자료구조 과제 4 보고서

공과대학 컴퓨터공학부
2021-16988 박재완

1. 정렬 알고리즘의 동작 방식

1.1. Bubble Sort (버블 정렬)

버블 정렬은 가장 큰 원소를 마지막으로 이동시켜주는 과정을, '마지막'의 위치를 하나씩 줄여가면서 배열의 길이만큼 반복하여 작동한다. 가장 큰 원소를 뒤로 이동시키기 위해서는 앞에서부터 인접한 두 원소씩 비교하며 큰 것을 뒤로 옮기도록 한다. 따라서 이렇게 옮기는 과정, 그리고 '마지막'의 위치를 바꾸는 과정을 이중 반복문으로 구현하였다. 이때 배열 안에서 원소의 위치를 바꾸는 작업은 swap()이라는 함수를 따로 정의해두고 여러 정렬 알고리즘에서 반복하여 사용하였다.

1.2. Insertion Sort (삽입 정렬)

삽입 정렬은 배열의 i 번째 인덱스에 위치한 원소를, '정렬이 된' 0 번 인덱스부터 (i-1)번 인덱스까지 하위 배열의 적합한 위치에 끼워넣는 과정으로 진행된다. 이 과정은 1 번 인덱스부터 시작되어, 앞에서부터 차례로 정렬을 해나가며 전체 배열이 정렬되도록 한다. 각 원소를 적합한 위치로 옮기는 과정을 배열의 길이만큼 반복하게 되어 버블 정렬과 마찬가지로 이중 반복문으로 구현하였다. 또한 위의 swap() 함수를 역시 이용하였다.

1.3. Heap Sort (힙 정렬)

힙 정렬은 완전 이진 트리의 일종인 '힙'을 이용한 정렬이다. 배열의 원소를 힙으로 만들어주면 힙의 성질을 이용하여 정렬이 가능하다. 힙은 트리 구조에서 특정 노드의 하위에는 언제나 두 개의 자식 노드가 있어야 하며, 가장 낮은 순위의 노드만 형제 노드가 없을 수 있다. 또한 언제나 부모 노드의 값이 자식 노드의 값보다 커야 한다.

따라서 이러한 조건을 국소적으로 만족시켜주기 위하여, 특정 노드에 대하여 모든 하위 노드 중 적합한 위치로 이동시켜주는 작업이 필요하다. 이를 percolateDown() 함수에서 구현하였다. 해당 노드가 자식 노드보다 값이 작을 경우 값을 바꾼 후, 다시 그 자식 노드부터 재귀적으로 percolateDown()이 수행되도록 하였다. 이를 가장 하위 노드까지 탐색하도록 하였다. 이후 이렇게 구현된 percolateDown() 함수를 가장 마지막 노드의 부모 노드부터 최상위 노드까지 순차적으로 실행하는 buildHeap() 함수를 정의하였다. 이는 임의의 정렬되지 않은 배열을 힙 특성에 맞게 정렬되도록 해주었다.

힙 특성에 맞게 정렬되면, 가장 최상위 노드는 언제나 배열의 최댓값이 된다. 따라서 최상위 노드를 가져오고, 나머지 노드들을 다시 힙 특성에 맞게 정렬하는 과정을 반복하면 가장 큰 원소부터 배열이 정렬되도록 할 수 있다. 이 과정은 deleteMax() 함수에서 구현하였는데, 최하위 노드와 최상위 노드를 교체하고, 최상위노드부터 최하위 노드보다 하나 이전의 노드까지 percolateDown()을 진행해주는 작업이었다. 이를 배열의 길이만큼 반복하여 순차적으로 정렬이 되도록 하였다. 이 경우 자연스럽게 배열의 최댓값이 최하위 노드부터 위치하게 되며 크기의 역순으로 정렬되어 전체적으로는 올바른 순서로 정렬된다.

1.4. Merge Sort (병합 정렬)

병합 정렬은 배열의 크기를 이등분하여 각각을 정렬한 다음 이를 '병합'하는 과정을 통해 정렬이 이루어진다. 이때 이등분한 배열의 정렬 역시 병합 정렬으로 진행하여, 재귀적으로 과정이 진행된다. 또한 배열을 나누는 기준은 크기의 절반 지점으로 하였다.

정렬된 두 배열을 병합하는 과정은 merge() 함수에서 정의해주었다. 앞에서부터 순서대로 비교해가며 전체 배열이 정렬되도록 하였다.

