알고리즘

- **효과적인 알고리즘**
  - 입력(Input)
    - 알고리즘 수행에 필요한 자료가 외부에서 입력
  - 출력(Output)
    - 알고리즘 수행 후 하나 이상의 결과를 출력
  - 명확성(Definiteness)
    - 수행할 작업의 내용과 순서를 나타내는 알고리즘의 명령어들은 명확하게 명세
  - 유한성(Finiteness)
    - 알고리즘은 수행 후 반드시 종료
  - 효과성(Effectiveness)
    - 알고리즘의 모든 명령어들은 기본적이며 실행이 가능해야 한다.

알고리즘

- **알고리즘 성능 분석 방법**
  - 공간 복잡도
    - 알고리즘을 프로그램으로 실행하여 완료하기까지 필요한 총 저장 공간의 양
    - 공간 복잡도 = 고정 공간 + 가변 공간
  - 시간 복잡도
    - 알고리즘을 프로그램으로 실행하여 완료하기까지의 총 소요시간
    - 시간 복잡도 = 컴파일 시간 + 실행 시간
      - 컴파일 시간 : 프로그램마다 거의 고정적인 시간 소요
      - 실행 시간 : 컴퓨터의 성능에 따라 달라질 수 있으므로 실제 실행시간 보다는 명령문의 실행 빈도수에 따라 계산
    - 실행 빈도수의 계산
      - 지정문, 조건문, 반복문 내의 제어문과 반환문은 실행시간 차이가 거의 없으므로 하나의 단위시간을 갖는 기본 명령문으로 취급

알고리즘

- **시간 복잡도**
  - n>1의 일반적인 경우에 대한 실행 빈도수
    - n에 따라 for 반복문 수행

```
00 fibonacci(n)
01     if(n<0) then
02         stop;
03     if(n≤1) then
04         return n;
05     f1 ← 0;
06     f2 ← 1;
07     for(i←2;i≤n;i←i+1){
08         fn←f1+f2;
09         f1←f2;
10         f2←fn;
11     }
12     return fn;
13 end
```

행 번호	실행 빈도수	행 번호	실행 빈도수
01	1	08	n-1
02	0	09	n-1
03	1	10	n-1
04	0	11	0
05	1	12	1
06	1	13	0
07	n		

