ch.06 정렬 알고리즘

정렬 알고리즘

내부정렬

: 입력의 크기 ≤ 주기억 장치의 공간

- 비교 정렬
  - 버블, 선택, 힙
  - 합병, 퀵, 힙, 쉘
- 기수 : 입력이 제한된 크기 이내의 숫자로 구분

외부정렬

: 입력의 크기 ≥ 주기억 장치의 공간

6.1 버블 정렬

(swapped variable 추가하면 더 개선된 ver.)
↳ 이미 정렬된 원소라면 또 정렬할 이유 X

이웃하는 숫자를 비교하여 작은 수를 앞쪽으로 이동하는 과정을 반복하여 정렬

- 입력 크기를 n이라고 할 때

```
① for pass= 1 to n-1 //입력을 한 번 처리하는 것
② for i=1 to n-pass
     if(A[i-1] > A[i])
       A[i-1] <-> A[i]

return 배열 A
```

time complexity

- for loop : (n-1) + (n-2) + … +1 = (n-1)n / 2
- if : O(1)

⇒ n(n-1)/2 * O(1) = O(n^2)

6.2 선택 정렬

입력 배열 전체에서 최솟값을 선택하여 배열의 0번째 원소와 자리바꿈을 반복

- 입력에 민감하지 않고 항상 일정한 time complexity → input insensitive
- 입력 크기를 n이라고 할 때

```
for i=0 to n-2
   min=i
   for j=i+1 to n-1 //A{i] ~ A[n-1]에서 최솟값을 찾는다
     if(A[j] < A[min])
       min=j
   A[i] <-> A[min] //min이 최솟값이 있는 원소의 인덱스

return 배열 A
```

time complexity

- for loop : (n-1) + (n-2) + … +1 = (n-1)n / 2
- if : O(1)

⇒ n(n-1)/2 * O(1) = O(n^2)

6.3 삽입 정렬

정렬된 부분, 정렬 안 된 부분으로 나눈 뒤 안 된 부분의 가장 왼쪽 원소를 정렬된 부분에 삽입 ⇒ 정렬된 부분 탐색

- 입력의 상태에 따라 수행 시간이 달라질 수 있음 ⇒ input sensitive
- 입력 크기를 n이라고 할 때

```
for i=1 to n-1{
  currentelement = A[i] //정렬이 안 된 부분의 가장 왼쪽 원소
  j <- i-1 //정렬된 부분의 가장 오른쪽부터 왼쪽 방향으로 삽입할 곳 탐색
                ↑ 벗어나지 X
while(j>=0 and A[j] > currentelement){
    A[j+1]=A[j] //자리 이동
    j <- j-1
  }
  A[j+1] <- currentelement
}
return 배열 A
```

time complexity

- 최악의 경우(=평균의 경우)
  - (n-1) + (n-2) + … +1 = (n-1)n / 2 → while loop 실행 횟수
  - O(1) → if가 참일 경우 → while loop 내부 수행 시간
  - n(n-1)/2 * O(1) = O(n^2)
- 평균이 최악과 같은 이유(연습 문제 정리할 것)
- 최적의 경우
  - 입력이 이미 정렬되어 있다면 n-1번의 비교 후 끝

(n-1)*O(1) ∝ O(n)

- 거의 정렬된 입력에 대해서는 다른 정렬 알고리즘보다 빠르다는 장점

6.4 shell 정렬

간격을 정하여 해당 간격 내에서 정렬 후 간격을 계속 줄여 나감

- 입력 크기를 n이라고 할 때,

```
for each gap h = [h0 > h1 > ... > hk = 1] //큰 gap부터 차례대로
  for i = h to n-1 { //삽입정렬
    CurrentElement = A[i]; //간격의 첫 시작점
    j=i;
    while(j >= h) and (A[j-h] > CurrentElement) { //배열의 범위 벗어나지 않도록
      A[j] = A[j-h];
      j = j-h;
    }
    A[j] = CurrentElement;
}

return A;
```

time complexity

- shell 정렬 수행 속도 : 간격 선정에 따라 좌우됨
- Hibbard 간격 = 2의 k승 - 1 → O(n의 1.5승)
- 입력 크기가 매우 크지 않은 경우에 좋은 성능을 보임

6.5 힙정렬

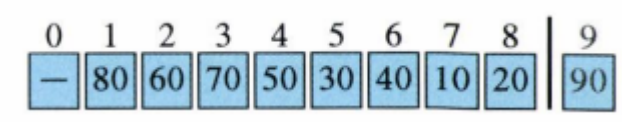
- heap : 완전 이진트리
  - 가장 높은 우선순위(가장 크거나 작은 값)이 root에 저장됨

- 부모노드 : A[i/2], 왼쪽 자식노드 : A[2i], 오른쪽 자식노드 : A[2i+1]

- HeapSort → maximum heap
  - 입력 크기가 n

```
배열 A에 대한 heap 생성

heapsize = n //heap 크기를 조절
for i=1 to n-1
  A[1] <-> A[heapSIZE] //root와 heap의 마지막 노드를 교환
  heapSize = heapSize -1 //heap크기를 1 감소
  DownHeap() //위배된 힙 조건 만족
return A;
```



  - 정렬된 거 하나 씩 제외하면서 heap에서 제거 후 정렬된 배열을 저장
  - heap 크기가 1이 되면 heap sort 종료
  - 선택 정렬에서 배열의 뒤부터 큰 값을 찾으면서 정렬하는 것과 동일
  - but 선택 정렬은 순차 탐색으로 최댓값을 찾음

time complexity

1. heap 생성 → O(n)
   - 변수 초기화 → O(1)
2. for loop → (n-1)번
   - loop 내부 → O(1) 시간
3. downheap, heap의 높이가 log2n 넘지 않음 → O(logn)

∴ ⇒ O(n) + (n-1) * O(logn) = O(nlogn)

6.6 비교정렬의 하한

- n개의 data : 최댓값이 (n-1)번 비교
- 결정 tree 사용 경우 → k개의 노드가 있는 binarytree의 높이 ≥ logk
  - n개의 data → n!의 노드 → log(n!) 높이가 최댓값
  - log(n!) = O(nlogn)
- 비교정렬의 lower bound : Ω(nlogn)

6.7 기수 정렬(radix sort)

제한적인 범위 내에 있는 숫자에 대해서 각 자릿수별로 정렬하는 알고리즘

- 어느 비교정렬 알고리즘보다 빠르다
- stability(안정성) : 정렬된 후에도 중복된 숫자의 순서가 입력과 동일
- 입력 크기 : n

```
for i = 1 to k
  각 숫자의 i자리 숫자에 대해서 안정한 정렬을 수행
return A;
```

time complexity

- for loop → k번 (입력 숫자의 최대 자릿수만큼 반복)
  - n개의 숫자의 i 자릿수를 읽으며 r개로 분류하여 개수를 셈 → O(n+r)

  ⇒ O(k(n+r))
- if, k나 r이 입력 크기인 n보다 매우 작으면 → O(n)