1.5. Quick Sort (퀵 정렬)

퀵 정렬은 배열의 원소 중 기준을 정하여 기준보다 작은 원소, 큰 원소를 기준의 왼쪽, 오른쪽에 오도록 재배치한 뒤 각 부분을 정렬하는 방법이다. 이때 각 부분 역시 퀵 정렬로 정렬하여 재귀적으로 과정이 진행된다. 배열을 나누는 기준 원소는 다양할 수 있지만, 임의로 가장 오른쪽 원소로 하였다. Random 한 원소가 입력될 경우 기준점의 위치는 중요하진 않지만, 편향된 입력이 올 경우 기준점의 위치에 따라 속도가 달라지기도 한다. 2.3 절에서 더욱 이야기해보고자 한다.

가장 첫 단계인 재배치는 partition() 함수에서 진행해주었다. 기준 원소의 '왼쪽'과 '오른쪽'의 경계가 될 위치를 변수로 잡고, 순차적으로 모든 원소를 돌면서 기준보다 작은 원소가 있을 경우 해당 위치로 옮겨준 뒤, 위치 변수를 1 씩 증가시켜주었다. 마지막에는 기준 원소를 경계 위치로 옮겨주어 기준원소보다 작은 원소들과 큰 원소들로 배열이 나누어지도록 하였다. 이후에는 각 부분을 재귀적으로 퀵 정렬하여 전체 배열이 정렬되도록 하였다.

1.6. Radix Sort (기수 정렬)

기수 정렬은 자연수를 일의 자리수부터 '안정성을 유지하며' 높은 자리수까지 차례로 정렬하는 방법이다. 일의 자리수를 기준으로 정렬한 뒤, 십의 자리, 백의 자리 등 배열에 포함된 가장 높은 자리수까지 정렬하여 결과적으로 전체적인 정렬이 되도록 한다. 이때 '안정성을 유지'함은 특정 자리수에서의 정렬에서 두 수가 같은 값을 가지면, 기존에 정렬되었던 순서가 유지됨을 말한다.

배열에 포함된 수 중 가장 높은 자리수를 구하는 과정을 maxIndex()에서 진행하였다. 각 수를 문자열로 변환한 뒤, 문자열의 길이 중 가장 긴 것을 최대 자리수로 잡았다.

각 자리수별로 정렬을 진행하는 것은 indexSort()에서 진행하였다. 우선 각 수를 탐색하며 해당 자리수에 0 부터 9 가 총 몇번 등장하였는지를 count 배열 (크기 10)에 담았다. 이를 통해 정렬이 될 때, 해당 자리수의 수가 0 인 수가 몇번부터 몇번 인덱스에 들어가야하는지, 1 인 수는 몇번부터 몇번 인덱스에 들어가야하는지 등을 차례로 계산할 수 있었다. 이를 기준으로, 각 수를 해당 자리수를 기준으로 정렬할 수 있었다. 이를 일의 자리수부터 maxIndex()에서 구한 최대 자리수까지 진행하면 전체 정렬이 완료되었다.

다만 이는 자연수만 입력되는 경우에 해당되고, 음수를 포함한 범위에서 입력될 경우 음수와 양수를 나누어 정렬한 뒤 합쳐주어야 한다. 이를 고려하여 코드를 작성하였다.

2. 알고리즘별 동작 시간 분석

2.1. 전체적인 비교

-1000 부터 1000 범위의 랜덤한 입력을 100 개, 500 개, 1000 개, 5000 개, 10000 개, 50000 개, 100000 개 입력할 때, 각 알고리즘별로 정렬하는데 얼마만큼의 시간이 걸렸는지를 100 회 측정하여 평균을 구해보았다.

전반적으로 힙 정렬, 병합 정렬, 퀵 정렬이 가장 빠른 편에 속했고, 버블 정렬과 삽입 정렬이 가장 느린 편에 속했다. 결과는 다음 표에 기록하였다.

입력 개수	버블 정렬	삽입 정렬	힙 정렬	병합 정렬	퀵 정렬	기수 정렬
100	1ms	1ms	1ms	0ms	1ms	1ms
500	11ms	4ms	2ms	1ms	1ms	6ms
1000	15ms	9ms	3ms	2ms	3ms	8ms
5000	45ms	23ms	7ms	6ms	6ms	22ms
10000	102ms	73ms	9ms	7ms	10ms	36ms
50000	3249ms	920ms	24ms	26ms	24ms	59ms
100000	13724ms	3598ms	37ms	40ms	34ms	98ms

표 1 -1000 이상 1000 이하의 정수를 정렬할 때, 입력 개수 및 알고리즘별 실행 시간 (100 회 시행 평균)

힙, 병합, 퀵 정렬은 비슷한 속도를 가지며 모두 빠른 정렬이 가능했는데, 자세하게 비교해보면 병합 정렬이 힙 정렬보다 입력이 적은 경우에서는 빨랐고, 입력이 많은 경우에서는 느렸다. 반면 퀵 정렬은 병합정렬과 비교한 상대적인 빠르기가 변하는 것을 볼 수 있었다. 이는 퀵 정렬이 입력의 수와는 관련이 크게 없고 입력된 수의 분포의 영향을 받음을 설명해준다. 버블 정렬과 삽입 정렬은 모두 매우 느렸으나, 버블 정렬이 삽입정렬보다 더 느린 것을 확인해볼 수 있었다. 기수 정렬은 중간 정도의 빠르기를 보였다.