총 실행 빈도수
= 1+0+1+0+1+1+n+(n-1)+(n-1)+(n-1)+0+1+0
= **4n+2**

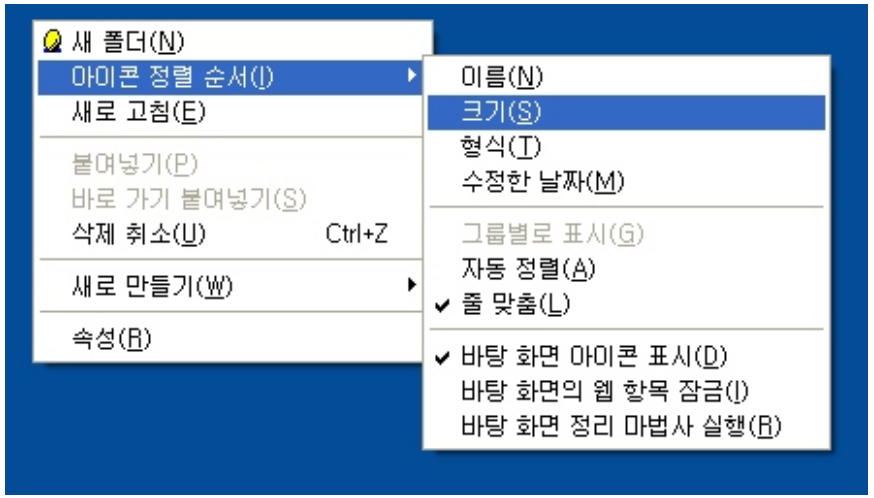
알고리즘

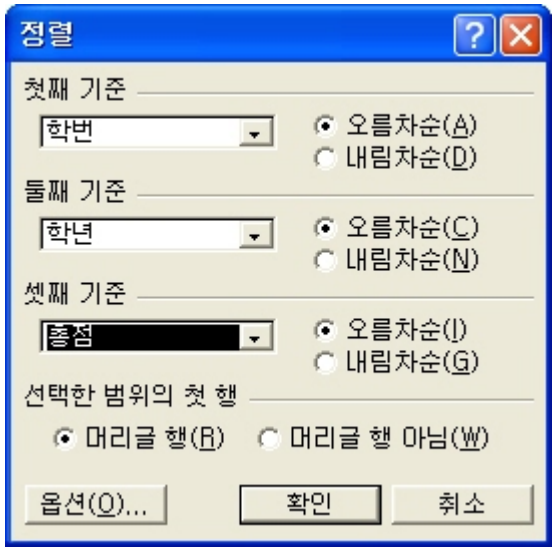
- **알고리즘 성능 분석 방법**
  - 각 실행 시간 함수에서 n값의 변화에 따른 실행 빈도수 비교

n	*O(1)*	*O(log n)*	*O(n)*	*O(n log n)*	*O(n^2)*	*O(n^3)*	*O(2^n)*	*O(n!)*
1	1	1	1	1	1	1	2	1
2	1	1	2	2	4	8	4	2
4	1	2	4	8	16	64	16	24
8	1	3	8	24	64	512	256	40320
16	1	4	16	64	256	4096	65536	…
32	1	5	32	160	1024	32768	4294967296	…

Sorting

- **정렬(sort)**
  - 2개 이상의 자료를 오름차순이나 내림차순으로 재배열하는 것
- **Sort key**
  - 자료를 정렬하는데 사용하는 기준이 되는 특정 값





Sorting

- **정렬방법의 분류**
  - 실행 방법에 따른 분류
    - 비교식 정렬(comparative sort)
      - 비교하고자 하는 각 키 값들을 한번에 두 개씩 비교하여 교환하는 방식으로 정렬을 실행
    - 분산식 정렬(distribute sort)
      - 키 값을 기준으로 하여 자료를 여러 개의 부분 집합으로 분해하고, 각 부분집합을 정렬함으로써 전체를 정렬하는 방식으로 실행

Sorting-Selection Sort

- **선택 정렬(selection sort)**
  - 전체 원소들 중에서 기준 위치에 맞는 원소를 선택하여 자리를 교환하는 방식으로 정렬
  - 수행 방법
    - 전체 원소 중에서 가장 작은 원소를 찾아 선택하여 첫 번째 원소와 자리를 교환한다.
    - 그 다음 두 번째로 작은 원소를 찾아 선택하여 두 번째 원소와 자리를 교환한다.
    - 그 다음에는 세 번째로 작은 원소를 찾아 선택하여 세 번째 원소와 자리를 교환한다.
    - 이 과정을 반복하면서 정렬을 완성한다.

Sort - Bubble Sort

- **선택 정렬 알고리즘 분석**
  - 메모리 사용공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - 비교횟수
    - 1단계 : 첫 번째 원소를 기준으로 n개의 원소 비교
      2단계 : 두 번째 원소를 기준으로 마지막 원소까지 n-1개의 원소 비교
      3단계 : 세 번째 원소를 기준으로 마지막 원소까지 n-2개의 원소 비교
    - i 단계 : i 번째 원소를 기준으로 n-i개의 원소 비교

$$\text{전체 비교횟수} = \sum_{i-1}^{n-1} n-i = \frac{n(n-1)}{2}$$

  - 어떤 경우에서나 비교횟수가 같으므로 시간 복잡도는 **$O(n^2)$**이 된다.

Sort - Bubble Sort

- **버블 정렬(bubble sort)**
  - 인접한 두 개의 원소를 비교하여 자리를 교환하는 방식
    - 첫 번째 원소부터 마지막 원소까지 반복하여 한 단계가 끝나면 가장 큰 원소가 마지막 자리로 정렬
    - 첫 번째 원소부터 인접한 원소끼리 계속 자리를 교환하면서 맨 마지막 자리로 이동하는 모습이 방울 모양과 같다고 하여 버블(bubble) 정렬

- **버블 정렬 수행 과정**
  - 초기상태: {69, 10, 30, 2, 16, 8, 31, 22}

    ① 인접한 두 원소를 비교하여 자리를 교환하는 작업을 첫 번째 원소부터 마지막 원소까지 차례로 반복하여 가장 큰 원소 69를 마지막 자리로 정렬.

