정렬 구현 원리 및 특징 조사

1. 선택정렬 (Selection Sort)

- 입력 배열이외에 다른 추가 메모리를 요구하지 않는다.

- 해당 순서에 원소를 넣을 위치는 이미 정해져 있고, 어떤 원소를 넣을지 선택하는 알고리즘.  
- 장점 : 자료의 이동 횟수가 미리 결정된다.

- 단점 : 안정성이 떨어진다. 값이 같을 경우 상대적인 위치가 변경될 수도 있다.

**<선택정렬 구현>**

1. 주어진 배열 중에서 최솟값을 찾는다.

2. 그 값을 맨 앞에 위치한 값과 교체한다.

3. 맨 처음 위치를 뺀 나머지 리스트를 같은 방법으로 교체한다.

4. 하나의 원소만 남을 때까지 위의 과정을 반복한다.

-> n번째 자료를 n+1번째 자료부터 마지막 자료까지 비교하여 가장 작은 값을 n번째에 놓는 과정을 반복하여 정렬를 수행한다. 1번 회전하고나면 가장 작은 값의 자료가 맨 앞에 오게 되므로 n+1번째 자료부터 정렬을 시작한다.

|  |
| --- |
| public static void SelectionSort(IList<int> list)  {  // 하나의 원소만 남을 때까지 과정을 반복한다.  for (int i = 0; i < list.Count; i++)  {  int minIndex = i;  // 맨 처음 위치를 뺀 나머지 리스트를 같은 방법으로 교체한다.  for (int j = i + 1; j < list.Count; j++)  {  // 주어진 배열 중 최솟값을 찾는다.  if (list[j] < list[minIndex])  // 그 값을 맨 앞에 위치한 값과 교체한다.  minIndex = j;  }  Swap(list, i, minIndex);  }  } |

2. 삽입정렬

- 모든 요소를 앞에서부터 차례대로 이미 정렬된 배열 부분과 비교하여 자신의 위치를 찾아 삽입함으로써 정렬을 완성하는 알고리즘.

- 장점 : 안정적이다. 알고리즘 자체가 간단하여 레코드가 적을 경우 다른 복잡한 정렬 보다 유리할 수 있다.

- 단점 : 비교적 많은 레코드들의 이동을 포함한다. 레코드 수가 많고 크기가 클 경우 적합하지 않다.

**<삽입정렬 구현>**

1. 두 번째 자료부터 시작하여 왼쪽의 자료들과 비교한다.

2. 삽입할 위치를 지정한 후 자료를 뒤로 옮긴다.

3. 지정한 자리에 자료를 삽입하여 정렬한다.

-> n(2 이상)번째 자료는 n-1, n-2 .. 번째 자료들과 비교해 삽입할 위치를 찾고 그 위치에 자료를 삽입하기 위해 자료를 한 칸씩 뒤로 이동시키는 정렬이다.

|  |
| --- |
| public static void InsertionSort(IList<int> list)  {  // 인덱스 0은 이미 정렬된 것으로 본다.  for (int i = 1; i < list.Count; i++)  {  // 현재 삽입될 숫자인 list[i]를 key 변수로 복사한다.  int key = list[i];  int j;  // 현재 정렬된 배열은 i-1까지이므로 i-1번째부터 역순으로 조사한다.  // j 값은 음수가 아니어야 되고  for (j = i - 1; j >= 0 && key < list[j]; j--)  {  // key 값보다 정렬된 배열에 있는 값이 크면 j번째를 j+1번째로 이동  list[j + 1] = list[j];  }  list[j + 1] = key;  }  } |

3. 버블정렬

- 서로 인접한 두 원소를 비교하여 크기가 순서대로 되어 있지 않으면 서로 교환하여 정렬하는 알고리즘.

- 장점 : 구현이 매우 간단하다.

- 단점 : 특정 요소가 최종 정렬 위치에 이미 있는 경우라도 교환되는 경우가 있기도 하며 일반적으로 자료의 교환 작업이 자료의 이동 작업보다 더 복잡하기 때문에 잘 쓰이지 않는다.

**<버블정렬 구현>**

1. 첫 번째 자료와 두 번째 자료를 비교한다

2. 이어서 두 번째 자료와 세 번째 자료를 비교한다.

3. 이런 식으로 end-1 번째 자료와 end를 비교하며 자료를 정렬한다.

4. end가 가장 큰 자료가 될 테니 end를 제외하고 정렬을 반복한다.

-> 과정을 반복하다보면 1회 수행마다 정렬에서 제외되는 데이터가 하나씩 늘어나며 정렬이 완성된다.

|  |
| --- |
| public static void BubbleSort(IList<int> list)  {  for (int i = 0; i < list.Count; i++)  {  for (int j = 1; j < list.Count; j++)  {  if (list[j - 1] > list[j])  Swap(list, j - 1, j);  }  }  } |

-- 선형 정렬

버블 정렬 : 셋 중에 제일 느리지만 단순함

선택 정렬 : 버블 정렬보다 좋음

삽입 정렬 : 셋 중에 제일 빠르지만 배열이 길어질수록 효율이 떨어짐

모두 시간 복잡도는 O(n^2)지만 선택, 삽입 정렬은 사용할 메모리가 제한적인 경우 사용하면 좋음

-- 분할정복 정렬

n개의 요소를 재위치시키기 위해 n/2개를 확인하는 정렬

시간 복잡도 : O(NlogN)

4. 힙정렬

- 최대 힙 트리나 최소 힙 트리를 구성해 정렬하는 방법. 내림차순 정렬은 최대 힙, 오름차순은 최소 힙을 구성하면 된다.

