7장. 계산복잡도의 소개: 정렬 문제

계산복잡도
Computational Complexity

- 알고리즘의 분석
  - ✓ 어떤 특정 알고리즘의 효율(efficiency)을 측정
  - ✓ 시간복잡도(time complexity)
  - ✓ 공간복잡도(space/memory complexity)

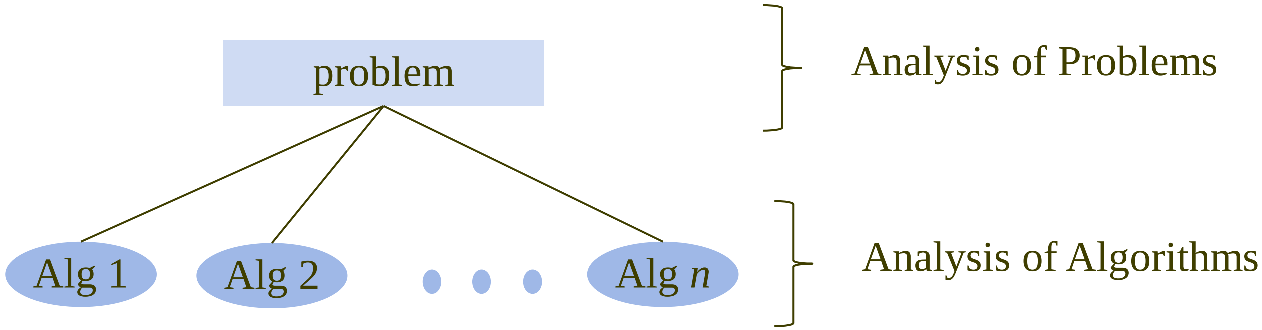
- 문제풀이 접근하는 2가지 방법

  (1) 문제를 푸는 더 효율적인 알고리즘을 개발

  (2) 더 효율적인 알고리즘 개발이 불가능함을 증명

  (예) 정렬문제인 경우 $\Theta(n \log n)$ 보다 좋은 알고리즘은 불가능함이 입증되었음.

- 문제의 분석
  - ✓ 일반적으로 “계산복잡도 분석”이란 “문제의 분석”을 지칭
  - ✓ 어떤 문제에 대해서 그 문제를 풀 수 있는 모든 알고리즘의 효율의 하한(lower-bound)을 결정한다.



(예) 행렬곱셈 문제

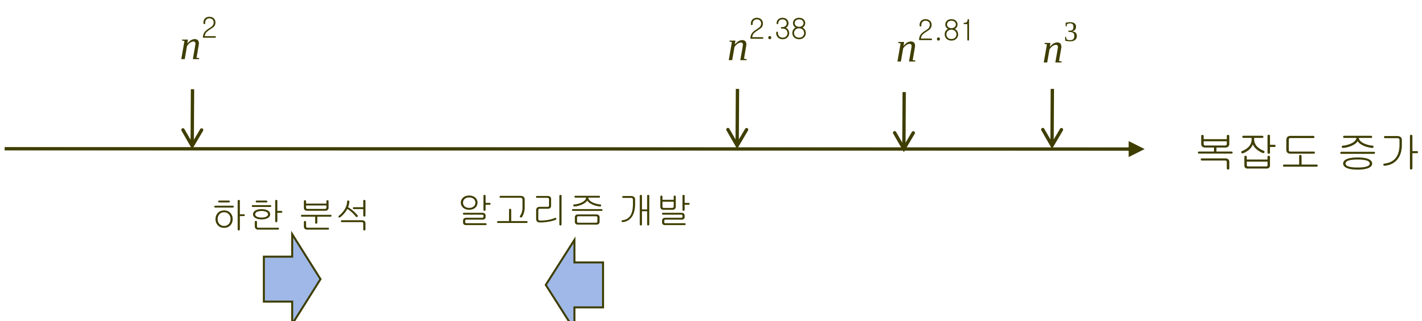
- 일반알고리즘: $\Theta(n^3)$
- Strassen의 알고리즘: $\Theta(n^{2.81})$
- Coppersmith/Winograd의 알고리즘: $\Theta(n^{2.38})$

Analysis of Algorithms

- 이 문제의 복잡도의 하한은 $\Omega(n^2)$
  - ✓ 이는 $\Theta(n^2)$ 알고리즘이 반드시 존재한다는 것을 의미하는 것은 아님.
  - ✓ $\Theta(n^2)$보다 더 좋은 알고리즘을 개발하는 것이 불가능함을 의미

Analysis of Problems

- 더 빠른 알고리즘이 존재할까?
  - ✓ 아직 이 하한 만큼 좋은 알고리즘을 찾지 못하였고,
  - ✓ 그렇다고 하한이 이보다 더 큰 것도 입증하지 못하였다.



계산복잡도

- 복잡도 하한이 $\Omega(f(n))$인 문제에 대해서 복잡도가 $\Theta(f(n))$인 알고리즘을 만들어 내는 것이 목표이다.
- 문제의 복잡도 하한보다 낮은 알고리즘을 만들어 낸다는 것은 불가능하다. (물론 상수적으로 알고리즘을 향상 시키는 것은 가능하다.)
- 보기: 정렬문제(sorting)
  - ✓ 교환정렬(Exchange sort): $\Theta(n^2)$
  - ✓ 합병정렬(Mergesort): $\Theta(n \lg n)$
  - ✓ 정렬문제의 계산복잡도 하한은 $\Omega(n \lg n)$ (키를 비교하여 정렬하는 경우에만 해당됨) – 키의 성질을 이용할 경우는 향상 시킬 수 있음.
  - ✓ 이 정렬 문제의 경우는 하한 만큼의 시간 복잡도를 가진 알고리즘을 찾았다.

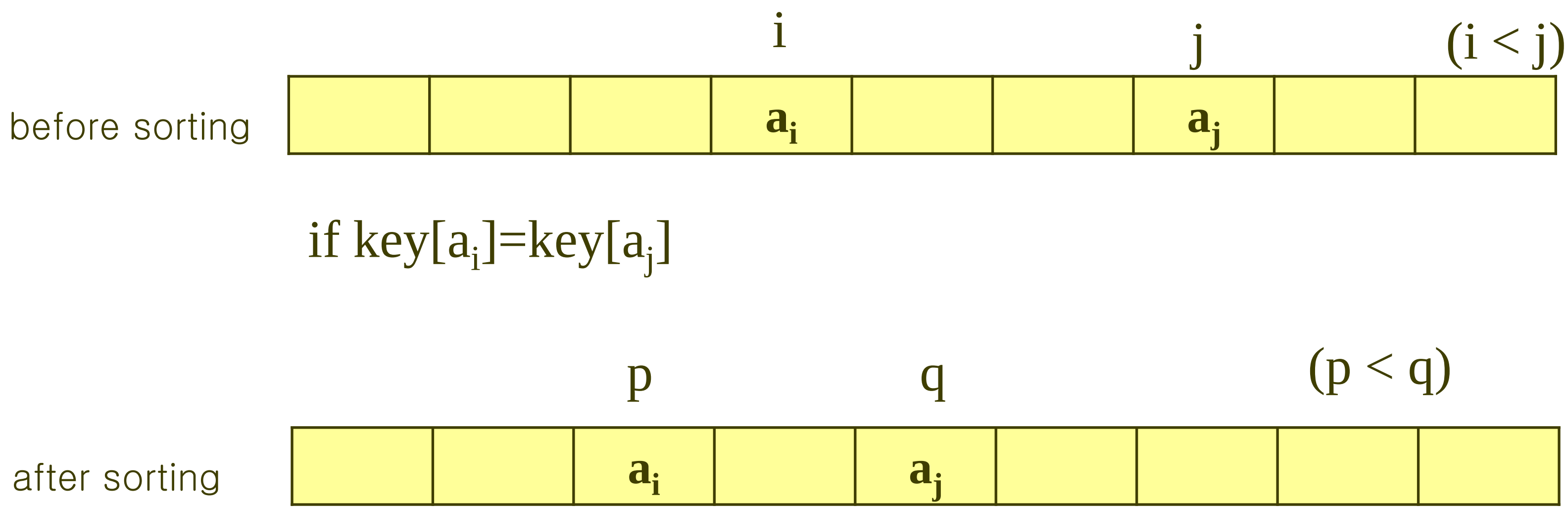
- 키의 비교횟수와 레코드의 지정(assignment) 횟수의 형식으로 알고리즘 분석

- ```
  temp = s[i];
  s[i] = s[j];
  s[j] = temp;
  ```

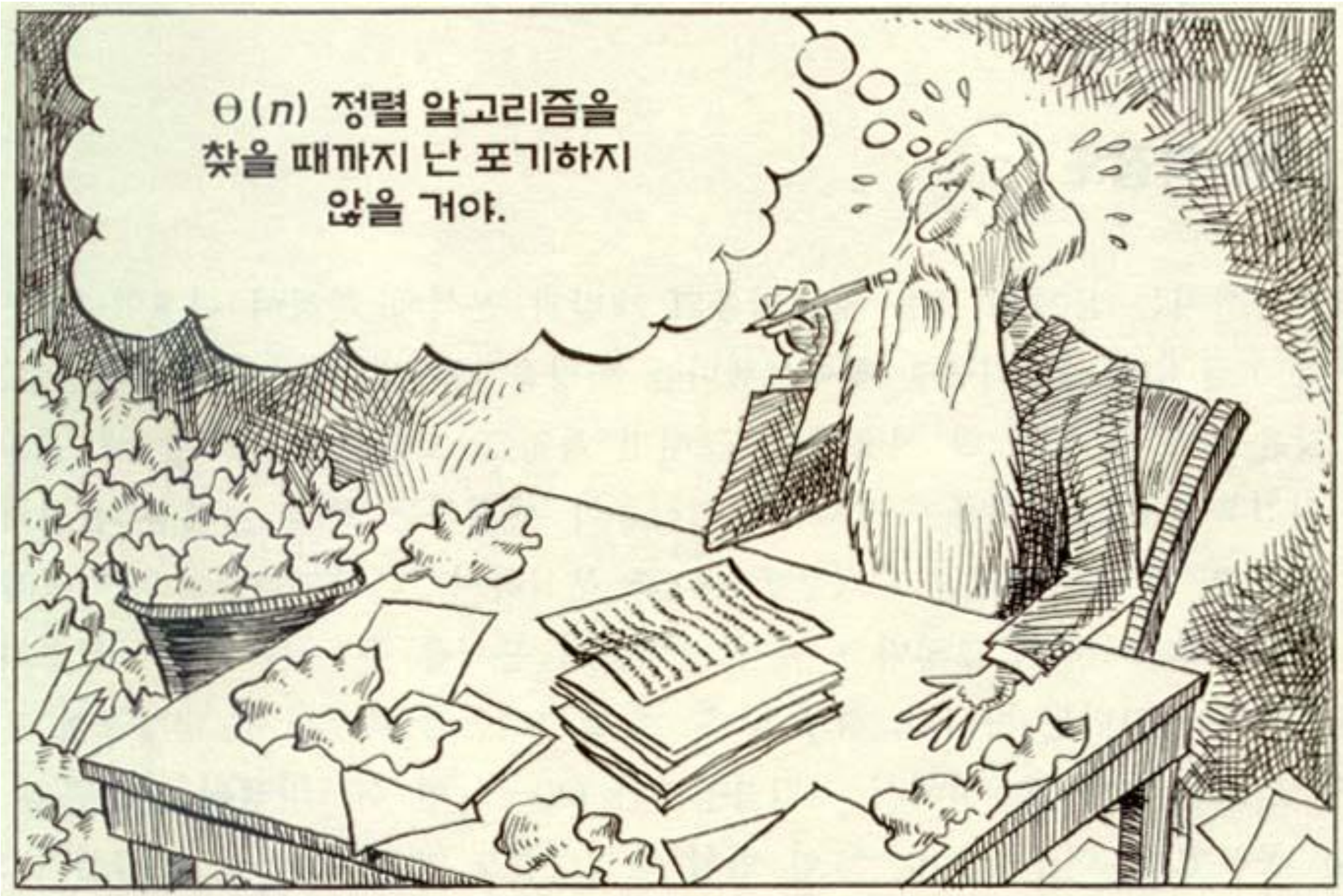
  - ✓ 이 경우 한 번의 교환이지만, 3번의 지정문 필요
  - ✓ 레코드의 크기가 크면 레코드를 지정하는 데 걸리는 시간이 길어지므로 분석에 포함