Sort - Bubble Sort

- **버블 정렬 알고리즘 분석**
  - 메모리 사용공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - 연산 시간
    - 최선의 경우 : 자료가 이미 정렬되어있는 경우
      - 비교횟수 : i번째 원소를 (n-i)번 비교하므로, n(n-1)/2 번
      - 자리교환횟수 : 자리교환이 발생하지 않음
    - 최악의 경우 : 자료가 역순으로 정렬되어있는 경우
      - 비교횟수 : i번째 원소를 (n-i)번 비교하므로, n(n-1)/2 번
      - 자리교환횟수 : i번째 원소를 (n-i)번 교환하므로, n(n-1)/2 번
    - 평균 시간 복잡도는 $O(n^2)$이 된다.

Sort - Quick Sort

- **퀵 정렬(quick sort)**
  - 정렬할 전체 원소에 대해서 정렬을 수행하지 않고, 기준 값을 중심으로 왼쪽 부분 집합과 오른쪽 부분 집합으로 분할하여 정렬하는 방법

    - 왼쪽 부분 집합에는 기준 값보다 작은 원소들을 이동시키고, 오른쪽 부분 집합에는 기준 값보다 큰 원소들을 이동시킨다.

    - 기준 값 : 피봇(pivot)
      - 일반적으로 전체 원소 중에서 가운데에 위치한 원소를 선택

Sort - Quick Sort

- **퀵 정렬(quick sort)**
  - 퀵 정렬은 다음의 두 가지 기본 작업을 반복 수행하여 완성
    - 분할(divide)
      - 정렬할 자료들을 기준 값을 중심으로 2개의 부분 집합으로 분할
    - 정복(conquer)
      - 부분 집합의 원소들 중에서 기준 값보다 작은 원소들은 왼쪽 부분 집합으로, 기준 값보다 큰 원소들은 오른쪽 부분집합으로 정렬
      - 부분 집합의 크기가 1 이하로 충분히 작지 않으면 순환호출을 이용하여 다시 분할
- 가장 유명하고, 정렬 알고리즘의 표준적인 방법

Sort - Quick Sort

- **퀵 정렬 수행 방법**
  - 부분 집합으로 분할하기 위해서 **L**과 **R**을 사용

    ①왼쪽 끝에서 오른쪽으로 움직이면서 크기를 비교하여 피봇보다 크거나 같은 원소를 찾아 L로 표시

    ②오른쪽 끝에서 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾아 R로 표시

    ③L이 가리키는 원소와 R이 가리키는 원소를 서로 교환한다.

  - L와 R이 만나게 되면 피봇과 R의 원소를 서로 교환하고, 교환한 위치를 피봇의 위치로 확정
  - 피봇의 확정된 위치를 기준으로 만들어진 새로운 왼쪽 부분 집합과 오른쪽 부분 집합에 대해서 퀵 정렬을 순환적으로 반복 수행
  - 모든 부분 집합의 크기가 1 이하가 되면 퀵 정렬을 종료

Sort - Quick Sort

- **퀵 정렬 알고리즘**

```
quickSort(a[],begin,end)
      if(m<n)then{
            p←partition(a,begin,end);
            quicksort(a[],begin,p-1);
            quicksort(a[],p+1,end);
      }
end quickSort()
```

Sort - Quick Sort

- **파티션 분할 알고리즘**

```
partition(a[],begin,end)
        pivot←(begin+end)/2;
        L←begin;
        R←end;
        while(L<R) do {
                while(a[L]<a[pivot] and L<R) do L++;
                while(a[R]<a[pivot] and L<R) do R--;
                if(L<R) then {//L의 원소와 R의 원소 교환
                        temp←a[L];
                        a[L]←a[R]
                        a[R]←temp;
                }
        }
        temp←a[pivot];//R의 원소와 피봇 원소 교환
        a[pivot]←a[R];
        return R;
end partition()
```

Sort - Quick Sort

- **퀵 정렬 알고리즘 분석**
  - 메모리 사용공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - 연산 시간
    - 최선의 경우
      - 피봇에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 n/2개씩 이등분이 되는 경우가 반복되어 수행 단계 수가 최소가 되는 경우
    - 최악의 경우
      - 피봇에 의해 원소들을 분할하였을 때 1개와 n-1개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대가 되는 경우
    - 평균 시간 복잡도 : $O(n \log_2 n)$
      - 같은 시간 복잡도를 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법임.

