HW4 Report

산업공학과 2018-19833 길영환

1. 정렬 알고리즘의 동작 방식

편의상 배열의 길이는 n이라 하자.

가. Bubble sort

배열을 앞에서부터 하나씩 읽어가며 자신과 자신보다 하나 뒤의 원소의 크기를 비교한다. 만약 자신보다 뒤의 원소가 더 작다면 두 원소를 교환하여 더 큰 값이 뒤로 가게 해준다. 이 과정을 반복하면 가장 큰 원소가 맨 끝에 위치한다. 그러면 마지막 원소를 제외한 나머지 원소들은 정렬이 안 되어있는 상태이다. 따라서 마지막 원소를 제외하고 길이가 1 작아진 문제를 풀면 정렬을 완료할 수 있다. bubble sort는 이중 for 문을 이용하여 정렬하고자 하는 범위를 1씩 줄여가며 위의 과정을 반복한다. 본 코드에서 len 이라는 변수는 for 문에서 A의 정렬 범위를 의미한다. Bubble sort의 시간 복잡도는 항상 n^2에 비례한다.

나. Insertion sort

Insertion sort에서는 i번째 원소 삽입 시, A[0]~[i-1]까지의 원소들은 정렬이 되어있는 상태이다. 이제 A[i]를 적절한 위치에 삽입해주면 된다. 앞의 값들은 이미 정렬이 되어있으므로, i-1 번째 원소부터 하나씩 앞으로 가면서 자신보다 큰 원소들은 한 칸씩 뒤로 밀어준다. 자신보다 같거나 작은 원소가 나타나는 순간, 이중 for 문 구조에서 내부 for 문이 중단되며, 발견한 자신보다 같거나 작은 원소 하나 뒤에 자신의 값을 위치시킨다. Insertion sort는 모든 원소가 같거나 정렬이 이미 끝난 상황이라면, n에 비례하는 시간 복잡도를 가진다. Insertion sort의 시간 복잡도는 평균적으로 n^2, 최악의 경우에도 n^2에 비례한다.

다. Heap sort

Heap sort는 아래의 두 가지 단계로 이루어진다.

1. 배열을 max 힙으로 만들어준다.

2. 힙의 최대원소를 삭제하고 삭제한 원소는 배열의 맨 뒤에 위치시킨다.

max 힙을 만들 때는 branch 함수를 이용하여 진행하는데 자식 노드가 존재하는 n/2개의 노드에 대해서만 힙 성질을 만족하도록 변환해주면 되기 때문에 총 n/2번 진행한다.

힙의 맨 위의 값이 배열의 최댓값이므로 이는 배열의 첫 번째 원소이므로 이를 배열의 마지막 원소와 맞바꿨다. 그러면 힙 구조가 깨지게 되는데, 배열의 0번째 원소와 마지막 원소를 제외하면 힙 구조를 만족하기 때문에 배열의 범위를 1 줄여서 생각하고, branch 함수를 이용하여 0번째 노드를 적절하게 배치하여 다시 힙으로 만들어주었다. 다시 힙 구조를 만족하므로 최대원소를 뒤로 보내는 위의 과정을 반복하면 정렬을 완료할 수 있다.

branch 함수는 특정 노드와 배열, 배열의 범위가 주어지고, 특정 노드 아래쪽은 힙 성질을 만족할 때, 힙 구조를 만족하도록 특정 노드의 값을 위치시켜주는 역할을 한다. 본 코드에서는 while 문을 이용하여 구현하였는데, 만약 자식 노드 중 큰 값(maxchild)이 부모 노드의 값보다 크다면 max 힙 성질을 만족하지 않으므로 maxchild와 부모의 노드 값을 교환한다. 만약 maxchild가 부모 노드 값보다 작거나 같다면 반복문을 즉시 중단한다. 이후, 반복문이 중단될 때까지 바뀐 위치에서도 위의 과정을 반복한다.