- 제자리 정렬(in-place sort) : 추가적으로 소요되는 저장장소가 상수
  ```

Stability

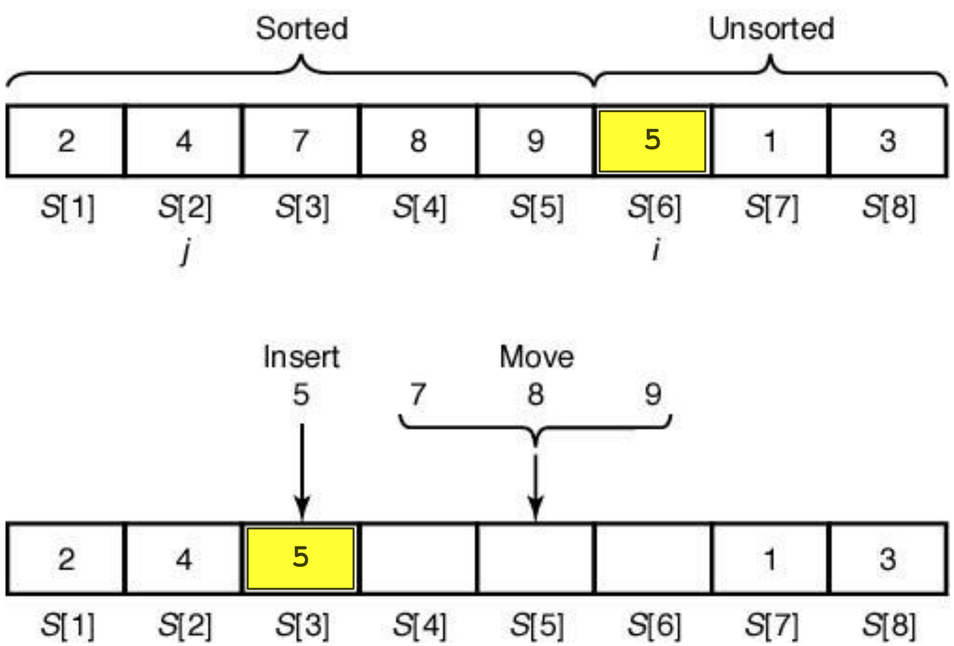


- ✓ 같은 키값을 갖는 데이터간의 정렬 전 순서가 정렬 후에도 유지되는 성질
- ✓ 이러한 성질을 갖는 정렬방법은 stable 하다고 한다.
- ✓ stable: insertion sort, merge sort, bubble sort – 추가적인 구현으로 stable하게 만들 수 있다.
- ✓ not stable: quick sort, heap sort, selection sort, exchange sort


Θ(n) 정렬 알고리즘을 찾을 때까지 난 포기하지 않을 거야.

삽입정렬 알고리즘 (Insertion Sort)

- 이미 정렬된 배열에 항목을 끼워 넣음으로써 정렬하는 알고리즘
- **알고리즘: 삽입정렬**
  - ✓ 문제: 비내림차순으로 *n*개의 키를 정렬
  - ✓ 입력: 양의 정수 *n*; 키의 배열 S[1..*n*]
  - ✓ 출력: 비내림차순으로 정렬된 키의 배열 S[1..*n*]



삽입정렬 알고리즘

```
void insertionsort(int n, keytype S[]){
      index i,j;
      keytype x;

      for(i=2; i<=n; i++){
        x = S[i];
        j = i - 1;
        while(j>0 && S[j]>x){
          S[j+1] = S[j];
          j--;
        }
        S[j+1] = x;
      }
    }
```

삽입정렬 알고리즘의 분석

- S[j]와 x를 비교하는 횟수를 기준:

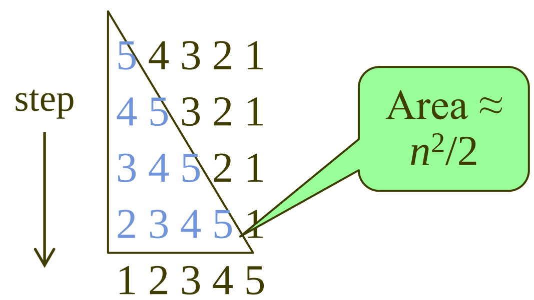
✓ 최악의 경우 시간복잡도 분석

*i*가 주어졌을 때, while-루프에서 최대한 *i*-1번의 비교가 이루어진다. 그러면 비교하는 총 횟수는 최대한

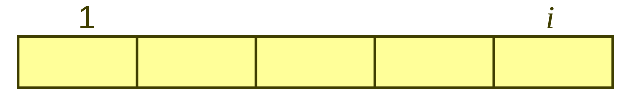
$$W(n) = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$$

(ex) 5 4 3 2 1 → 4 5 3 2 1 → 3 4 5 2 1
→ 2 3 4 5 1 → 1 2 3 4 5

```
void insertionsort(int n, keytype S[]){
  index i,j;
  keytype x;
  for(i=2; i<=n; i++){
    x = S[i];
    j = i - 1;
    while(j>0 && S[j]>x){
        S[j+1] = S[j];
        j--;
        }
    S[j+1] = x;
     }
}
```



삽입정렬 알고리즘의 분석



- ✓ 평균의 경우 시간복잡도 분석

  i가 주어졌을 때, x가 삽입될 수 있는 장소가 i개 있다.

삽입할장소의인덱스	1	2	$\cdots$	$i-1$	i
비교횟수	$i-1$	$i-1$	$\cdots$	2	1

j=0에 의해 중지

- ✓ x를 삽입하는데 필요한 비교 횟수는:

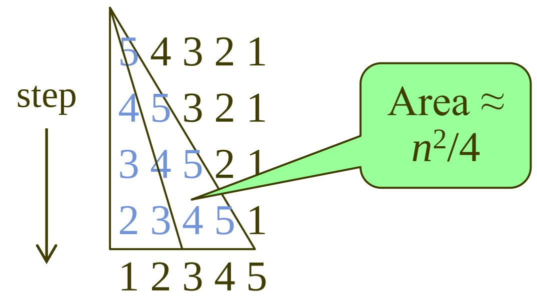
$$1\times\frac{1}{i}+2\times\frac{1}{i}+\cdots+(i-1)\times\frac{1}{i}+(i-1)\times\frac{1}{i}=\frac{1}{i}\sum_{k=1}^{i-1}k+\frac{i-1}{i}=\frac{(i-1)i}{2i}+\frac{i-1}{i}=\frac{i+1}{2}-\frac{1}{i}$$

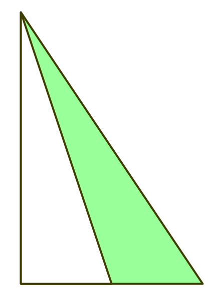
정렬 후 해당 위치의 데이터가 될 확률

```
void insertionsort(int n, keytype S[]){
  index i,j;
  keytype x;
  for(i=2; i<=n; i++){
    x = S[i];
    j = i - 1;
    while(j>0 && S[j]>x){
        S[j+1] = S[j];
        j--;
        }
    S[j+1] = x;
     }
}
```

따라서 정렬하는데 필요한 평균 비교 횟수는:

$$\sum_{i=2}^{n}\left(\frac{i+1}{2}-\frac{1}{i}\right)=\sum_{i=2}^{n}\frac{i+1}{2}-\sum_{i=2}^{n}\frac{1}{i}\approx\frac{(n+4)(n-1)}{4}-\ln n\approx\frac{n^2}{4}$$





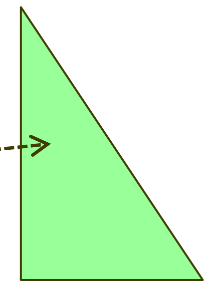
- 공간복잡도 분석:
  - in-place sorting algorithm
  - 저장장소가 추가로 필요하지 않다.
  - 따라서 M(n) = Θ(1).

삽입정렬 알고리즘의 분석

- 레코드의 지정 횟수를 기준(첨자변경 제외):

✓ 최악의 경우 시간복잡도 분석

$$W(n)=\sum_{i=2}^{n}((i-1)+2)=\frac{(n+4)(n-1)}{2}\approx\frac{n^2}{2}$$



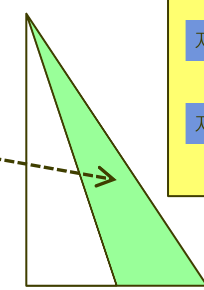
(ex) $\underline{5}$ 4 3 2 1 → $\underline{4}$ $\underline{5}$ 3 2 1 → $\underline{3}$ $\underline{4}$ $\underline{5}$ 2 1
→ $\underline{2}$ $\underline{3}$ $\underline{4}$ $\underline{5}$ 1 → 1 2 3 4 5

```
void insertionsort(int n, keytype S[]){
    index i,j; keytype x;

    for(i=2; i<=n; i++){
지정 →  x = S[i];
        j = i - 1;
        while(j>0 && S[j]>x){
지정 →      S[j+1] = S[j];
            j--;
        }
지정 →  S[j+1] = x;
    }
}
```

✓ 평균의 경우 시간복잡도 분석 :

$$A(n)=\frac{n(n+7)}{4}-1\approx\frac{n^2}{4}$$



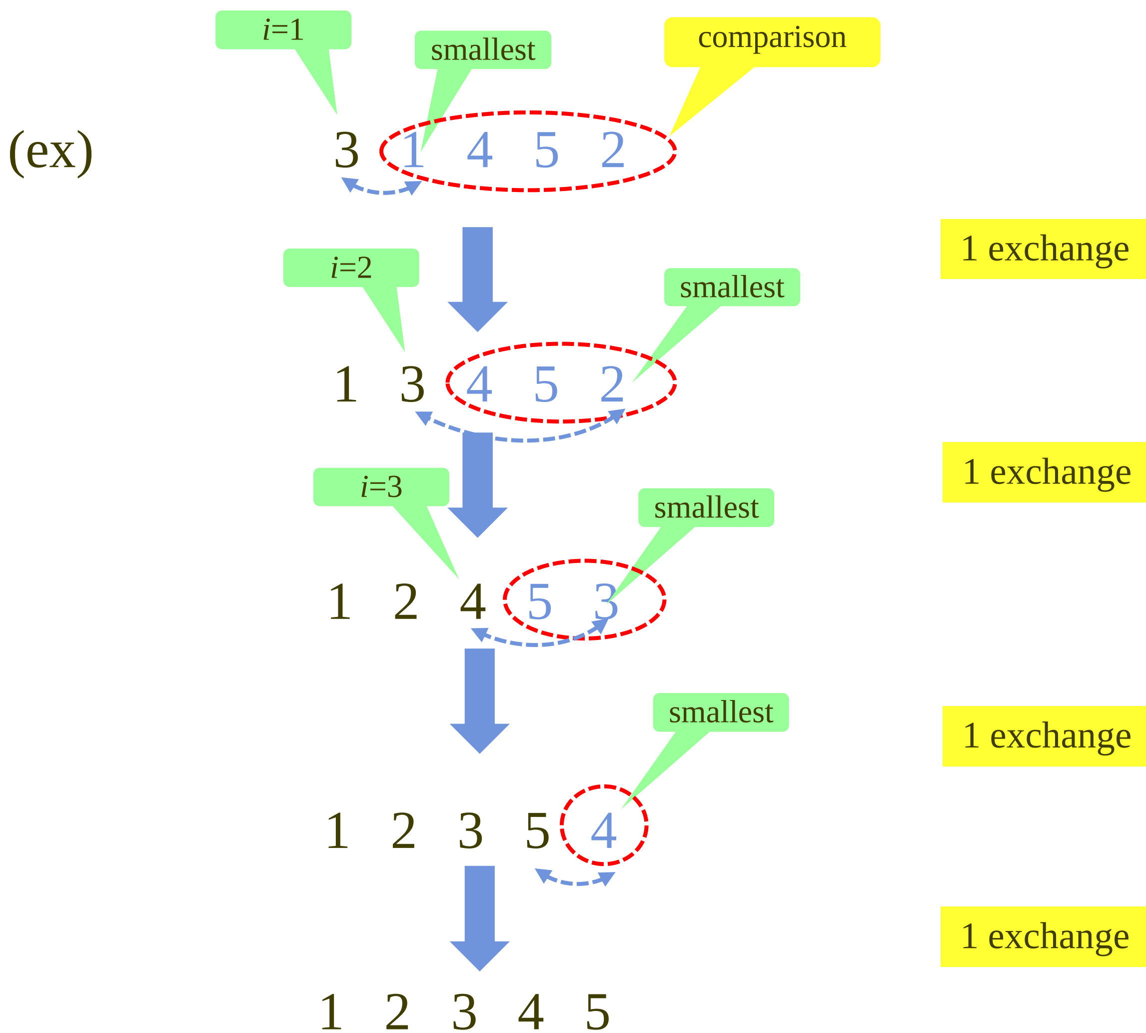
algorithm	number of comparisons	number of assignments	extra space
insertion sort	$W(n) = n^2/2$ $A(n) = n^2/4$	$W(n) = n^2/2$ $A(n) = n^2/4$	in-place sort

선택정렬 알고리즘(selection sort)

- 문제: 비내림차순으로 *n*개의 키를 정렬
- 입력: 양의 정수 *n*; 키의 배열 S[1..*n*]
- 출력: 비내림차순으로 정렬된 키의 배열 S[1..*n*]

```
void selectionsort(int n, keytype S[]){
         index i, j, smallest;

         for(i=1; i<=n-1; i++){
           smallest = i;
           for(j=i+1; j<=n; j++)
             if (S[j]<S[smallest])
               smallest = j;
           exchange S[i] and S[smallest];
         }
       }
```