**<힙 정렬 구현>**

1. 정렬해야 할 n개의 요소들로 최소 힙을 만든다. -> 오름차순

2. 한 번에 하나씩 요소를 힙에서 꺼내서 배열에 저장한다.

-> n개의 요소들로 완전 이진 트리 형태를 만든 후, 최소 힙일 경우 데이터를 하나씩 꺼내면 최솟값부터 정렬된 상태로 나오게 된다. 이 데이터를 배열에 복사해 저장하면 된다.

|  |
| --- |
| public static void HeapSort(IList<int> list)  {  // 우선순위 큐 사용  PriorityQueue<int, int> pq = new PriorityQueue<int, int>();  // 우선순위 큐에 list를 Enqueue해 힙 트리를 만든다.  for (int i = 0; i < list.Count; i++)  {  pq.Enqueue(list[i], list[i]);  }  // 만든 힙 트리를 하나씩 Dequeue하며 list에 넣는다.  for (int i = 0; i < list.Count; i++)  {  list[i] = pq.Dequeue();  }  } |

**<힙 정렬의 특징>**

- 힙 정렬은 추가적인 배열(메모리)이 필요하지 않아서 병합정렬보다 효율적이다.

- 힙 정렬은 먼저 힙 구조로 만든 후 정렬을 수행하기 때문에 O(nlogN)의 시간복잡도를 가진다.

- 이진 트리를 이용한 힙 정렬은 동일한 값의 순서까지 지켜줄 수 없기 때문에 불안정 정렬이다.

- 이론상 퀵 정렬보다 빠르지만 최대원소 제거 후 힙 구조를 재조정하는 연산이 배열을 전체적으로 다 확인하고 지나가기 때문에 참조지역성의 특성을 활용한 cache의 성능가속을 얻기가 어려워져 실제 프로그램에선 퀵 정렬보다 느리다.

5. 병합정렬

**<병합정렬 구현>**

1. 리스트의 길이가 0이나 1이면 이미 정렬된 것으로 본다.

2. 분할(Divide) : 입력한 배열을 같은 크기의 2개의 부분 배열로 분할한다.

3. 정복(Conquer) : 부분 배열을 정렬한다. 크기가 작지 않다면 다시 분할 정복을 적용한다.

4. 결합(Combine) : 정렬된 부분 배열들을 하나의 배열에 병합한다. -> 실제 정렬 수행하는 중요한 부분

**<병합정렬 특징>**

- 추가적인 리스트(메모리)가 필요하다.

- 정확히 절반씩 나눈다는 점에서 최악의 경우에도 O(NlogN)을 보장한다.

- 정렬에 사용되는 배열은 전역 변수로 선언해야 한다. 함수 안에서 배열을 선언하게 될 경우 매번 배열을 선언해야 한다는 점에서 메모리 자원의 낭비가 커질 수 있다. -> 기존의 데이터를 담을 추가적인 배경 공간이 필요하다는 점에서 메모리 활용이 비효율적인 문제가 있다.

- 병합 정렬은 추가 배열을 사용하는 반면 퀵 정렬은 추가 메모리를 사용하지 않고 내부 교환만으로 수행되기에 최악의 경우가 아닐 경우 퀵 정렬이 병합 정렬에 비해 빠른 성능을 보인다. 하지만 안정적 정렬이라는 장점이 있다.

6. 퀵정렬

**<퀵 정렬 구현>**

1. 분할(Divide) : 입력 배열을 피벗을 기준으로 비균등하게 2개의 부분 배열로 분할한다.

=> (피벗을 중심으로 왼쪽은 피벗보다 작은 요소들, 오른쪽은 피벗보다 큰 요소들)

2. 정복(Conquer) : 분할한 양쪽을 각각 재귀로 퀵 정렬한다.

3. 결합(Combine) : 정렬된 부분 배열들을 하나의 배열에 병합한다.

-> 한번 진행될 때마다 최소한 하나의 원소(Pivot)는 최종적으로 위치가 정해지므로, 이 알고리즘은 반드시 끝난다는 것을 보장할 수 있다.

-> 피벗을 선택한 뒤 (첫 번째를 선택했다는 기준) 왼쪽에서 오른쪽으로 가며 피

**<퀵 정렬 특징>**

- Pivot을 어떤 값으로 설정하는지에 따라 퀵 정렬의 성능이 좌우된다.

- 추가 메모리 공간을 필요로 하지않고 내부적으로 순환해 정렬을 수행한다.

- 정렬된 리스트에 대해서는 퀵 정렬의 불균형 분할에 의해 오히려 수행시간이 늘어난다.

-> 퀵 정렬의 불균형 분할을 방지하기 위해 피벗을 선택할 때 더욱 리스트를 균등하게 분할할 수 있는 데이터를 선택한다. ( 크기 순으로 중간 값을 피벗으로 선택한다. )