Sort - Insert Sort

- **삽입 정렬(insert sort)**
  - 정렬되어있는 부분집합에 정렬할 새로운 원소의 위치를 찾아 삽입하는 방법
  - 정렬할 자료를 두 개의 부분집합 S와 U로 가정
    - 부분집합 S: 정렬된 앞부분의 원소들
    - 부분집합 U: 아직 정렬되지 않은 나머지 원소들
    - 정렬되지 않은 부분집합 U의 원소를 하나씩 꺼내서 이미 정렬되어있는 부분집합 S의 마지막 원소부터 비교하면서 위치를 찾아 삽입
    - 삽입 정렬을 반복하면서 부분집합 S의 원소는 하나씩 늘리고 부분집합 U의 원소는 하나씩 감소하게 한다. 부분집합 U가 공집합이 되면 삽입 정렬이 완성
  - 선택 정렬보다 두 배 정도 빨라서 평균적인 성능이 $O(n^2)$ 알고리즘들 중에서 뛰어난 축으로, 다른 정렬 알고리즘의 일부로도 자주 사용
  - 대입이 많고, 데이터의 상태, 데이터 한 개의 크기에 따라 성능 편차가 심함

Sort - Insert Sort

- **삽입 정렬 알고리즘 분석**
  - 메모리 사용공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - 연산 시간
    - 최선의 경우 : 원소들이 이미 정렬되어있어서 비교횟수가 최소인 경우
      - 이미 정렬되어있는 경우에는 바로 앞자리 원소와 한번만 비교한다.
      - 전체 비교횟수 = $n-1$
      - 시간 복잡도 : $O(n)$
    - 최악의 경우 : 모든 원소가 역순으로 되어있어서 비교횟수가 최대인 경우
      - 전체 비교횟수 = $1+2+3+\cdots+(n-1) = n(n-1)/2$
      - 시간 복잡도 : $O(n^2)$
    - 삽입 정렬의 평균 비교횟수 = $n(n-1)/4$
    - 평균 시간 복잡도 : **$O(n^2)$**

Sort - Shell Sort

- **셸 정렬(shell sort)**
  - 일정한 간격(interval)으로 떨어져있는 자료들끼리 부분집합을 구성하고 각 부분집합에 있는 원소들에 대해서 삽입 정렬을 수행하는 작업을 반복하면서 전체 원소들을 정렬하는 방법
    - 전체 원소에 대해서 삽입 정렬을 수행하는 것보다 부분집합으로 나누어 정렬하게 되면 비교연산과 교환연산 감소
  - 셸 정렬의 부분집합
    - 부분집합의 기준이 되는 간격을 매개변수 h에 저장
    - 한 단계가 수행될 때마다 h의 값을 감소시키고 셸 정렬을 순환 호출
      - h가 1이 될 때까지 반복
  - 셸 정렬의 성능은 매개변수 h의 값에 따라 달라진다.
    - 정렬할 자료의 특성에 따라 매개변수 생성 함수를 사용
    - 일반적으로 사용하는 h의 값은 원소 개수의 1/2을 사용하고 한 단계 수행될 때마다 h의 값을 반으로 감소시키면서 반복 수행

Sort - Shell Sort

- **셸 정렬 알고리즘**

```
shellSort(a[],n)
        interval ← n;
        while(interval≥1) do {
                interval ← interval/2;
                for(i←0;i<interval;i←i+1) do {
                        intervalSort(a[], i, n, interval);
                }
        }
end shellSort()
```

Sort - Shell Sort

- **셸 정렬 알고리즘 분석**
  - 메모리 사용공간
    - n개의 원소에 대하여 n개의 메모리와 매개변수 h에 대한 저장공간 사용
  - 연산 시간
    - 비교횟수
      - 처음 원소의 상태에 상관없이 매개변수 h에 의해 결정
    - 일반적인 시간 복잡도 : **$O(n^{1.25})$**
    - 셸 정렬은 삽입 정렬의 시간 복잡도 $O(n^2)$ 보다 개선된 정렬 방법