이때, branch 함수는 $\log n$에 비례하는 시간이 소요되고, 처음 배열을 힙으로 만들어줄 때,

branch함수를 n/2번 호출하였고, 배열의 원소를 삭제하는 과정에서는 삭제를 n-1번 실행했으므로 n-1번 호출하였다. 따라서 Heap sort는 항상 $n\log n$에 비례하는 시간 복잡도를 가진다.

라. Merge sort

우선 배열을 반으로 나누고 앞부분의 배열과 뒷부분의 배열에서 각각 재귀적으로 merge sort를 호출한 뒤, 정렬된 두 배열을 merge 함수를 이용하여 합친다. 정렬된 두 배열을 합치는 과정에서는 n에 비례하는 만큼의 비교가 이루어진다. 본 알고리즘의 특성상 배열을 반으로 나눈 뒤, 크기가 반이 된 문제를 다시 풀게 되므로 recursion의 깊이는 $\log n$이다. 따라서 merge sort의 시간복잡도는 $nlogn$에 비례한다. merge sort는 in-Place sorting이 아니라는 단점이 있다. 일반적인 Merge sort는 정렬된 두 주 배열을 병합하여 보조 배열에 저장했다가 다시 주 배열에 되써준다. 이러한 과정을 개선하기 위해 본 코드에서는 주 배열과 보조 배열의 역할을 번갈아 가며 수행하도록 바꿔주었다. 이를 구현하기 위해서 본 코드에서는 우선 보조 배열(nvalue)에 주 배열의 값을 복사하였고, msort 함수에서 merge 함수와 msort를 재귀적으로 호출할 때, 주 배열과 보조 배열의 위치를 바꾸어서 넣어주었다.

마. Quick sort

pivot 원소를 지정하고 배열의 원소 중 pivot 원소보다 큰 원소들을 pivot의 뒤에 위치시킨다. 그러면 자연스럽게 pivot 원소보다 같거나 작은 원소들은 pivot의 앞에 위치하게 되고, pivot보다 큰 원소들은 pivot의 뒤에 위치하게 된다. 이후, pivot을 기준으로 나눠진 두 개의 partition에 대하여 재귀적으로 quick sort를 호출하여 정렬을 진행한다. 본 코드에서는 배열의 맨 마지막 원소를 pivot으로 지정하였다. quick sort의 경우, partition이 반으로 쪼개진다는 보장이 없으므로 recursion의 깊이가 $\log n$이 아닐 수 있다. 예를 들어, 배열의 최대원소나 최소 원소가 pivot으로 선정될 경우, partition이 2개로 쪼개지지 않게 된다. 만약, 계속해서 배열의 최소, 혹은 최대원소가 pivot으로 선정될 경우, recursion의 깊이가 n이 되고, 시간 복잡도는 n^2이 된다. Quick sort는 최선의 경우와 평균적인 경우의 시간 복잡도는 $n\log n$에 비례한다.

바. Radix sort

우선, 배열에서 가장 자릿수가 큰 원소의 자릿수를 저장한다. 이후, 각 원소의 일의 자리의 수부터 시작하여 가장 큰 자리의 수까지 각 자리의 수별로 원소를 독립적으로 정렬한다. 이때, 각 정렬은 stable sort를 만족하도록 진행한다. 각 자리의 수별로 정렬할 때는 범위가 -9부터 9까지인 수 n개를 정렬하는 것이므로 Counting sort의 알고리즘을 이용하였다. Radix sort의 시간 복잡도는 n에 비례한다.

본 코드에서는 배열의 원소 최대원소와 최소 원소를 찾아 각각의 자릿수 값 계산하고 더 큰 값을 digits 변수에 저장하였다. (가장 작은 원소가 음수일 수 있고, 음의 n 자릿수 정수의 자릿수 값을 n으로 간주한다) 이후, 각 자리의 수별로 독립적으로 stable sort를 진행하는데 value 배열을 해당 자리의 수를 기준으로 정렬한 결과를 sorvalue에 저장하였고, 다음 반복문을 수행하기 위해 sorvalue와 value의 주솟값을 맞바꿔주었다. counting sort 알고리즘을 적용할 때는 각자리 수의 범위가 -9~9이므로 freq 배열의 [해당 숫자+9] 번째 자리에 해당 숫자가 나타난 횟수를 저장하고, 이후 저장된 값을 누적하여 각 원소의 위치를 나타내도록 하였다.

