정렬 알고리즘

컴퓨터공학과 2학년 20191125 박형민

* 정의
* 정렬 알고리즘이란 원소들을 번호순이나 사전 순서와 같이 일정한 순서대로 열거하는 알고리즘이다. 효율적인 정렬은 탐색이나 병합 알고리즘처럼 (정렬된 리스트에서 바르게 동작하는) 다른 알고리즘을 최적화하는데 중요하다. 또 정렬 알고리즘은 데이터의 정규화나 의미 있는 결과물을 생성하는데 흔히 유용하게 쓰인다. 이 알고리즘의 결과는 반드시 다음 두 조건을 만족해야 한다.

1. 출력은 비 내림차순(각각의 원소가 전 순서 원소에 비해 이전의 원소보다 작지 않는 순서)이다.
2. 출력은 입력을 재배열하여 만든 순열이다.

* 분류
* 정렬 알고리즘은 다음과 같은 기준으로 분류된다:
* 원소들의 크기 비교에 따라 계산 복잡도 (최선, 최악, 평균 동작). 직렬 정렬 알고리즘의 경우 최선 동작은 O(*n* log *n*), 최선 동작 중 병렬 정렬은 O(log2 *n*), 최악 동작은 O(*n*2)이다. (점근 표기법 참고.) 직렬 정렬의 이상적인 동작은 O(*n*)이지만 평균 케이스에는 가능치 않다. 최적의 병렬 정렬은 O(log *n*)이다. 비교 기반 정렬 알고리즘은 대부분의 입력에 대해 최소 Ω(*n* log *n*)개의 비교가 필요하다.
* 원소들의 교환 횟수에 따른 계산 복잡도.
* 메모리 사용량 (및 다른 컴퓨터 자원의 사용량). 특히 일부 정렬 알고리즘들은 제자리(in-place, 인플레이스)이다. 정확히 말해 제자리 정렬은 정렬되는 항목 외에 O(1)개의 메모리만 필요하며 O(log(*n*))개의 추가적인 메모리를 인플레이스로 간주한다.
* 재귀. 일부 알고리즘들은 재귀성을 띄거나 재귀성을 띄지 않을 수 있으며 다른 알고리즘들은 그 둘의 특성을 지닐 수 있다. (예: 머지 소트)
* 안정성: 안정적인 정렬 알고리즘들은 동일 키(예: 값)의 레코드의 상대적인 순서를 관리한다.
* 비교 정렬인지 아닌지의 여부. 비교 정렬은 비교 연산자를 통해 2개의 요소만을 비교함으로써 데이터를 검사한다.
* 일반적인 방식: 삽입, 교환, 선택, 병합 등. 교환 정렬에는 거품 정렬과 퀵 소트가 있다. 선택 정렬에는 셰이커 소트와 힙 소트가 있다.
* 알고리즘이 직렬인지 정렬인지의 여부. 이 토론의 나머지 부분은 거의 예외적으로 직렬 알고리즘에 집중하며 직렬 조작을 다룬다.
* 순응성: 입력을 미리 정리하는 일이 실행 시간에 영향을 미치는지에 여부. 이를 고려사항에 넣는 알고리즘들은 순응적이다.
* 종류
* 정렬 알고리즘은 그 특징에 따라 몇 가지로 분류할 수 있다.
* 버블 정렬
* 1번째와 2번째 원소를 비교하여 정렬하고, 2번째와 3번째, ..., n-1번째와 n번째를 정렬한 뒤 다시 처음으로 돌아가 이번에는 n-2번째와 n-1번째까지, ...해서 최대  ​번 정렬한다. 한 번 돌 때마다 마지막 하나가 정렬되므로 원소들이 거품이 올라오는 것처럼 보여 버블 정렬이다. 버블 정렬은 이미 정렬된 자료에서는 1번만 돌면 되기 때문에 최선의 성능을 보여준다. 가장 손쉽게 구현하여 사용할 수 있지만, 만들기가 쉽고 직관적일 뿐이지 알고리즘적 관점에서 보면 비효율적인 정렬 방식이다.
* 선택 정렬
* 버블 정렬이 비교하고 바로 바꿔 넣는 걸 반복한다면 이쪽은 일단 1번째부터 끝까지 훑어서 가장 작은 게 1번째, 2번째부터 끝까지 훑어서 가장 작은 게 2번째……해서 (n-1)번 반복한다. 어찌 보면 인간이 사용하는 정렬 방식을 가장 많이 닮았다. 어떻게 정렬이 되어 있든 일관성 있게  에 비례하는 시간이 걸린다는 게 특징. 또한, 버블 정렬보다 두 배 정도 빠르다.
* 삽입 정렬
* k번째 원소를 1부터 k-1까지와 비교해 적절한 위치에 끼워 넣고 그 뒤의 자료를 한 칸 씩 뒤로 밀어내는 방식으로, 평균적으론중 빠른 편이나 자료구조에 따라 선 뒤로 밀어내는데 걸리는 시간이 크다. 다만 이미 정렬되어 있는 자료구조에 자료를 하나씩 삽입/제거하는 경우에는 현실적으로 최고의 정렬 알고리즘이 되는데, 탐색을 제외한 오버헤드가 매우 적기 때문이다.
* 퀵 정렬
* 적절한 원소 하나를 기준(피벗, pivot)으로 삼아 그보다 작은 것을 앞으로 빼내고 그 뒤에 피벗을 옮겨 피벗보다 작은 것, 큰 것으로 나눈 뒤 나누어진 각각에서 다시 피벗을 잡고 정렬해서 각각의 크기가 0이나 1이 될 때까지 정렬한다. 이렇게 피벗을 잡고 이보다 작은 원소들을 왼쪽으로, 보다 큰 원소들을 오른쪽으로 나누는 걸 partition step이라 한다. 퀵 정렬에도 이 partition step을 어떻게 하느냐 에 따라 바리에이션이 매우 많으며, 성능 차이도 날 수 있다.
* 트리 정렬