```
void selectionsort(int n, keytype S[]){
index i, j, smallest;

for(i=1; i<=n-1; i++){
    smallest = i;
    for(j=i+1; j<=n; j++)
        if (S[j]<S[smallest])
            smallest = j;
    exchange S[i] and S[smallest];}
}
```

(ex) 5 4 3 2 1 ?

선택정렬 알고리즘의 분석

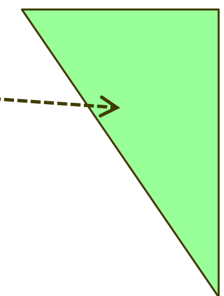
```
void selectionsort(int n, keytype S[]){
index i, j, smallest;

for(i=1; i<=n-1; i++){
    smallest = i;
    for(j=i+1; j<=n; j++)
        if (S[j]<S[smallest])
            smallest = j;
    exchange S[i] and S[smallest];}
}
```

- **비교하는 횟수를 기준**
  - ✓ **모든 경우 시간복잡도 분석 :**

    i가 1일 때 비교횟수는 n-1, i가 2일 때 비교횟수는 n-2,…, i가 n-1일 때 비교횟수는 1이 된다. 이를 모두 합하면,

$$T(n) = \frac{n(n-1)}{2}$$



- **지정(assignment)하는 횟수를 기준** (첨자변경 제외):
  - ✓ 1번 교환하는데 3번 지정하므로 $T(n) = 3(n-1)$

algorithm	**number of comparisons**	**number of assignments**	**extra space**
selection sort	$T(n) = n^2/2$	$T(n) = 3n$	in-place sort

(ex) S=[4,4,1,5]

[4a, 4b, 1, 5] ➡ [1, 4b, 4a, 5]

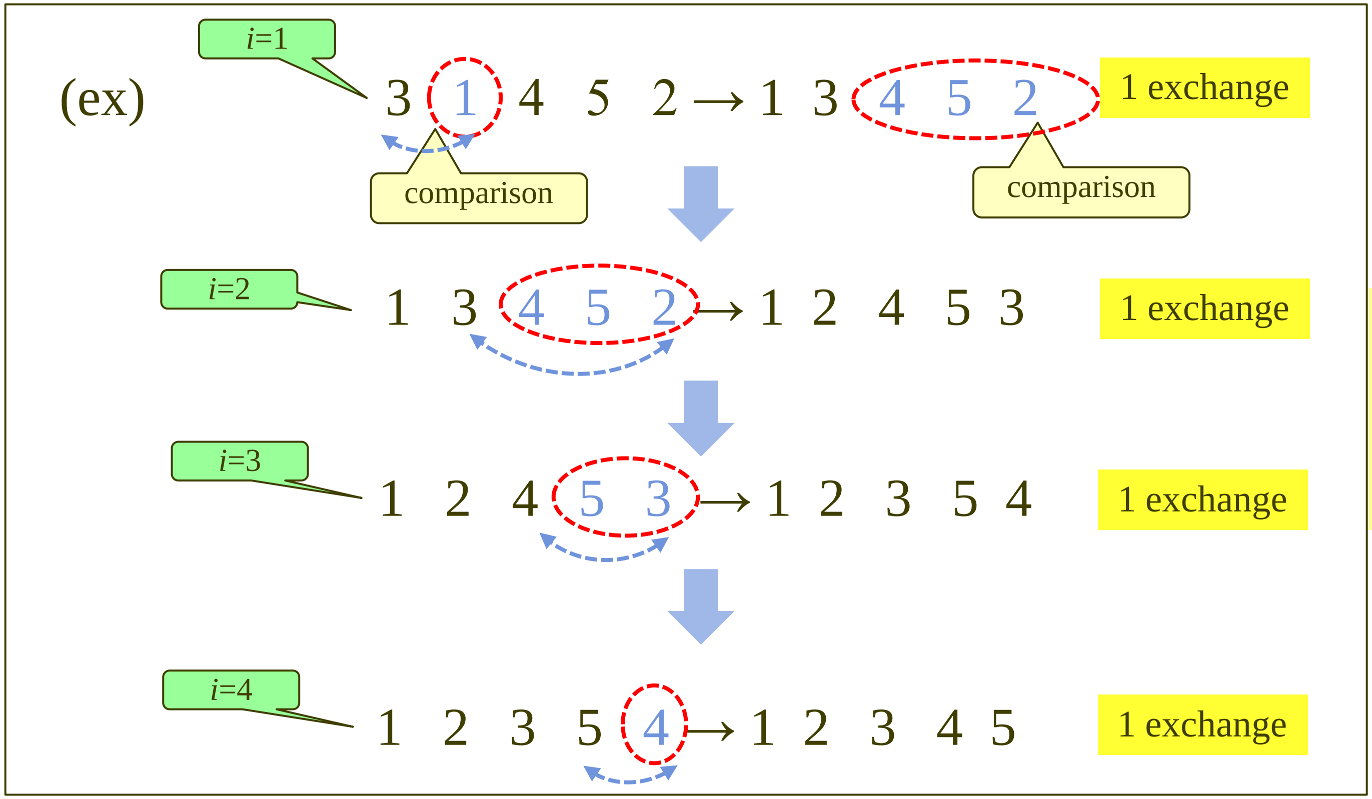
- Not stable

교환정렬 알고리즘(Exchange Sort)

문제: 비내림차순(nondecreasing order)으로 *n*개의 키를 정렬하라
입력: 양의 정수 *n*, 키의 배열 S(첨자는 1부터 *n*)
출력: 키가 비내림차순으로 정렬된 배열 S

```
void  exchangesort(int n, keytype S[ ]) {
      index i,j;

      for (i=1; i<=n-1; i++)
         for (j=i+1; j<=n; j++)
            if(S[j] < S[i])
               exchange S[i] and S[j]
}
```



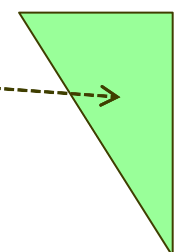
(ex) 5 4 3 2 1 ?

```
void exchangesort(int n, keytype S[ ]){
      index i,j;

      for (i=1; i<=n-1; i++)
         for (j=i+1; j<=n; j++)
            if(S[j] < S[i])
               exchange S[i] and S[j]
}
```

Comparison:

$T(n) = n^2/2$



- 하나의 exchange는 3번의 assignments 필요.
- [worst] 모든 비교마다 exchange 발생
- [average] 비교의 ½경우에 exchange 발생
- $W(n) = 3n^2/2$, $A(n) = 3n^2/4$

algorithm	**number of comparisons**	**number of assignments**	**extra space**
exchange sort	$T(n) = n^2/2$	$W(n) = 3n^2/2$ $A(n) = 3n^2/4$	in-place sort

(ex) S=[4,4,1,5]

[4a, 4b, 1, 5] ➡ [1, 4b, 4a, 5]

- Not stable

거품정렬 (Bubble Sort)

```
void  bubblesort(int n, keytype S[ ]) {
      index  i,j;

      for (i=n; i>=1; i--)
          for (j=2; j<=i; j++)
             if(s[j-1] > s[j])
                 exchange S[j-1] and S[j]
}
```

- ✓ 비교하는 횟수를 기준:

$$W(n) = A(n) = \frac{n(n-1)}{2}$$

- ✓지정(assignment)하는 횟수를 기준:

$$W(n) = \frac{3n(n-1)}{2}, A(n) = \frac{3n(n-1)}{4}$$

(ex)

i=5: 3 1 4 5 2 → 1 3 4 5 2 → 1 3 4 2 5 (sorted: 5) 2 exchanges

i=4: 1 3 4 2 5 → 1 3 2 4 5 (sorted: 4 5) 1 exchange

i=3: 1 3 2 4 5 → 1 2 3 4 5 (sorted: 3 4 5) 1 exchange

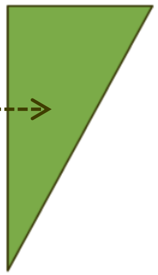
i=2: 1 2 3 4 5 → 1 2 3 4 5 (sorted: 2 3 4 5) 0 exchange

```
void  bubblesort(int n, keytype S[ ]) {
     index  i,j;

     for (i=n; i>=1; i--)
        for (j=2; j<=i; j++)
           if(s[j-1] > s[j])
              exchange S[j-1] and S[j]
}
```

Comparison:

$T(n) = n^2/2$



Assignment:

$W(n) = 3n^2/2,\ A(n) = 3n^2/4$

(ex) 5 4 3 2 1 ?

algorithm	number of comparisons	number of assignments	extra space
bubble sort	$T(n) = n^2/2$	$W(n) = 3n^2/2$ $A(n) = 3n^2/4$	in-place sort

알고리즘	비교횟수	지정횟수	추가저장장소 사용량
삽입정렬	$W(n) = n^2/2$ $A(n) = n^2/4$	$W(n) = n^2/2$ $A(n) = n^2/4$	제자리정렬
선택정렬	$T(n) = n^2/2$	$T(n) = 3n$	제자리정렬
교환정렬	$T(n) = n^2/2$	$W(n) = 3n^2/2$ $A(n) = 3n^2/4$	제자리정렬
거품정렬	$T(n) = n^2/2$	$W(n) = 3n^2/2$ $A(n) = 3n^2/4$	제자리정렬

- 삽입정렬은 어느 정도 정렬된 데이터에 대해서는 빠르게 수행된다.
- 삽입정렬은 교환정렬 보다는 항상 최소한 빠르게 수행된다고 할 수 있다.
- 선택정렬이 교환정렬 보다 빠른가? - 일반적으로는 선택정렬 알고리즘이 빠르다고 할 수 있다.
- 입력이 이미 정렬되어 있는 경우, 선택정렬은 지정이 이루어지지만(자신의 위치에서) 교환정렬은 지정이 이루어지지 않으므로 교환정렬이 빠르다.
- 선택정렬 알고리즘이 삽입정렬 알고리즘 보다 빠른가? - n의 크기가 크고, 키의 크기가 큰 자료구조 일 때는 지정하는 시간이 많이 걸리므로 선택정렬 알고리즘이 더 빠르다.

한번 비교하는데 최대한 하나의 역을 제거하는 알고리즘의 하한선

- n개의 키, 양의 정수 1, 2,…, n 가정
- n개의 양수는 $n!$개의 순열(permutation)이 존재. 즉, $n!$가지의 순서 존재.
- k_i를 i번째 자리에 위치한 정수라고 할 때, 하나의 순열은 $[k_1, k_2,\ldots, k_n]$으로 나타낼 수 있다.   (예) [3, 1, 2]는 $k_1 = 3$, $k_2 = 1$, $k_3 = 2$로 표시.
- $i < j$와 $k_i > k_j$의 조건을 만족하는 쌍(pair) (k_i, k_j)를 순열에 존재하는 역(inversion)이라고 한다.

(예) 순열 [3, 2, 4, 1, 6, 5]에는 5개의 역이 존재

역={ (3,2), (3,1), (2,1), (4,1), (6,5) }

- **정리 7.1:**

키를 비교만 하여 n개의 서로 다른 키를 정렬하고, 한 번 비교한 후에 최대한 하나의 역만을 제거하는 알고리즘은 최악의 경우에 최소한

$$\frac{n(n-1)}{2}$$

횟수만큼의 비교를 수행하며,

평균적으로 최소한

$$\frac{n(n-1)}{4}$$

의 비교를 수행해야 한다.

증명:

– **경우 1**: (최악의 경우)

순열 $[n, n\text{-}1,\ldots, 2, 1]$은 $n(n\text{-}1)/2$개의 역을 가진다. 알고리즘이 한 번의 비교를 통해 하나의 역을 제거하므로, 총 비교 횟수는 $n(n\text{-}1)/2$.

– **경우 2**: (평균적으로)

임의의 순열 $P = [k_1, k_2,\ldots, k_n]$에 대해, P의 전치순열(transpose) $P^T = [k_n,\ldots, k_2, k_1]$. 쌍(pair) (s, r) $(s>r)$은 반드시 P에 속하거나, 아니면 P^T에 속하게 된다. 가능한 쌍은 총 $n(n\text{-}1)/2$개 이므로, P와 P^T에는 정확하게 총 $n(n\text{-}1)/2$개의 역이 존재. 따라서 P와 P^T에 존재하는 역의 평균 개수는 $n(n\text{-}1)/4$이 된다. 따라서 그만큼의 비교를 수행해야 한다.

(예) P=[3 4 1 2] 에는 역이 4개 존재.

P^T=[2 1 4 3] 에는 역이 2개 존재. 4+2=6=4x3/2

즉, 가능한 하나의 역은 P 또는 P^T 에 존재한다.

- 교환, 삽입, 선택, 버블 정렬은 한번 비교할 때 기껏해야 하나의 역만을 제거할 수 있으므로 시간복잡도가 최악의 경우 $n(n-1)/2$, 평균적으로는 $n(n-1)/4$ 보다 좋을 수 없다.

(예) 4 3 2 1. 역의 개수 = 6

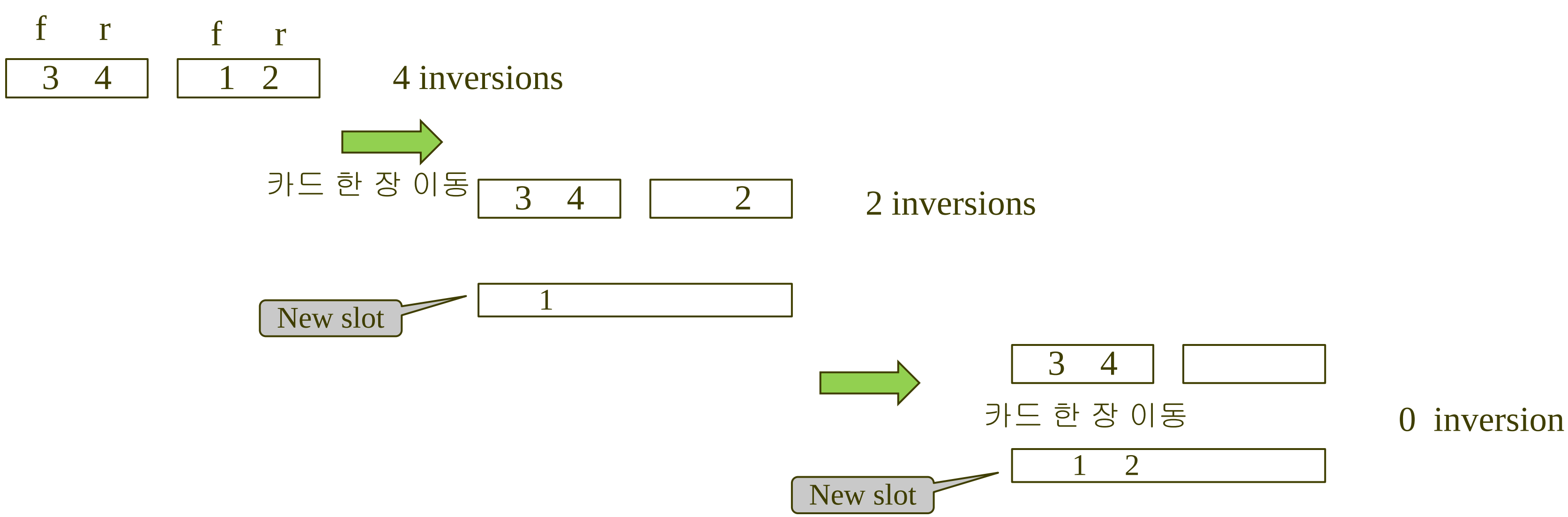
방법	한 번 비교 후	역의 개수
insertion sort	3 4 2 1	5
selection sort*	3 4 2 1	5
exchange sort	3 4 2 1	5
bubble sort	3 4 2 1	5

* 한 번 비교 후에 데이터는 실제적으로 이동하지 않으나, 내용적으로는 이동이 있는 것과 같으며, 역이 한 개 제거된 상태가 내부적으로 적용된다.

합병정렬 알고리즘 재검토

- 합병정렬은 비교마다 하나 이상의 역을 제거하므로 앞서 살펴본 교환, 삽입, 선택 정렬보다 효율적이다.
  - ✓ 예) [3,4], [1,2]를 합병할 때 3과 1를 비교하면 1이 작으므로 1이 결과 배열의 첫 슬롯에 들어간다. 이를 통해 (3, 1), (4, 1) 두 개의 역을 제거한다.
  - ✓ 3과 2가비교된 후 2가 결과배열에 들어가면서 역 (3,2), (4,2)가 제거된다.

- 합병



Quicksort

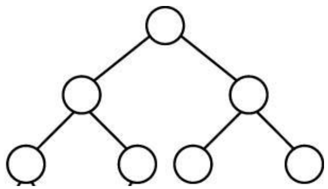
- 문제: *n*개의 정수를 비내림차순으로 정렬
- 입력: 정수 $n > 0$, 크기가 *n*인 배열 S[1..*n*]
- 출력: 비내림차순으로 정렬된 배열 S[1..*n*]