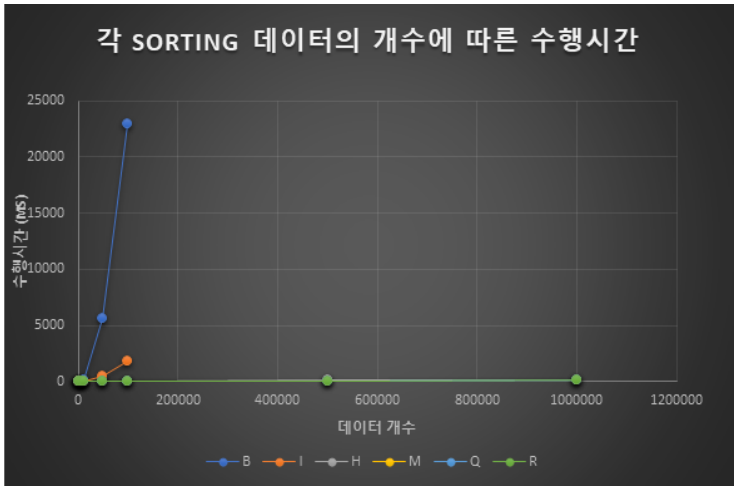
2. 동작 시간 분석

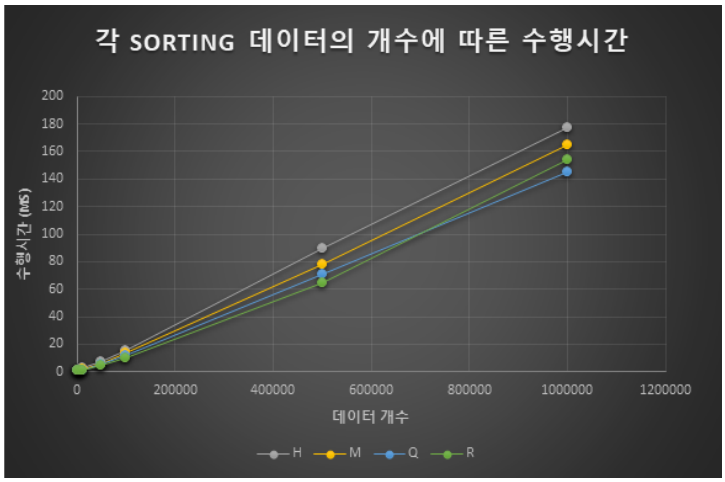
가. 각 정렬의 이론적인 시간 복잡도

	Worst case	Average case	Best case
Bubble sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$
Insertion sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n)$
Heap sort	$\theta(nlogn)$	$\theta(nlogn)$	$\theta(nlogn)$
Merge sort	$\theta(nlogn)$	$\theta(nlogn)$	$\theta(nlogn)$
Quick sort	$\theta(n^2)$	$\theta(nlogn)$	$\theta(nlogn)$
Radix sort	$\theta(n)$	$\theta(n)$	$\theta(n)$

나. 실험결과

$[-10^6, 10^6]$범위의 숫자 1000개, 5000개, 10000개, 50000개, 100000개, 500000개, 1000000개의 데이터에 대해 각 정렬의 수행시간을 측정한 결과는 다음과 같다. 버블 정렬과 삽입정렬의 경우 데이터의 개수가 500000 이상인 경우에 대해서는 수행시간이 너무 오래 걸리는 관계로 100000개 까지만 실험을 진행하였다. 데이터 추출 시에는 세 번째 이후 수행하여 추출한 데이터 10개의 평균값으로 비교하였다.

