 새로 마련한 배열공간의 왼쪽에 차례로 위치시킨다. 이렇게 merge하면, 두 배열은 sorting 성질을 유지한 채 하나의 배열로 합쳐진다. (a) 분할 과정은 반씩 쪼개져 이루어지므로 그 깊이가 $\log_2 n$이며, 각 merge_sort마다 그 안에서 merge()메소드가 호출되므로 총 시간 복잡도는 O(nlogn)이다. (최악/최선/평균 모두 O(nlogn)). 병합한 값을 저장할 새로운 공간을 필요로 하므로 in-place가 아니고, 본 코드에서는 merge과정에서 왼쪽 배열의 값이 오른쪽 배열의 값보다 작거나 같을 경우 새로운 공간으로 해당 왼쪽배열 값을 이동시키므로, 같은 값일 때 순서가 바뀌지 않아 stable하다.

⑤ **Quick Sort**

Quick Sort에서는 먼저 partition메소드를 통해서 인풋 배열의 가장 오른쪽에 있는 값을 피벗으로 설정(이는 임의로 이루어진 작업이다), 그보다 작은 값은 왼쪽, 큰 값은 오른쪽에 재배치한다. (1-피벗보다 작은 수의 집합)(2-피벗보다 큰 수의 집합)(3-피벗) 이런 식으로 배치되면, 마지막에 3-피벗을 1-공간과 2-공간 사이에 넣어준다. 그리고 모든 재배치 작업이 마무리 된 후 피벗원소의 index가 반환되면, 그를 기준으로 피벗원소의 왼쪽부분과 오른쪽 부분에 대하여 각각 quick_sort 메소드 재귀호출을 통해 정렬을 수행해준다. partition으로 분할을 하는 데에는 모든 요소를 한 번씩 반드시 훑어야 하므로 시간복잡도는 θ(n)이 소요된다. partition이 어떻게 이루어 졌느냐에 따라(pivot값 선정과 연관) 성능이 다르게 나타나는데, 최악의 경우(한쪽으로 모두 쏠렸을 때-pivot이 최대나 최소였을 때) 재귀의 깊이가 n-1이 되므로 Quick Sort의 복잡도가 총 O(n^2)이 된다. 그러나 평균적으로는 반으로 나뉘어 재귀의 깊이가 $\log_2 n$이 되므로 O(nlogn)의 복잡도를 지닌다. 기존 배열에서 모든 재배치 작업이 이루어지므로 in-place이고, 본 코드에서는 partition에서 피벗보다 작거나 같을

때 왼쪽에 위치시키므로, stable하다(하지만 코드를 어떻게 짜느냐에 따라 unstable한 sorting을 구현하는 것 또한 가능하다.) -> [개선 가능] : 분할 후 재귀호출의 과정을 반복하며 배열크기가 어느 정도 작아졌을 때 크기가 작은 배열을 정렬하는 데에 있어 보다 효율적인 Insertion sort를 활용하게 되면 속도문제를 좀 더 개선할 수 있다. 또한, 분할을 위한 pivot의 선정이 효율성을 결정하므로, 맨 앞, 중간, 맨 뒤 원소 세 개 중 중앙값에 해당하는 원소를 pivot으로 설정하는 것도 성능개선에 도움을 준다.