```
void quicksort (index low, index high) {
     index pivotpoint;
     if (high > low) {
          partition(low,high,pivotpoint);
          quicksort(low,pivotpoint-1);
          quicksort(pivotpoint+1,high);
     }
}
```

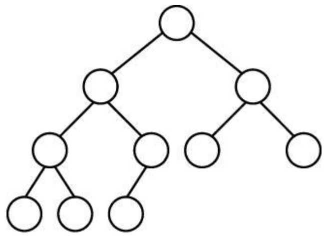
추가공간: 평균적으로 $\Theta(\lg n)$
- 재귀에 의한 인덱스 저장공간

binary tree의 종류

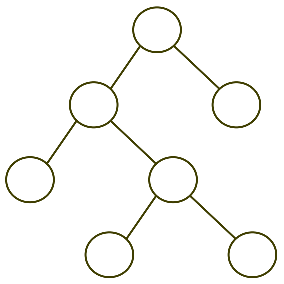
- 완전이진트리(complete(perfect) binary tree): 트리의 내부에 있는 모든 마디에 두 개씩 자식마디가 있는 이진 트리. 따라서 모든 잎의 깊이(depth) *d*는 동일하다.



- 실질적인 완전이진트리(essentially complete binary tree)
  - ✓ 깊이 *d* - 1까지는 완전이진트리이고,
  - ✓ 깊이 *d*의 마디는 왼쪽 끝에서부터 채워진 이진트리.

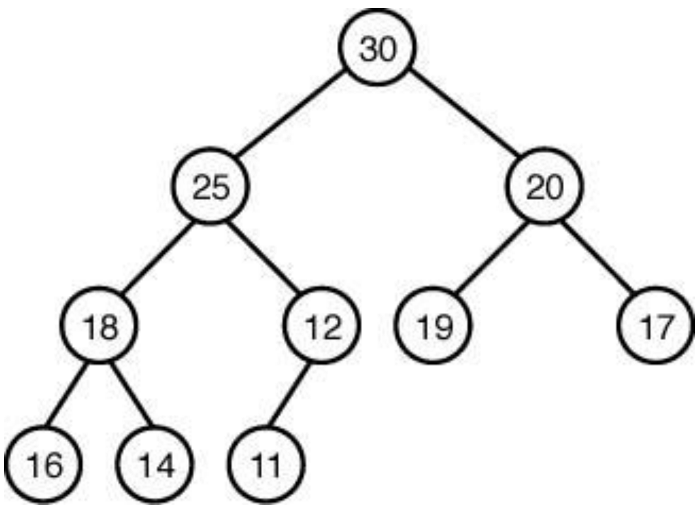


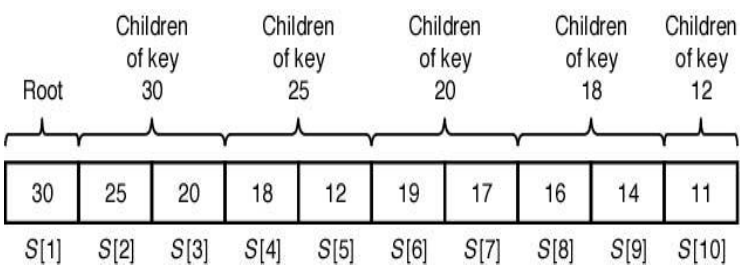
- full binary tree (**proper binary tree** or **2-tree**)는 모든 노드가 영 또는 2개의 자식노드를 갖는다.



힙(heap)

- 힙의 성질(heap property): 어떤 마디에 저장된 값은 그 마디의 자식마디에 저장된 값보다 크거나 같다. – max heap
- 힙(heap): 힙의 성질을 만족하는 실질적인 완전이진트리





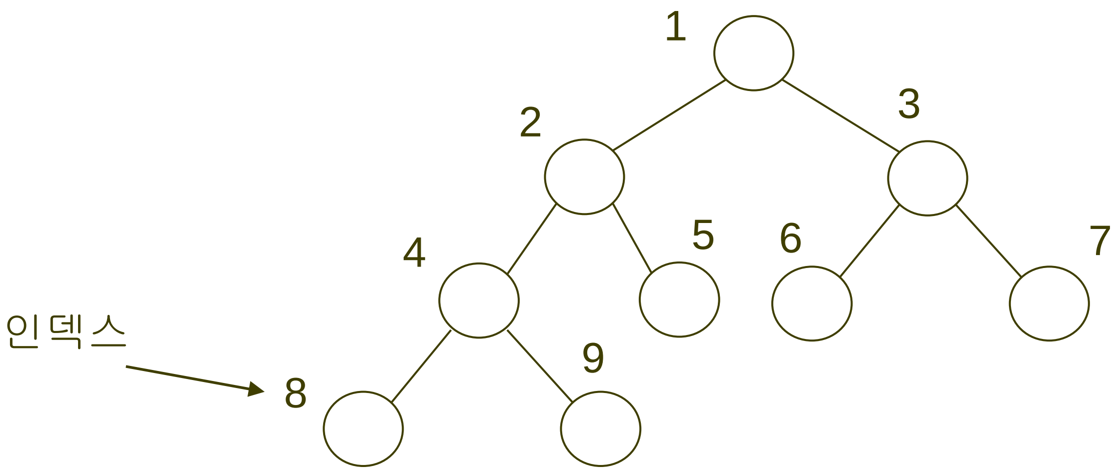
힙의 자료구조(배열)

- 힙 구조의 특성
  1. 최대값의 확인 – O(1)
  2. 최대값제거 및 재구성 – O(lg n)
  3. 데이터의 추가, 삭제, 변경 - O(lg n)

- 최대값을 항상 유지해야 하는 Queue를 구현하는데 적합 – priority queue

- 힙 구조의 해석

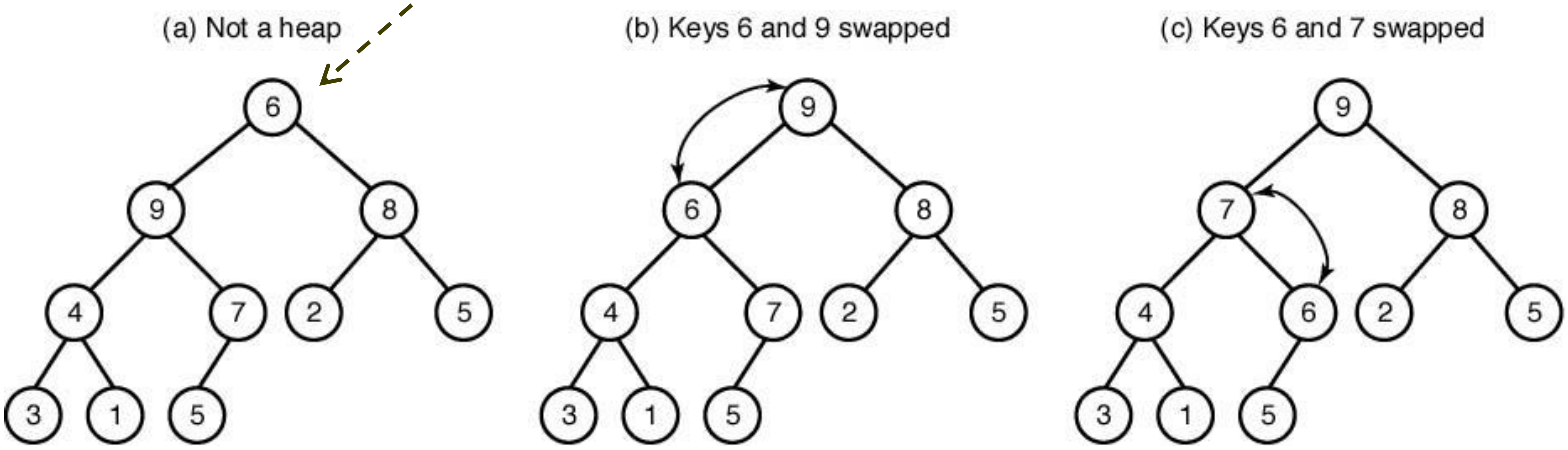
  - ✓ index i 노드의
    - ❖ left child index = $2 \times i$
    - ❖ right child index = $2 \times i + 1$
    - ❖ 부모 노드 인덱스 = $\lfloor n/2 \rfloor$



Siftdown

sift: 채로 치다

- 힙 성질을 만족하도록 재구성 방법
  - ✓ 루트에 있는 키가 힙성질을 만족하지 않음.



✓ 교체하는 child node를 결정하기 위해 2회의 비교 필요

- 힙성질을 만족하도록 조정

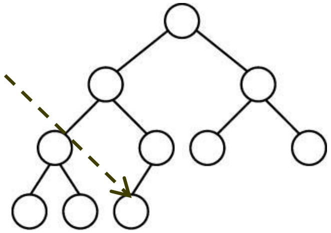
```
void siftdown(heap& H){
        node parent, largerchild;

        parent = root of H;
        largerchild = parent's child containing larger key;

        while(key at parent is smaller than key at largerchild){
            exchange key at parent and key at largerchild;
            parent = largerchild;
            largerchild = parent's child containing larger key;
          }
}
```

- 루트에서 키를 추출하고 힙 성질을 회복하는 의사코드

```
keytype root(heap& H){
      keytype keyout;
         keyout = key at the root;
         move the key at the bottom node to the root;
         delete the bottom node;
         siftdown(H);
         return keyout;
      }
```



힙정렬

- 힙 정렬 아이디어
  1. n개의 키를 이용하여 힙을 구성한다.
  2. 루트에 있는 제일 큰 값을 제거한다. → 힙 재구성
  3. step 2를 n−1번 반복한다.

힙정렬

```
void removekeys(int n, heap H, keytype S[]){
   index i;
   for(i=n; i>=1; i--)
     S[i] = root(H);
}
```

```
void makeheap(int n, heap& H){
   index i;
   heap Hsub;
   for(i=d-1; i>=0; i--)
     for(all subtree Hsub whose roots have depth i)
       siftdown(Hsub);
}
```

d=H의 높이, i는 depth의 index

```
void heapsort(int n, heap H, keytype S[]){
   makeheap(n,H);
   removekeys(n,H,S);
}
```

heap 정렬

```
struct heap{
  keytype S[1..n];
  int heapsize;};

void siftdown(heap& H, index i){
  index parent, largerchild;
  keytype siftkey;
  bool spotfound;

  siftkey = H.S[i];
  parent = i;
  spotfound = false;
  while( 2*parent ≤ H.heapsize && !spotfound){
    if(2*parent < H.heapsize &&
          H.S[2*parent] < H.S[2*parent+1])
       largerchild = 2*parent + 1;
    else
       largerchild = 2*parent;
    if(siftkey < H.S[largerchild]){
       H.S[parent] = H.S[largerchild];
       parent = largerchild;
    }
    else
       spotfound = true;
  }
  H.S[parent] =siftkey;
}
```

```
keytype root(heap& H) {
  keytype keyout;
  keyout = H.S[1];
  H.S[1] = H.S[heapsize];
  H.heapsize = H.heapsize -1;
  siftdown(H,1);
  return keyout;
}
```

```
void removekeys(int n, heap& H, keytype S[]){
  index i;
  for (i=n; i≥1; i--)
     S[i] = root(H);
}
```

```
void makeheap(int n, heap& H){
  index i;