Sort - Merge Sort

- **병합 정렬(merge sort)**
  - 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법
  - 부분집합으로 분할(divide)하고, 각 부분집합에 대해서 정렬 작업을 완성(conquer)한 후에 정렬된 부분집합들을 다시 결합(combine)하는 분할 정복(divide and conquer) 기법 사용
  - 병합 정렬 방법의 종류
    - 2-way 병합 : 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
    - n-way 병합 : n개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법

- 병합 정렬의 큰 결점은 데이터 전체 크기만한 메모리가 더 필요
- 시간이 데이터 상태에 별 영향을 받지 않고, 시간 복잡도가 $O(n \log n)$인 알고리즘 중에 유일하게 안정적

Sort - Merge Sort

- 2-way 병합 정렬 : 세 가지 기본 작업을 반복 수행하면서 완성.

⑴ 분할(divide) : 입력 자료를 같은 크기의 부분집합 2개로 분할한다.

⑵ 정복(conquer) : 부분집합의 원소들을 정렬한다.

부분집합의 크기가 충분히 작지 않으면 순환호출을 이용하여 다시 분할 정복 기법을 적용한다.

⑶ 결합(combine) : 정렬된 부분집합들을 하나의 집합으로 결합한다.

Sort - Merge Sort

- **병합 정렬 알고리즘**

```
mergeSort(a[],m,n)
        if(a[m:n]의 원소수>1) then {
                전체 집합을 두 개의 부분집합으로 분할;
                mergeSort(a[], m, middle);
                mergeSort(a[], middle+1, n);
                merge(a[m:middle], a[middle+1:n);
        }
end mergeSort()
```

Sort - Merge Sort

- **병합 정렬 알고리즘 분석**
  - 메모리 사용공간
    - 각 단계에서 새로 병합하여 만든 부분집합을 저장할 공간이 추가로 필요
    - 원소 n개에 대해서 (2 x n)개의 메모리 공간 사용
  - 연산 시간
    - 분할 단계 : n개의 원소를 분할하기 위해서 $\log_2 n$번의 단계 수행
    - 병합 단계 : 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 n번의 비교연산 수행
    - 전체 병합 정렬의 시간 복잡도 : **$O(n \log_2 n)$**

Sort - Radix Sort

- **기수 정렬(radix sort)**
  - 원소의 키값을 나타내는 기수를 이용한 정렬 방법
    - 정렬할 원소의 키 값에 해당하는 버킷(bucket)에 원소를 분배하였다가 버킷의 순서대로 원소를 꺼내는 방법을 반복하면서 정렬
      - **원소의 키를 표현하는 기수만큼의 버킷 사용**
      - **예) 10진수로 표현된 키 값을 가진 원소들을 정렬할 때에는 0부터 9까지 10개의 버킷 사용**
    - 키 값의 자리수 만큼 기수 정렬을 반복
      - 키 값의 일의 자리에 대해서 기수 정렬을 수행하고,
      - 다음 단계에서는 키 값의 십의 자리에 대해서,
      - 그리고 그 다음 단계에서는 백의 자리에 대해서 기수 정렬 수행
    - 한 단계가 끝날 때마다 버킷에 분배된 원소들을 버킷의 순서대로 꺼내서 다음 단계의 기수 정렬을 수행해야 하므로 큐를 사용하여 버킷을 만든다.

Sort - Radix Sort

- **기수 정렬 알고리즘**

```
radixSort(a[], n)
        for(k←1;k≤n;k←k+1) do {
                for(i←0;i<n;i←i+1) do {
                        k번째 자릿수 값에 따라서 해당 버킷에 저장;
                        enqueue(Q[k], a[i]);
                }
                p←-1;
                for(i←0;i≤9;i≤i+1) do {
                        while(Q[i]≠NULL) do {
                                p←p+1;
                                a[p]←dequeuer(Q[i]);
                        }
                }
        }
end radixSort()
```

Sort - Radix Sort

- **기수 정렬 알고리즘 분석**
  - 메모리 사용공간
    - 원소 n개에 대해서 n개의 메모리 공간 사용
    - 기수 r에 따라 버킷 공간이 추가로 필요
  - 연산 시간
    - 연산 시간은 정렬할 원소의 수 n과 키 값의 자릿수 d와 버킷의 수를 결정하는 기수 r에 따라서 달라진다.
      - 정렬할 원소 n개를 r개의 버킷에 분배하는 작업 : (n+r)
      - 이 작업을 자릿수 d 만큼 반복
    - 수행할 전체 작업 : d(n+r)
    - 시간 복잡도 : **O(d(n+r))**