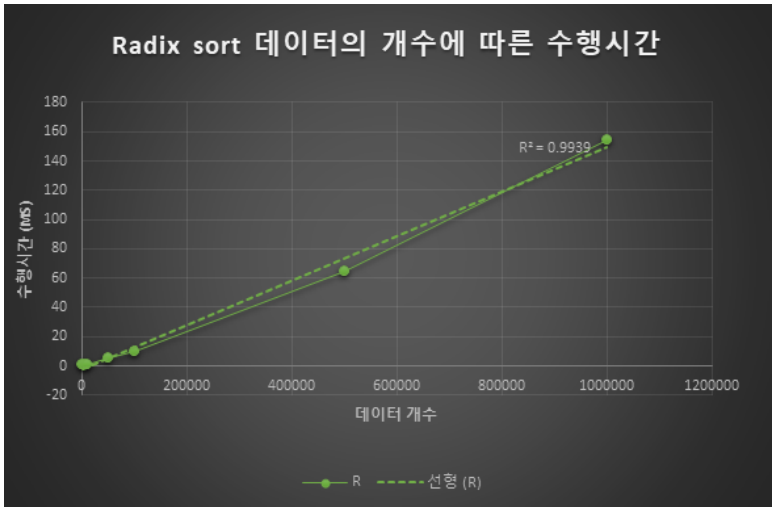




속도의 경우, 전체적으로 Radix≒Quick>Merge>Heap>Insertion>Bubble 순으로 빨랐다. 본 실험의 결과를 살펴보면, 데이터의 개수가 1000개 이하일 때는 수행시간이 너무 짧기 때문에 정렬 간 수행시간이 크게 차이나지 않았다. 그러나 데이터가 5000개만 돼도 Bubble sort와 Insertion sort의 수행시간이 Heap sort, Merge sort, Quick sort, Radix sort에 비해 상당히 크게 나타났다.(Bubble sort의 경우 다른 고급정렬대비 24배 이상, Insertion sort의 경우 5배 이상) 이러한 정렬 수행시간의 차이는 데이터의 개수가 커질수록 더욱 크게 나타났다.

일반적으로 Radix sort는 n에 비례하는 시간 복잡도를 가지고, Quick sort는 nlogn에 비례하는 시간 복잡도를 가지기 때문에 Radix sort가 더 빠른 수행속도를 기록할 것으로 예상하였으나, in-Place sorting이 아닌 점, 각 자리 수별 stable sorting을 처리하는 과정에서 메모리 발생에 의한 상수 factor가 더 큰 영향을 미쳐, Quick sort보다 수행속도가 느리게 나타난 것으로 생각된다. Quick sort는 이러한 상수 factor에 의한 영향이 상대적으로 적어, 다른 $nlogn$에 비례하는 고급정렬보다도 좀 더 수행시간이 빨랐던 것으로 생각된다.

다음은 Radix sort 시간 복잡도가 n에 비례한 이론적 결과를 확인하기 위해 Radix sort 결과 그래프를 선형 추세선으로 추정한 결과이다. R^2값이 0.9939로 1에 상당히 가까운 것을 확인할 수 있다.



다. 추가 논의

Quick sort는 동일한 원소가 많을 경우, partition 분할시, 균일하게 2등분 되지 않을 확률이 높아진다. 이에 대한 검증을 위해 [-10, 10]의 범위에서 10000회 추출하여 각 sorting 방법별로 측정한 수행시간 결과는 다음과 같다.

	Bubble sort	Insertion sort	Heap sort	Merge sort	Quick sort	Radix sort
평균 수행시간 (ms)	135	36	3	2	13	2

위의 표에서 확인 할 수 있듯이 [-1000000, 1000000]범위에서 추출한 경우와 달리, Quick sort의 수행시간이 Heap sort, Merge sort, Radix sort에 비해 현저하게 큰 것을 확인할 수 있다.