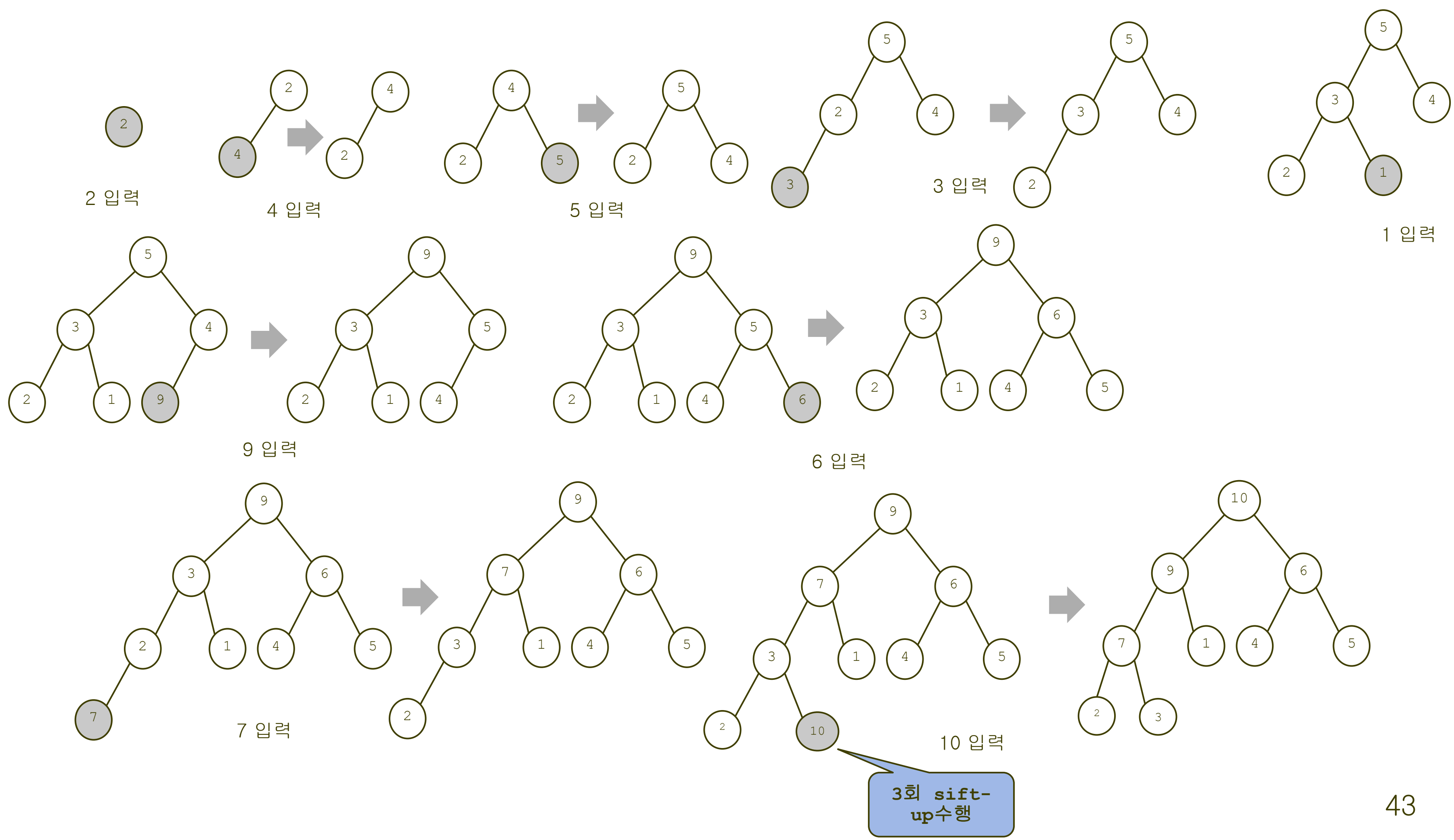
  H.heapsize=n;
  for(i=└n/2┘; i ≥ 1, i--)
     siftdown(H,i);
}
```

i는 노드번호의 index

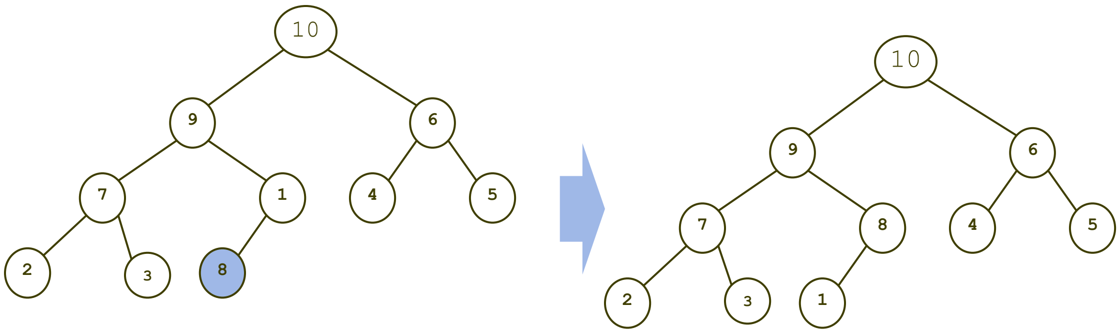
- make heap 방법
    - 방법1: 데이터가 입력되는 순서대로 heap을 매번 구성
    - 방법2: 모든 데이터를 트리에 넣은 상태에서 heap 구성

데이터 : 2 4 5 3 1 9 6 7 10 8

(방법1) sift-up 수행, 데이터가 입력되는 순서대로 heap을 매번 구성



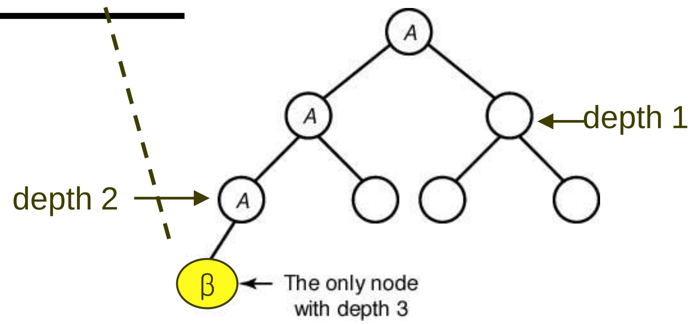
데이터 : 2 4 5 3 1 9 6 7 10 8



8 입력

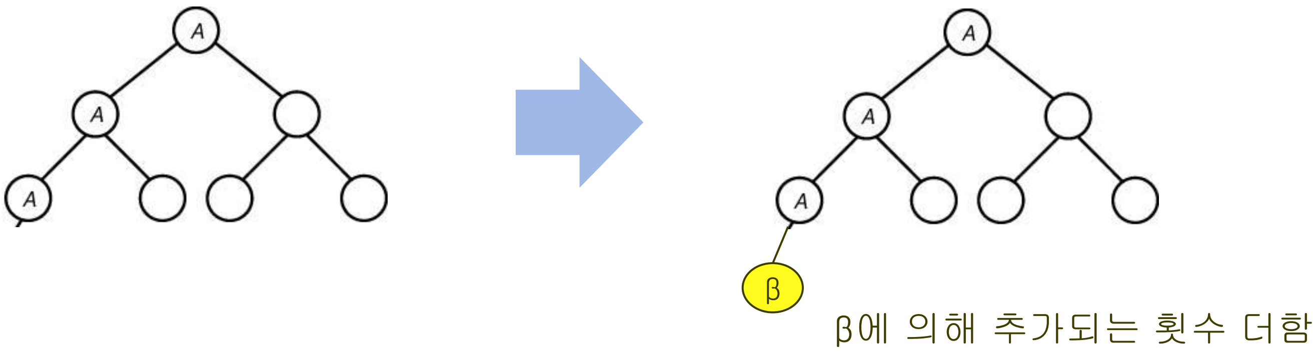
- **makeheap 방법(1)의 최악의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  - ✓ 단위연산: sift-up 프로시저에서의 키의 비교
  - ✓ 입력크기: n, 총 키의 개수. $n = 2^k$라 가정
  - ✓ d를 트리의 깊이라고 하면, $d = \lg n$. 이때 d의 깊이를 가진 마디는 정확히 하나이고 그 마디는 d개의 조상(ancestor)을 가진다. 일단 깊이가 d인 그 마디가 없다고 가정하고 키가 sift-up되는 상한값(upper bound)을 구함.

depth	node수	키가 sift-up되는 최대횟수
0	2^0	0
1	2^1	1
2	2^2	2
⋮	⋮	⋮
j	2^j	j
⋮	⋮	⋮
$d-2$	2^{d-2}	$d-2$
$d-1$	2^{d-1}	$d-1$



$$\text{총} \quad \sum_{j=0}^{d-1} 2^j j$$

- 일단 β노드가 없는 것으로 가정해서 분석한 후 β노드에 의해 추가적으로 발생하는 sift-up 횟수를 더한다.



- 한 번의 sift-up에서는 1번의 키 비교가 필요하다.

$$총\ sift-up\ 횟수\ S = \sum_{j=0}^{d-1} j2^j$$

$$= 1\times 2^1 + 2\times 2^2 + \ldots + j\times 2^j + \ldots + (d-2)\times 2^{d-2} + (d-1)\times 2^{d-1} \qquad (1)$$

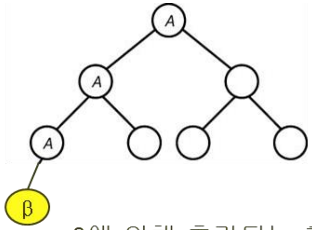
$$2S = \qquad 1\times 2^2 + 2\times 2^3 + \ldots + (d-3)\times 2^{d-2} + (d-2)\times 2^{d-1} + (d-1)\times 2^d \qquad (2)$$

$$(2) - (1) = S = (d-1)\times 2^d - (2^1 + 2^2 + 2^3 + \cdots + 2^{d-1})$$

$$= (\lg n - 1)n - \frac{2(2^{d-1}-1)}{2-1}$$

$$= n\lg n - n - 2^d + 2$$

$$= n\lg n - 2n + 2$$



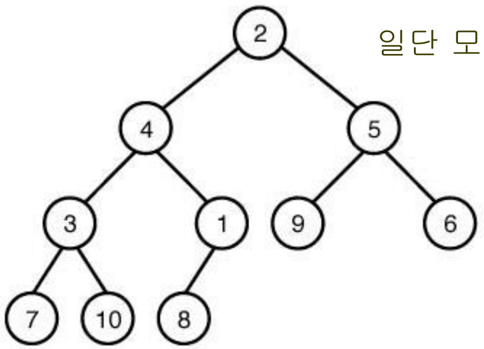
β에 의해 추가되는 횟수 더함

- depth가 d인 노드에 의한 추가 sift-up 횟수는 $d=\lg n$ 이므로 총 횟수는 $(n+1)\lg n - 2n+2$
- sift-up 1회당 1회의 비교. 그러므로 비교횟수는 $(n+1)\lg n - 2n+2$
- 즉 $O(n \lg n)$ 시간이 필요함

makeheap (방법2), 모든 데이터를 트리에 넣은 상태에서 heap 구성

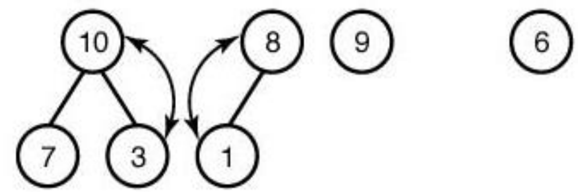
데이터 : 2 4 5 3 1 9 6 7 10 8

(a) The initial structure

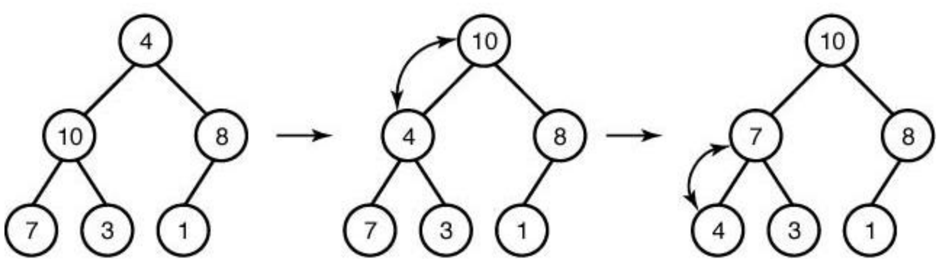


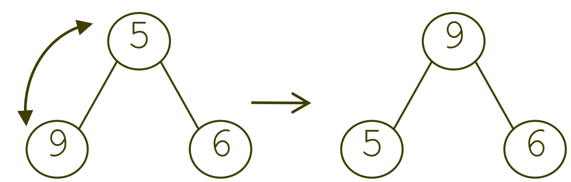
일단 모두 넣은 상태

(b) The subtrees, whose roots have depth $d-1$, are made into heaps.

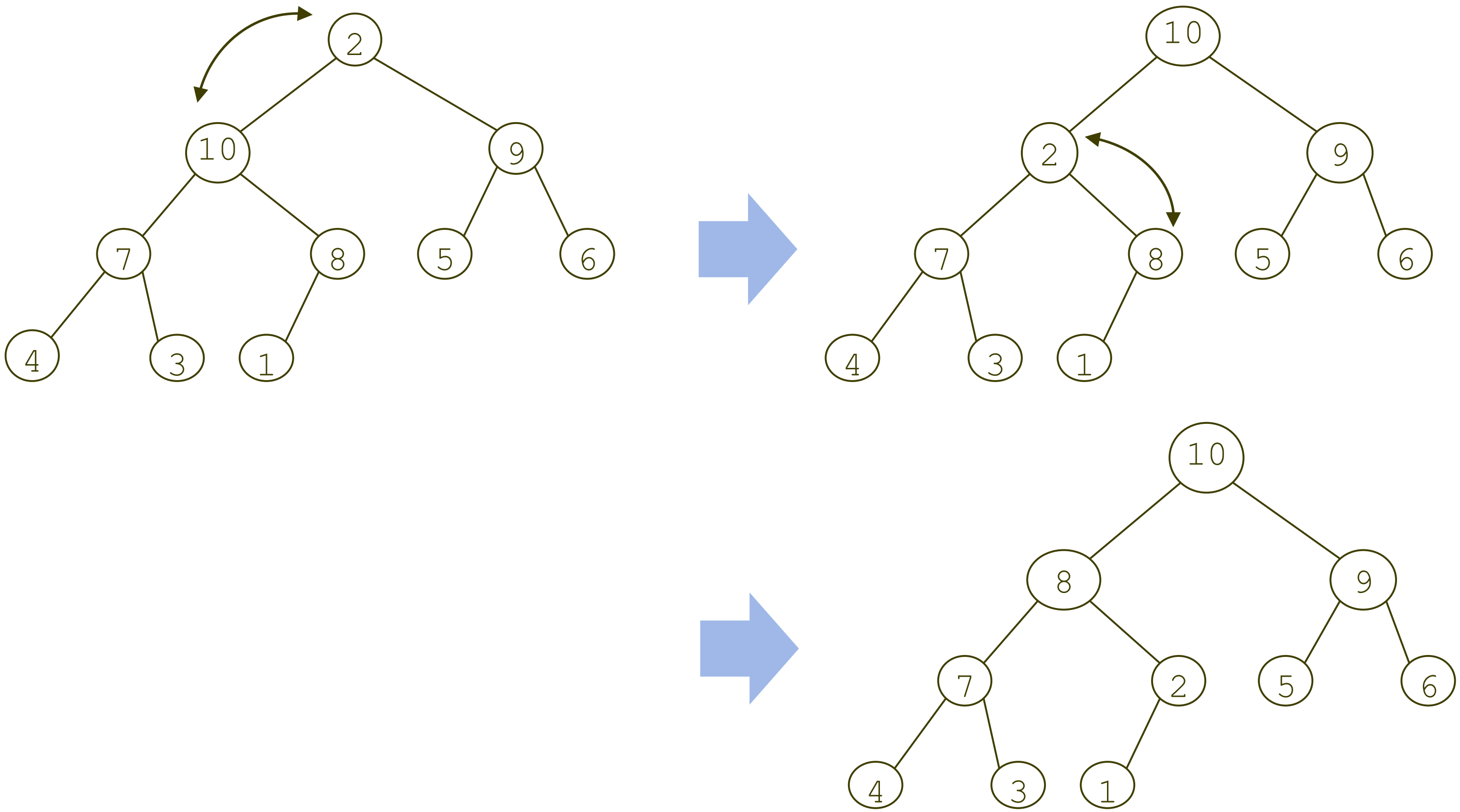


(c) The left subtree, whose root has depth $d-2$, are made into a heap.

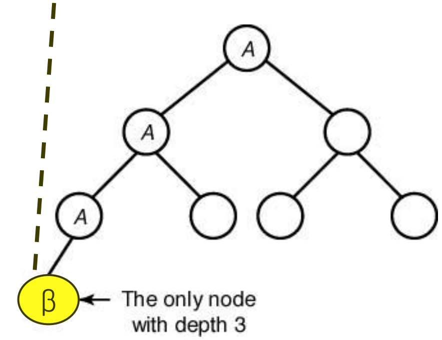




(d) depth가 d-3 인 노드의 siftdown



- **makeheap 방법(2)의 최악의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  - ✓ 단위연산: sift-down 프로시저에서의 키의 비교
  - ✓ 입력크기: n, 총 키의 개수. $n = 2^k$라 가정
  - ✓ d를 실질적인 완전이진트리의 깊이라고 하면, $d = \lg n$. 이때 <u>d의 깊이를 가진 마디는</u> 정확히 하나이고, 그 마디는 d개의 조상(ancestor)을 가진다. 일단 깊이가 d인 그 마디가 없다고 가정하고 키가 sift되는 상한값(upper bound)을 구해 보자.



depth	node수	키가 sift-down되는 최대횟수
0	2^0	$d-1$
1	2^1	$d-2$
2	2^2	$d-3$
$\vdots$	$\vdots$	$\vdots$
j	2^j	$d-j-1$
$\vdots$	$\vdots$	$\vdots$
$d-2$	2^{d-2}	1
$d-1$	2^{d-1}	0

$$\text{총} \quad \sum_{j=0}^{d-1} 2^j (d-j-1)$$

$$\text{총 } sift-down \text{ 횟수} = \sum_{j=0}^{d-1} 2^j (d-j-1) = (d-1)\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j2^j$$

$$\sum_{j=0}^{d-1} 2^j = \frac{2^d - 1}{2-1} = 2^d - 1 = n-1$$

$S = \sum_{j=0}^{d-1} j2^j$를 계산하기 위해

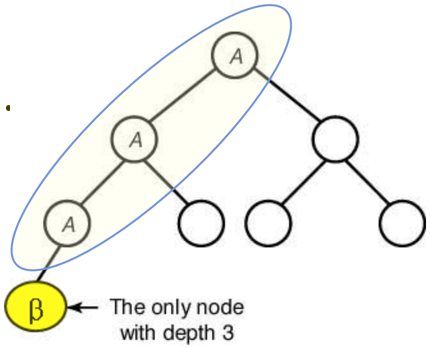
$$\mathrm{S} = 1\times 2^1 + 2\times 2^2 + \cdots + j\times 2^j + \cdots + (d-2)\times 2^{d-2} + (d-1)\times 2^{d-1} \quad (1)$$

$$2S = 1\times 2^2 + 2\times 2^3 + \cdots + (d-3)\times 2^{d-2} + (d-2)\times 2^{d-1} + (d-1)\times 2^d \quad (2)$$

$$\begin{aligned}(2)-(1) = \mathrm{S} &= (d-1)\times 2^d - (2^1 + 2^2 + 2^3 + \cdots + 2^{d-1}) \\ &= (\lg n - 1)n - \frac{2(2^{d-1}-1)}{2-1} \\ &= n\lg n - n - 2^d + 2 \\ &= n\lg n - 2n + 2\end{aligned}$$

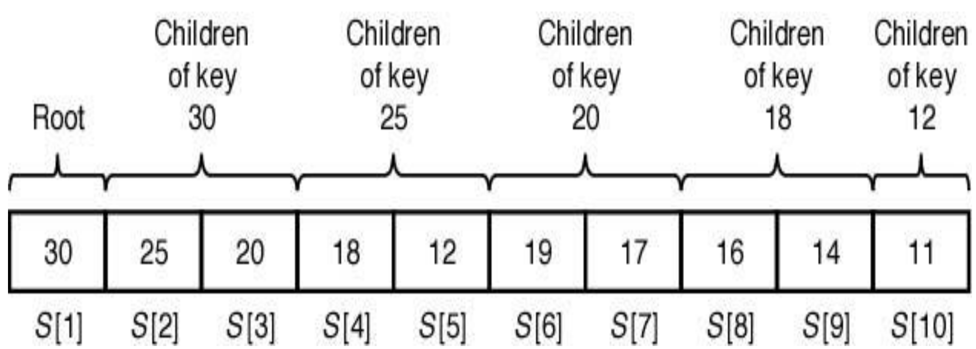
$$
\begin{aligned}
총\ siftdown\ 횟수 &= (d-1)(n-1) - (n\lg n - 2n + 2) \\
&= (\lg n - 1)(n-1) - n\lg n + 2n - 2 \\
&= n\lg n - n - \lg n + 1 - n\lg n + 2n - 2 \\
&= n - \lg n - 1
\end{aligned}
$$

- depth가 d인 노드(β노드)에 의한 추가 sift-down 횟수는 d=$lg\ n$ 이므로 총 횟수는 $(n-1)$:
  [이유]β노드의 ancestor들(d개)이 한번씩 sift-down이 추가로 발생할 수 있음.
- 한 번의 sift-down에서는 2번의 키 비교가 필요하다.
- 비교 횟수는 $2(n-1)$
- 즉 $O(n)$ 시간이 필요함



힙정렬 알고리즘의 공간복잡도

- 이 알고리즘이 제자리정렬 알고리즘인가?
  - ✓ 힙을 배열로 구현한 경우에는 제자리정렬 알고리즘
  - ✓ 공간복잡도: $\Theta(1)$



힙정렬 알고리즘 시간복잡도

- **알고리즘:**