Sort - Heap Sort

- **히프 정렬(heap sort)**
  - 히프 자료구조를 이용한 정렬 방법
  - 히프에서는 항상 가장 큰 원소가 루트 노드가 되고 삭제 연산을 수행하면 항상 루트 노드의 원소를 삭제하여 반환
    - 최대 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 내림차순으로 정렬 수행
    - 최소 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 오름차순으로 정렬 수행
  - 히프 정렬 수행 방법

(1) 정렬할 원소들을 입력하여 최대 히프 구성

(2) 히프에 대해서 삭제 연산을 수행하여 얻은 원소를 마지막 자리에 배치

(3) 나머지 원소에 대해서 다시 최대 히프로 재구성

원소의 개수만큼 (2)~(3) 을 반복 수행

Sort - Heap Sort

- **히프 정렬 알고리즘**

```
heapSort(a[])
        n←a.length-1;
        for(i←n/2;i≥1;i←i-1) do {      //배열 a[]를 허프로 변환
                makeHeap(a, i, n);
        for(i←n-1;i≥1;i←i-1) do {
                temp←a[1];             //히프의 루트 노드 원소를
                a[1]←a[i+1];           //배열의 비어있는
                a[i+1]←temp;           //마지막 자리에 저장

                makeHeap(a, 1, i);
        }
end heapSort()
```

Sort - Heap Sort

- 히프 정렬 알고리즘의 히프 재구성 연산 알고리즘

```
makeHeap(a[], h, m)
        for(i←2*h;j≤m;j←2*j) do {     //배열 a[]를 허프로 변환
                if(j<m) then
                        if(a[j]<a[j+1] then j←j+1;
                if(a[h]≥a[j]) then exit;
                else a[j/2]←a[j];
        }
        a[j/2]←a[h];
end makeHeap()
```

Sort - Heap Sort

- **히프 알고리즘 분석**
  - 메모리 사용공간
    - 원소 n개에 대해서 n개의 메모리 공간 사용
    - 크기 n의 히프 저장 공간
  - 연산 시간
    - 히프 재구성 연산 시간
      - n개의 노드에 대해서 완전 이진 트리는 $\log_2(n+1)$의 레벨을 가지므로 완전 이진 트리를 히프로 구성하는 평균시간은 $O(\log_2 n)$
      - n개의 노드에 대해서 n번의 히프 재구성 작업 수행
    - 평균 시간 복잡도 : **$O(n \log_2 n)$**

Sort – Tree Sort

- **트리 정렬(tree sort)**
  - 8장의 이진 탐색 트리를 이용하여 정렬하는 방법
  - 트리 정렬 수행 방법

(1) 정렬할 원소들을 이진 탐색 트리로 구성한다.

(2) 이진 탐색 트리를 중위 우선 순회 한다.

-중위 순회 경로가 오름차순 정렬이 된다.

Sort – Tree Sort

- **트리 정렬 수행 과정**
  - 초기값 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 트리 정렬 방법으로 정렬하는 과정을 살펴보자.

    ① 원소가 8개를 차례대로 트리에 삽입하여 이진 탐색 트리 구성

    ② 이진 탐색 트리를 중위 우선 순회 방법으로 순회하면서 저장

Sort – Tree Sort

- **트리 정렬 알고리즘**

알고리즘 10-11 트리 정렬 알고리즘

```
treeSort(a[], n)
    for (i ← 0; i < n; i ← i+1) do
        insert(BST, a[i]);          // 이진 탐색 트리의 삽입 연산
    inorder(BST);                   // 중위 순회 연산
end treeSort()
```

Sort – Tree Sort

- **트리 정렬 알고리즘 분석**
  - 메모리 사용공간
    - 원소 n개에 대해서 n개의 메모리 공간 사용
    - 크기 n의 이진 탐색 트리 저장 공간
  - 연산 시간
    - 노드 한 개에 대한 이진 탐색 트리 구성 시간 : $O(\log_2 n)$
    - n개의 노드에 대한 시간 복잡도 : **$O(n \log_2 n)$**