⑥ **Radix Sort**

least significant digit에서부터 most significant digit까지 한 자리씩 훑으며 각 자리 수에 대한 크기별 재배치를 수행한다. stable한 정렬구조를 유지하는 것이 핵심이다. 인풋 배열 중 절댓값이 가장 큰 수(=>곧, 최대 자릿값을 가진 수)를 기준으로 1부터 10, 10^2, .. 이렇게 10의 거듭제곱씩으로 나눠 몫이 0이 되기 직전까지 각 자리수별로 for loop을 돈다. 매 for loop에는 해당 자릿수에 대한 sorting을 수행하기 위해 count배열이 선언되는데, 최소 –9에서 최대 9까지의 수가 존재할 수 있으므로 크기는 19이며, count[k]는 해당 자릿수가 k–9인 수의 개수 정보를 담게 된다(k=0~18). 다음 작업에서 count[i]값을 count[i+1]에 더해 count의 각 요소가 누적 개수 정보를 담도록 조정한 후, 각 요소별로 대응되는 index 값을 계산해 임시 공간 output에 인풋배열 요소들을 적절히 위치시켜준다. input array에 이 output값을 복사해 넣고, 모든 자리수를 훑을 때까지 이와 같은 과정을 반복한다. 비교연산이 없고, 각 digit에 해당하는 bucket(본 코드에선 count 배열 원소)에 인풋 배열 요소정보를 담는 구조이다. 따라서 절댓값이 가장 큰 수의 자릿값을 k, 총 배열의 원소의 개수를 n이라고 했을 때, 모든 배열의 요소를 한 번씩 다 훑는 작업을 k번 수행해 주어야 하므로 총 k*O(n)의 시간이 소요된다. 매번 각 해당하는 자릿수에 대하여 크기순으로 나열하기 위한 새 배열을 필요로 하므로 in-place가 아니며, 앞쪽에서부터 차례로 훑으며 본 요소가 어느 bucket에 해당하는지를 선택하므로 stable한 정렬방법이다.

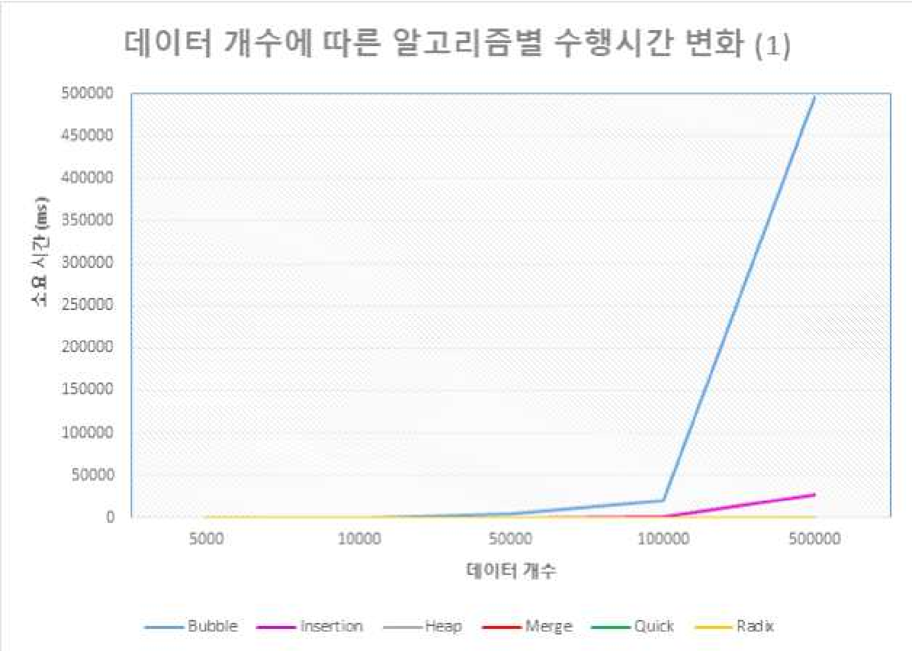
2. 동작 시간 분석 --

▶ 입력 데이터의 범위는 임의의 –100000이상, 100000이하의 정수로 설정하였다.

▶ 입력 데이터 개수는 수행시간 문제에 따라 Bubble Sort, Insertion Sort의 경우 5000, 10000, 50000, 100000, 500000개, 나머지 Heap, Merge, Quick, Radix Sort의 경우 그에 더해 1000000, 5000000, 10000000개에 대하여도 실험하였다.