```
void heapsort(int n, heap& H){
   makeheap(n,H);
   removekeys(n,H,H.S);
}
```

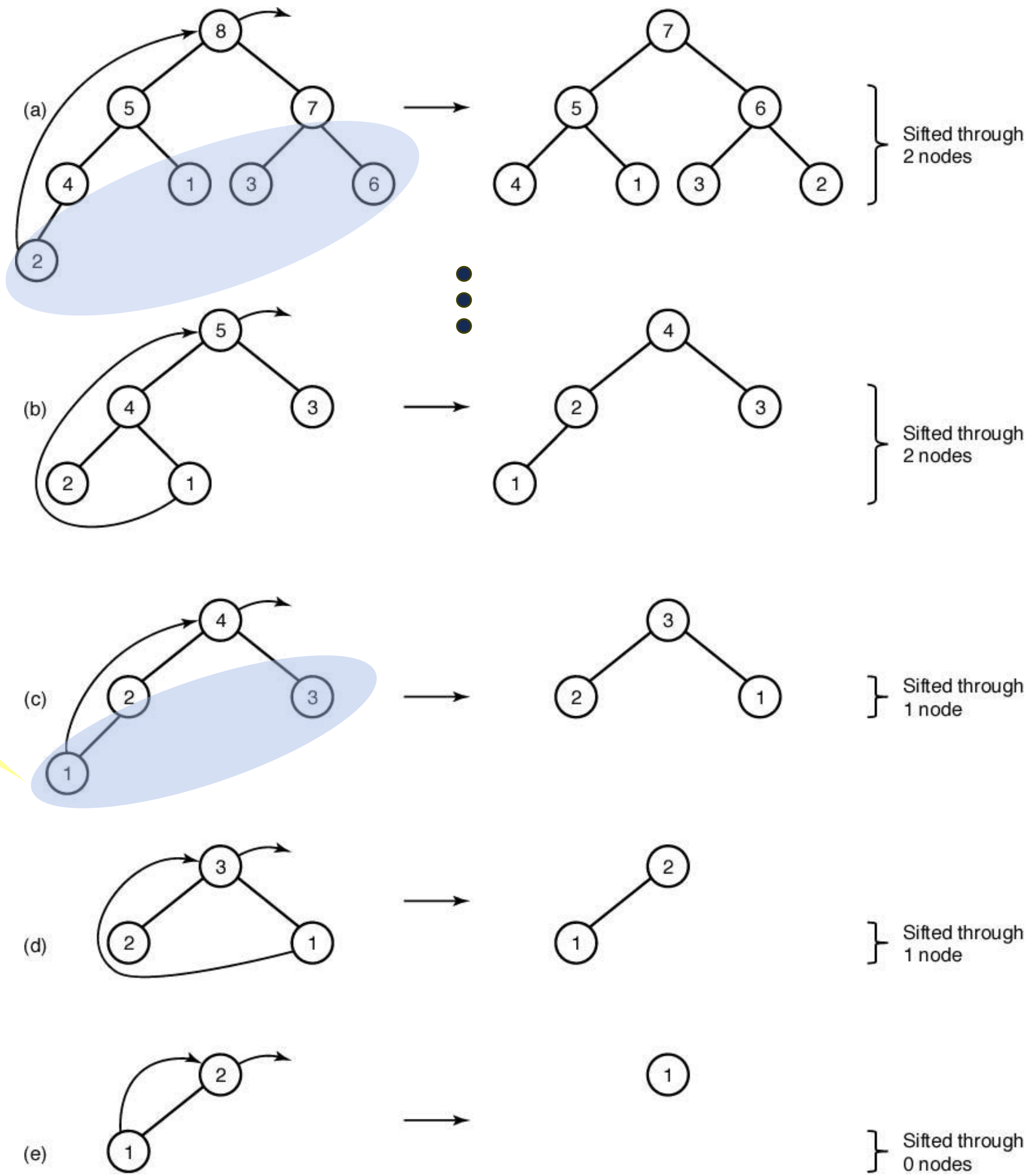
$2(n\text{-}1)$

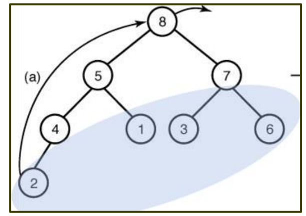
$2n \lg n - 4n + 4$

Removekeys

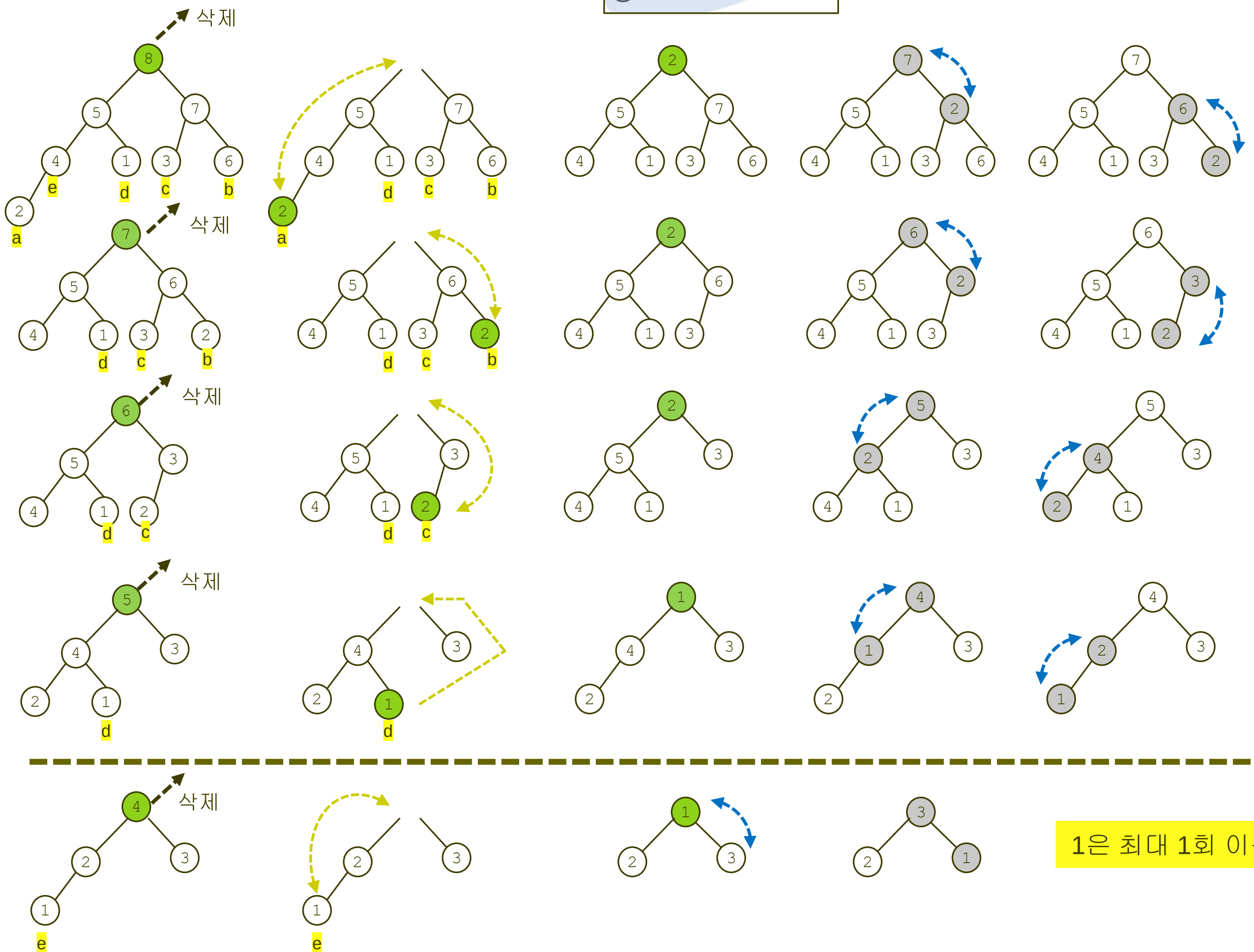
4개의 키에 대해서는 2회의 sift-down 가능

2개의 키에 대해서는 1회의 sift-down 가능





의 설명



2회 이동

2회 이동

2회 이동

2회 이동

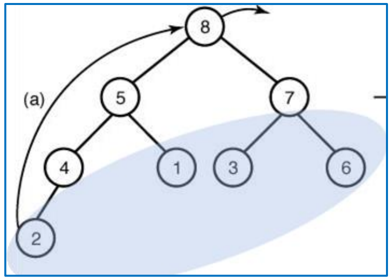
1은 최대 1회 이동 가능

✓ removekeys의 분석: $n = 2^k$라 가정.

먼저 $n = 8$이고 $d = \lg 8 = 3$인 경우, 처음 4개의 키를 제거하는데 sift되는 횟수가 2회, 다음 2개의 키를 제거하는데 sift되는 횟수가 1회, 그리고 마지막 2개의 키를 제거 하는 데는 sift되지 않았다. 따라서 총 sift횟수는 $1(2) + 2(4) = \sum_{j=1}^{3-1} j2^j$ 가 된다. 따라서 일반적인 경우는

$$\sum_{j=1}^{d-1} j2^j = (d-2)2^d + 2 = d \cdot 2^d - 2 \cdot 2^d + 2 = n \lg n - 2n + 2$$

가 된다. 그런데 한번 sift-down될 때 마다 2번씩 비교하므로 실제 비교횟수는 $2n \lg n - 4n + 4$ 이 된다.



✓ makeheap 과 removekey의 통합:

키를 비교하는 총 횟수는 n이 2^k일 때

$2(n\text{-}1) + 2n \lg n - 4n + 4 = 2(n \lg n - 2n + 1) \approx 2n \lg n$ 을 넘지 않는다.

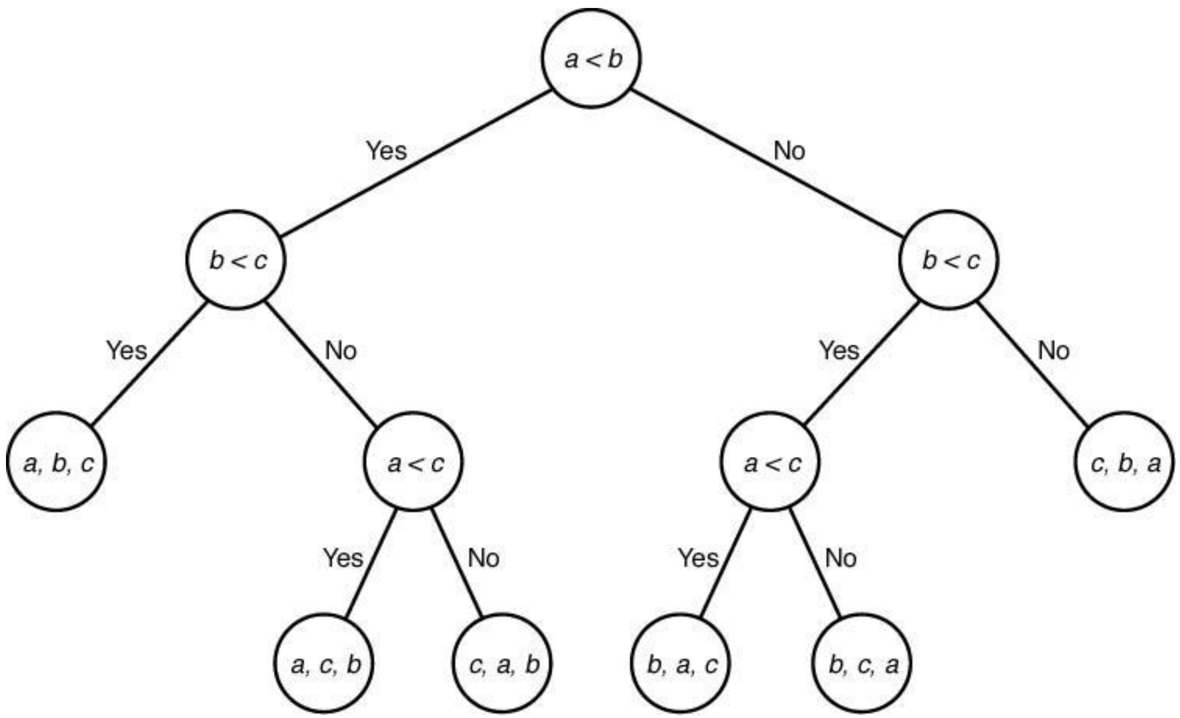
따라서 최악의 경우 $W(n) \in \Theta(2n \lg n)$

$\Theta(n \lg n)$ 알고리즘의 비교

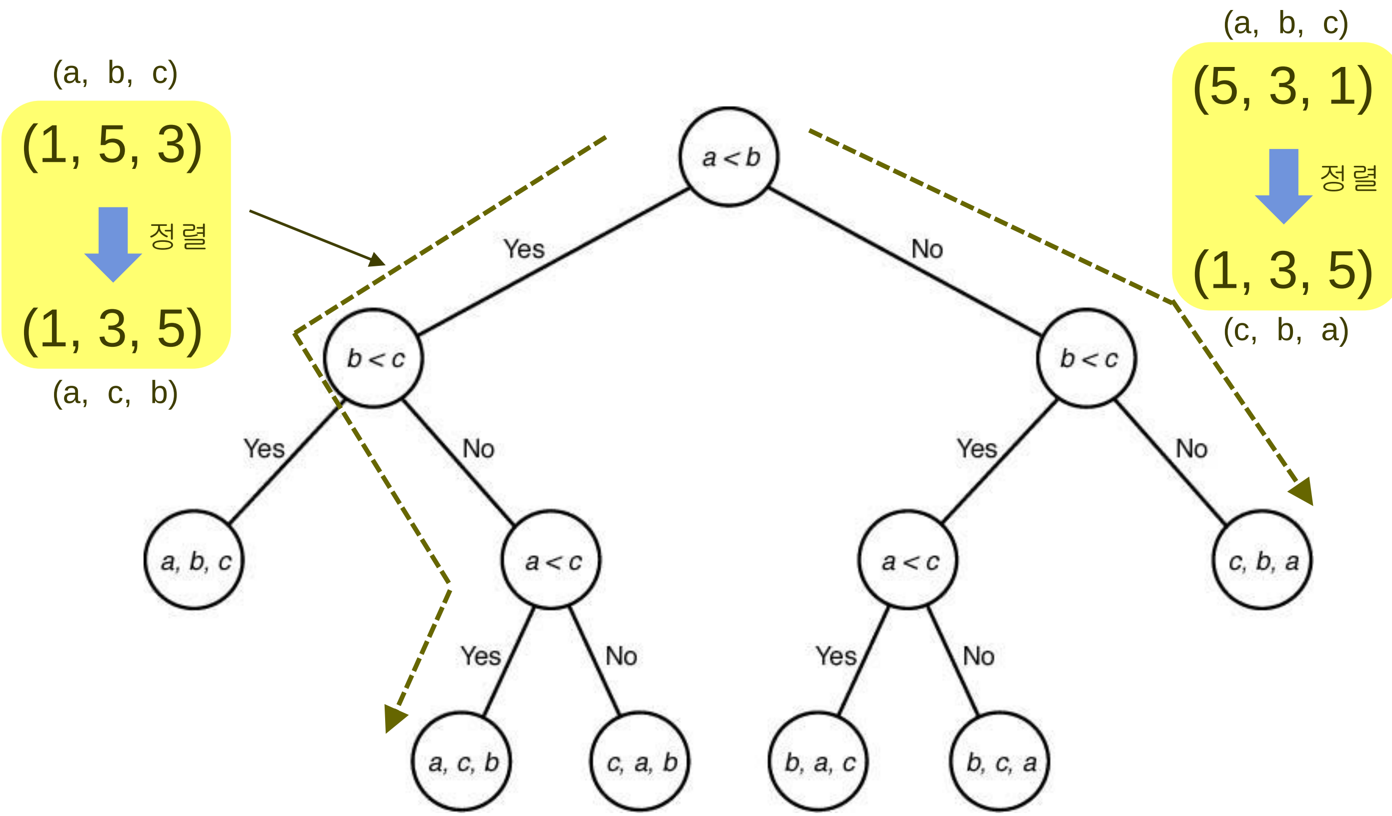
알고리즘	비교횟수	지정횟수	추가저장장소사용량
합병정렬	$W(n) = A(n) = n \lg n$	$T(n) = 2n \lg n$	$\Theta(n)$
빠른정렬	$W(n) = n^2/2$ $A(n) = 1.38n \lg n$	$A(n) = 0.69n \lg n$	$\Theta(\lg n)$ (재귀에 의한 공간)
힙정렬	$W(n) = A(n) = 2n \lg n$	$W(n) = A(n) = n \lg n$	제자리정렬

키의 비교만으로 정렬하는 경우 하한

- $n \lg n$ 보다 더 빠른 정렬 알고리즘을 개발할 수 있을까?
  - ✓ 키의 비교 횟수를 기준으로 하는 한, 더 빠른 알고리즘은 불가능.
- 정렬알고리즘에 대한 결정트리
  - ✓ 3개의 키 a,b,c 를 정렬하는 알고리즘의 결정트리(decision tree).



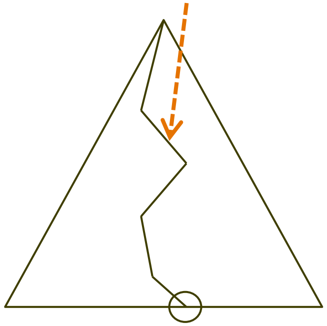
- decision tree: root에서 출발하여 노드의 조건을 따라가며 말단 노드에 도달
- 말단 노드는 하나의 결정(decision)을 나타냄. 여기서는 정렬된 상태



3개의 키 a,b,c 를 정렬하는 알고리즘의 결정트리(decision tree).

✓ n개의 키 정렬 문제의 결정트리

❖ 만약 n개의 키의 각 순열(permutation)에 대해서, 뿌리마디로부터 잎마디로 이르는 경로가 있는 경우, 결정트리는 유효하다(valid). 즉, 크기가 n인 어떤 입력에 대해서도 정렬할 수 있다.



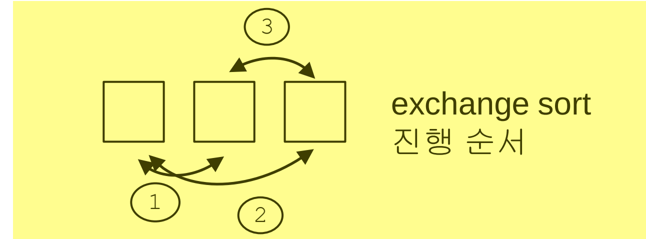
```
void  exchangesort(int  n, keytype  S[ ]) {
     index  i,j;