실제로 각 알고리즘의 평균 시간 복잡도를 구해보면, 순서대로 $\Theta(n^2)$, $\Theta(n^2)$, $\Theta(n\log n)$, $\Theta(n\log n)$, $\Theta(n\log n)$, $\Theta(n)$이다. 힙, 병합, 퀵 정렬의 속도가 빠르고 버블, 삽입 정렬의 속도가 느림을 설명해준다. 또한 각각의 best case 시간 복잡도는 순서대로 순서대로 $\Theta(n^2)$, $\Theta(n)$, $\Theta(n)$, $\Theta(n\log n)$, $\Theta(n\log n)$, $\Theta(n)$인데, 이는 힙 정렬이 입력이 많아질 때 병합 정렬보다 빨라지고 버블 정렬이 삽입 정렬보다 전반적으로 빠른 것을 설명해준다. 각각의 worst case 시간 복잡도는 순서대로 $\Theta(n^2)$, $\Theta(n^2)$, $\Theta(n\log n)$, $\Theta(n\log n)$, $\Theta(n^2)$, $\Theta(n)$이다. 이는 퀵 정렬이 병합 정렬이나 힙 정렬보다 느려지는 경우가 생김을 설명해준다.

다만 기수 정렬에서 예외가 발생하였는데, 이는 본 과제에서 기수 정렬은 음수 범위까지 포함하였기 때문이다. 때문에 overhead 가 증가하여 힙 정렬, 병합 정렬, 퀵 정렬보다는 확실히 느려지는 것을 확인할 수 있었다.

2.2. 병합 정렬의 성능 향상

병합 정렬에서 merge() 함수를 단순히 모든 원소를 비교해가며 합치는 것으로 구현할 경우, 시간이 과도하게 오래걸리는 것을 확인할 수 있었다. 특히 100000 개 정도의 수를 입력하는 몇몇 상황에서는 유의미한 시간 내에 정렬이 완료되지 않아 결과를 확인할 수가 없었다. 따라서 이를 해결하기 위하여, 병합하고자 하는 두 배열 중 앞 배열의 최댓값이 뒤 배열의 최솟값보다 작은 경우, 따로 병합을 하지 않도록 처리를 해주었다. 이 경우 평균적인 상황에서 실행 시간이 줄어드는 것을 확인할 수 있었다. 2.1 절의 결과도 모두 이 처리를 포함한 결과이다.

-1000 이상 1000 이하의 수 100000 개를 입력하는 경우, 해당 처리를 해주지 않은 경우는 100 번 시행에서 평균 48ms 이 걸렸고, 처리를 해준 경우는 100 번 시행에서 평균 36ms 이 걸렸다. 실행 시간이 단축됨을 확인할 수 있다.

2.3. 퀵 정렬의 성능 향상

퀵 정렬은 기준 원소를 정하고 그보다 작은 원소, 큰 원소를 구분하여 정렬을 진행하기 때문에 기준 원소가 전체 배열을 정렬하였을 때 어느 정도 위치에 있는가에 따라 빠르기가 달라진다. 나누었을 때 각 부분의 크기가 비슷한 경우는 병합 정렬보다도 빠른 결과를 얻을 수 있지만, 한 부분에 많은 원소가 몰려있는 경우 느려질 수 있으며 이 경우가 2.2 절에서 서술된 worst case 이다. 시간 복잡도는 $\Theta(n^2)$로 증가한다.

이를 실제로 확인해보았는데, 고르게 분포된 경우와 그렇지 않은 경우를 비교해보았다. -1000 부터 1000 사이의 수 10000 개를 입력하는 경우와 1 과 3 사이의 수 10000 개를 입력하는 경우, 100 번 시행의 평균으로 전자는 22ms 가 결린 반면 후자는 45ms 가 걸렸다. 따라서 퀵 정렬은 평균적으로 빠르지만, 극단적인 분포의 입력에 대하여는 속도가 느려질 수도 있음을 알 수 있었다.