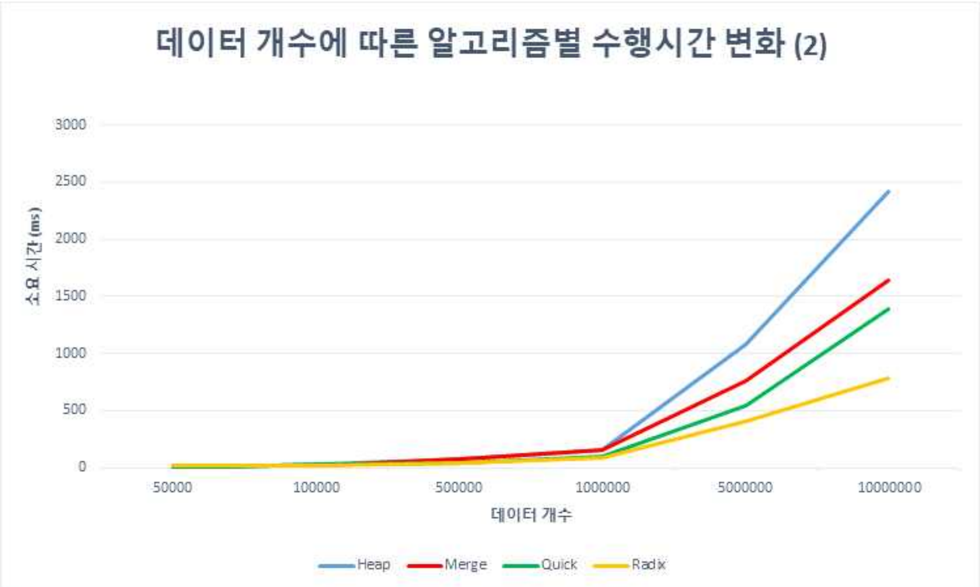
- ① *데이터 개수 : 5000 ~ 500000 일 때* (**Bubble Sort, Insert Sort** 중심)

데이터의 개수가 50000을 넘어가면 bubble sort와 insert sort는 지나치게 많은 시간이 소요되므로 (160000개 기준 약 3분소요) 5000, 10000, 50000, 100000, 500000개까지의 input만 넣는 방향으로 실험을 진행하였다(그러나 삽입정렬은 작은 배열에 한해 O(nlogn) 알고리즘들보다 더 빠른 경우도 있다). 첫 번째 실험결과 그래프를 보면, input data개수가 500000이하일 때 Bubble, Insertion Sort 이외의 정렬방법에서는 수행시간이 거의 0에 수렴하고, Bubble의 경우 Insertion Sort보다 훨씬 가파르게 시간복잡도가 증가하는 추세를 확인할 수 있다. 이는 Bubble Sort가 항상 n개의 data에 대해 O(n^2)의 수행시간을 필요로 하는데 반해, Insertion Sort는 O(n^2)알고리즘에 속함에도 인풋 배열이 정렬된 형태에 가까울수록 보다 효율적으로 작동하는 구조이기 때문에 나타난 결과라고 추론해 볼 수 있다.



- ② *데이터 개수 : 5000 ~ 10000000* (<u>**Heap, Merge, Quick, Radix Sort** 중심</u>)

두 번째 차트를 보면, 수행시간을 나타내는 그래프가 Heap > Merge > Quick > Radix 순으로 가팔라지는 것을 볼 수 있는데, 이는 앞 세 알고리즘이 O(nlogn), Radix sort가 c*O(n)(c는 상수)의 시간복잡도(비교연산 수행X)를 지니기 때문인 것으로 해석된다. (Radix Sort는 최악의 경우와 평균의 경우 모두 O(n)으로, 매번 O(nlogn) 정렬 알고리즘들보다 유의미하게 완만한 그래프 모양을 보이는 것을 확인할 수 있다.)



- ③ 그 외 유의점 : 하나 유의해야 할 점은, 다른 알고리즘들과는 달리 Quick Sort의 경우 input 데이터의 숫자 범위가 늘어날수록 영향을 받아 수행시간이 규칙적으로 감소했다는 것이다. 이는 퀵소트에서 숫자 범위가 적을 때 자신과 같은 원소를 처리해야 하는 문제 등 partition의 비효율이 일어날 가능성이 더 커지기 때문에 나타나는 것으로 생각된다.