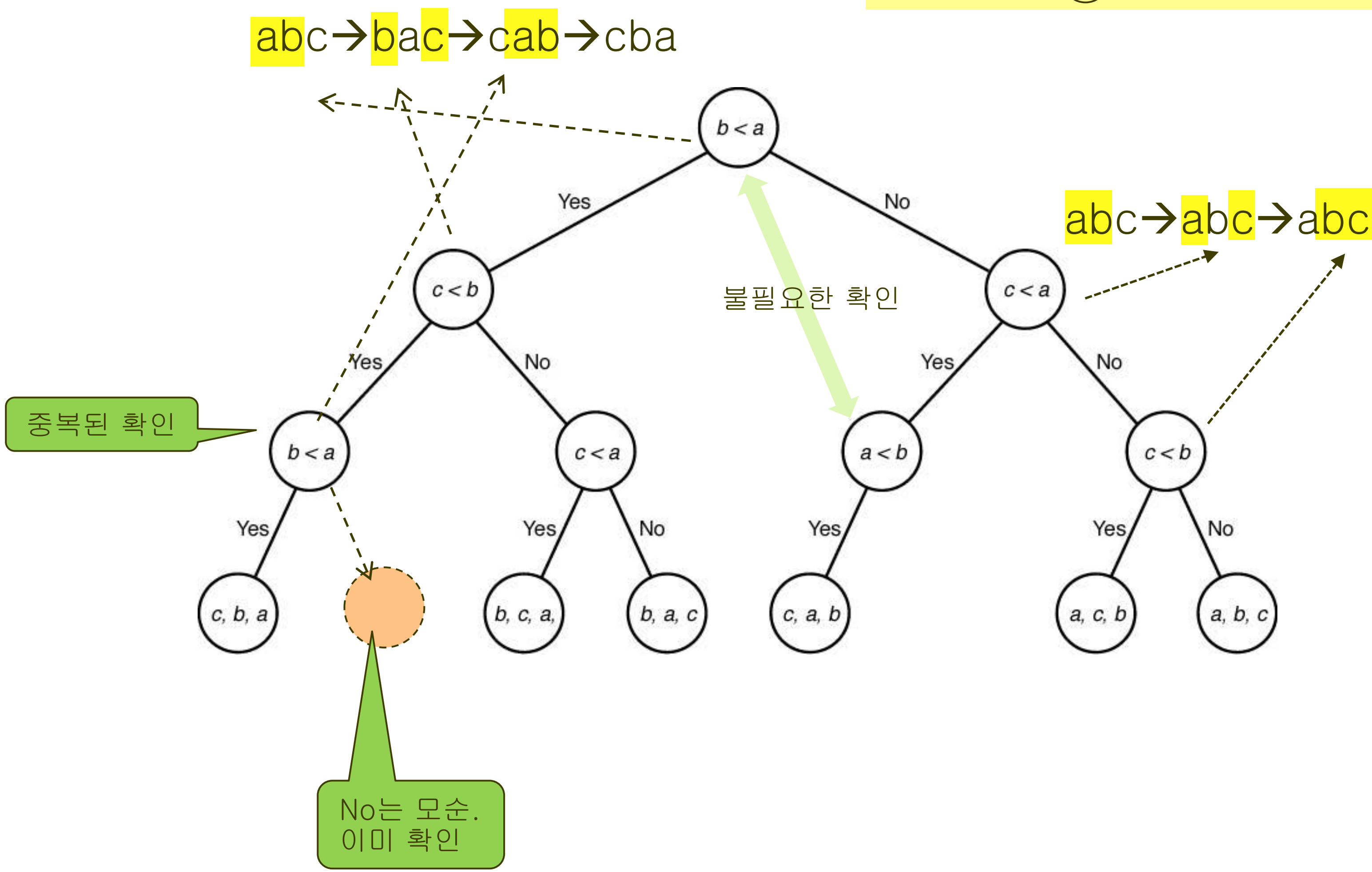
     for  (i=1; i<=n-1; i++)
        for  (j=i+1; j<=n; j++)
           if(S[j] < S[i])
              exchange S[i] and S[j]
}
```

- 3개 입력의 교환정렬 알고리즘의 결정트리에서는 불필요한 비교를 하고 있다.
  - 어떤 시점에서 비교가 이루어 질 때, 그 이전에 이루어졌던 비교의 결과를 전혀 알 수 없기 때문. → 최적(optimal)이 아닌 알고리즘에서 나타남.
- 가지친 결정트리(pruned decision tree): 일관성 있는 순서로 결정을 내림으로서 뿌리마디로부터 모든 잎마디에 도달할 수 있는 경우, 다음 화면의 결정트리는 가지친(pruned) 결정트리이다.