1. 정렬될 배열의 맨 첫 값이 루트 노드가 된다.
2. 다음 값부터는 기존 노드 값과 비교한다. 루트 노드에서 출발해서 추가될 노드 값이 기존 노드 값보다 작은 경우는 왼쪽 자식을, 기존 노드 값보다 크거나 같을 경우는 오른쪽 자식을 찾는다. 내림차순은 반대로 기존 노드 값보다 크면 왼쪽, 작거나 같으면 오른쪽을 찾으면 된다.
3. 위 2에서 해당 방향의 자식 노드가 없으면 그 방향의 자식 노드로 추가한다. 있으면 그 방향의 자식 노드로 가서 크기를 비교하고 위 2와 같은 방법으로 해당 방향의 자식 노드가 있으면 계속 그 방향으로 가서 조사하고 없으면 그 방향의 자식 노드로 추가한다.
4. 모든 값이 노드로 추가되었으면 해당 트리를 중위 순회 방식으로 순회하여 그 순서대로 값을 정렬해 나간다.

* 기수 정렬
* 데이터가 x진법이라고 가정하자. 0번부터 x-1번까지의 리스트를 만들어 놓고, 각 데이터를 순서대로 현재 자릿수의 숫자가 가리키는 리스트에 밀어넣고, 리스트를 0번부터 x-1번까지 순서대로 이어붙인다. 이 과정을 가장 낮은 자릿수부터 가장 높은 자릿수까지 반복하면 정렬이 끝나게 된다.
* 위에 나온 알고리즘은 모두 데이터들의 직접적인 비교를 이용하는데, 이렇게 데이터끼리 직접적인 비교를 하여 정렬할 경우 시간 복잡도는 보다 작아질 수 없다. 이 기수 정렬은 자릿수가 있는 데이터(정수, 문자열 등)에서만 수행이 가능하며, **데이터들의 직접적인 비교 없이** 정렬을 수행한다. 비교를 이용한 정렬이 아니기 때문에 k가 상수일 경우 시간 복잡도가 O(*n*)으로 **퀵 정렬보다 빠른 시간 복잡**도가 나오는 것이 가능하다. 어떻게 데이터끼리 비교하지도 않고 정렬을 할 수 있는지는 후술. 다만 이 알고리즘은 자릿수가 적은 4바이트 정수 등에서나 제대로 된 성능을 발휘할 수 있으며, 자릿수가 무제한에 가까운 문자열 정렬 등에 사용할 경우 오히려 퀵 정렬보다 느릴 수 있고, 부동 소수점의 경우는 부호여부, 지수부, 가수부에 대해 각각 기수 정렬을 실행해야 한다.
* 카운팅 정렬
* O(*n*+*k*) (k는 데이터의 **최댓값**을 의미한다.)
* 카운팅 정렬은 가장 큰 데이터에 따라 효율이 좌지우지된다. 쉽게 설명하자면 특정 데이터의 개수(1이 두 개 있다면 2)를 데이터의 값에 대응하는 위치에 저장한 뒤, 자신의 위치에서 앞에 있던 값을 모두 더한 배열을 만든 뒤, 거기서 데이터가 들어가야 할 위치를 찾아내는 정렬 알고리즘이다. 이 경우에 데이터의 최댓값을 k라 두면, 시간 복잡도는 O(*n*+*k*). 하지만, 만약 k가 억 단위를 넘어간다면?  n이 아무리 작아도 동작시간이 크다. 그럴 땐 위의 정렬들을 사용하는 게 바람직하다. 반대로 k가 매우 작다면, 오히려 선형시간의 효과를 볼 수 있다. 즉, k가 작다는 조건이라면 매우 효율적인 정렬. 또한 카운팅 정렬은 배열을 사용하는 특성상, 정수라는 전제를 깔고 한다.
* 출처 [(https://ko.wikipedia.org/wiki/%EC%A0%95%EB%A0%AC\_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98)]((https:/ko.wikipedia.org/wiki/%EC%A0%95%EB%A0%AC_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98)),
* <https://namu.wiki/w/%EC%A0%95%EB%A0%AC%20%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98>