```
void  exchangesort(int  n, keytype  S[ ]) {
      index  i,j;
      for (i=1; i<=n-1; i++)
         for (j=i+1; j<=n; j++)
            if(S[j] < S[i])
               exchange S[i] and S[j
```

입력데이타: s[1]=a, s[2]=b, s[3]=c

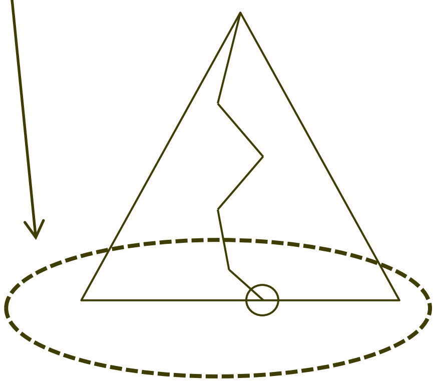




3개 키의 교환정렬에 해당하는 가지친 결정트리

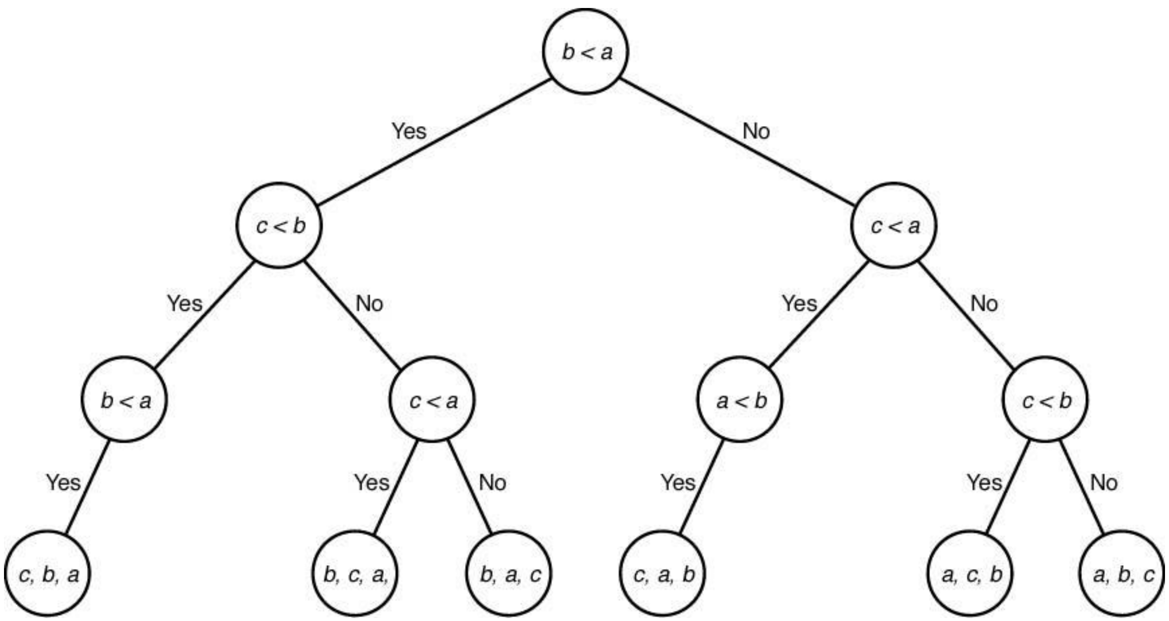
✓ **보조정리 7.1**:

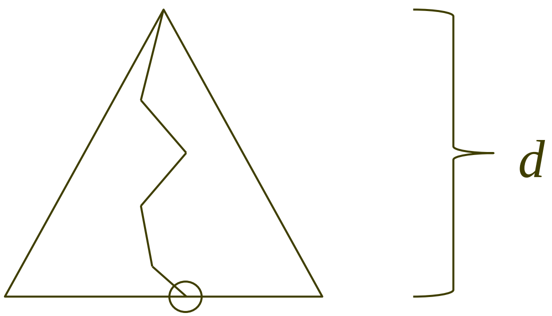
*n*개의 서로 다른 키를 정렬하는 결정적(deterministic)알고리즘은, 그에 상응하는 정확하게 *n*!개의 잎마디를 가진, 유효하며 가지친 이진 결정트리가 존재한다.



Lower Bound for Worst-Case

- **Lemma 7.2**: The worst-case number of comparisons done by a decision tree is equal to its depth.





결정트리로 구한 최악의 경우 하한

- **보조정리 7.3**: 이진트리(binary tree)의 잎마디의 수가 m이고, 깊이가 d이면, $d \geq \lceil \lg m \rceil$ 이다.

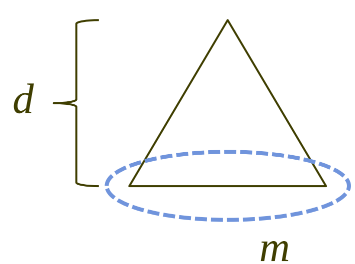
증명: d에 대하여 귀납법으로 증명. 우선 $2^d \geq m$임을 먼저 보인다.

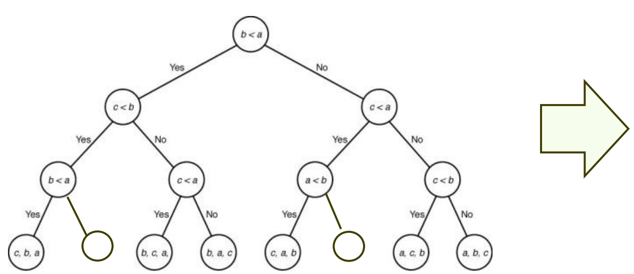
- ✓ **귀납출발점**: $d = 0$: 마디의 수가 하나인 이진트리. 따라서 명백히 $2^d \geq 1$.
- ✓ **귀납가정**: 깊이가 d인 모든 이진트리에 대하서, $2^d \geq m$가 성립한다고 가정.
- ✓ **귀납절차**: 깊이가 $d + 1$인 모든 이진트리에 대해서, $2^{d+1} \geq m'$임을 보이면 된다. 여기서 m'은 잎마디의 수이다.

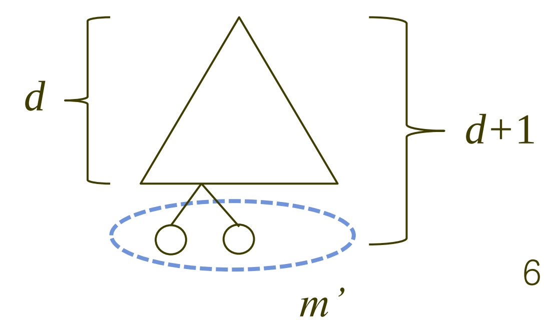
$2^{d+1} = 2 \times 2^d \geq 2m$ 귀납가정에 의해서 성립

$\geq m'$ 각 부모마디는 기껏해야 자식마디 2개를 가지므로

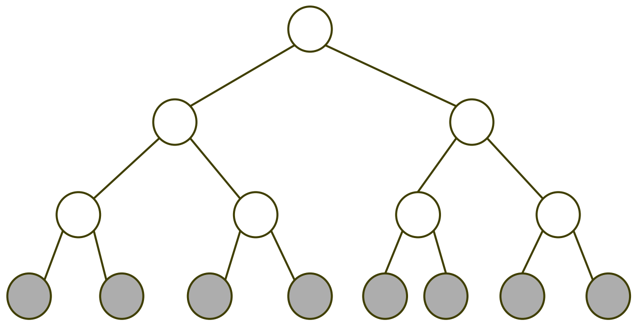
그러므로 $2^d \geq m$이 성립한다. 여기서 양변에 lg를 씌우면, $d \geq \lg m$이 된다. 그런데 d는 정수이므로, $d \geq \lceil \lg m \rceil$ 이 된다.



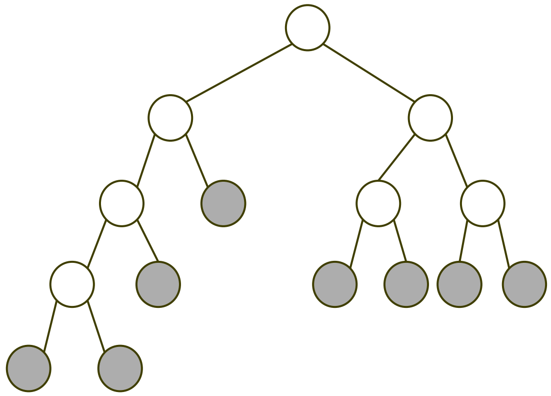




- $d \geq \lceil \lg m \rceil$
- d는 적어도 $\lceil \lg m \rceil$가 되어야 한다.



8개의 leaf 가 있는 높이가 제일 작은 이진 트리. d=3



d=4인 8개의 leaf 를 갖는 이진 트리

- **정리 7.3:** ***n***개의 서로 다른 키를 비교함으로써 만 정렬하는 결정적 알고리즘은 최악의 경우 최소한 $\lceil n \lg n - 1.45n \rceil$번의 비교를 수행한다.

✓ 증명: 보조정리 7.1에 의하면, $n!$개의 잎마디를 가진 가지친, 유효한, 이진결정트리가 존재한다. 다시 보조정리 7.3에 의하면, 그 트리의 깊이 ≥ $\lceil \lg (n!) \rceil$가 되고, 보조정리 7.2에 의해서, 결정트리의 최악의 경우의 비교횟수는 그 트리의 깊이와 같다.

- **Lemma 7.4**: For any positive integer n, $\lg(n!) \geq n \lg n - 1.45n$.

proof: The proof requires knowledge of integral calculus. We have

$$\begin{aligned}
\lg(n!) &= \lg[n(n-1)(n-2)\cdots 2] \\
&= \sum_{i=2}^{n} \lg i \\
&\geq \int_{1}^{n} \lg x \, dx \\
&= \frac{1}{\ln 2}(n \ln n - n + 1) \\
&\geq n \lg n - 1.45n
\end{aligned}$$

결론:

키 값의 비교를 통한 정렬은 $\Omega(n \lg n)$ 의 복잡도를 갖는다. 즉, $n \lg n$ 보다 더 빠른 알고리즘을 개발할 수는 없다.

- **정리 7.4:**

$\boldsymbol{n}$개의 서로 다른 키를 비교함으로써 만 정렬하는 결정적 알고리즘은 평균의 경우 최소한 $\lfloor n \lg n - 1.45n \rfloor$ 번의 비교를 수행한다.

최악의 경우 최소한 $\lceil n \lg n - 1.45n \rceil$ 번의 비교

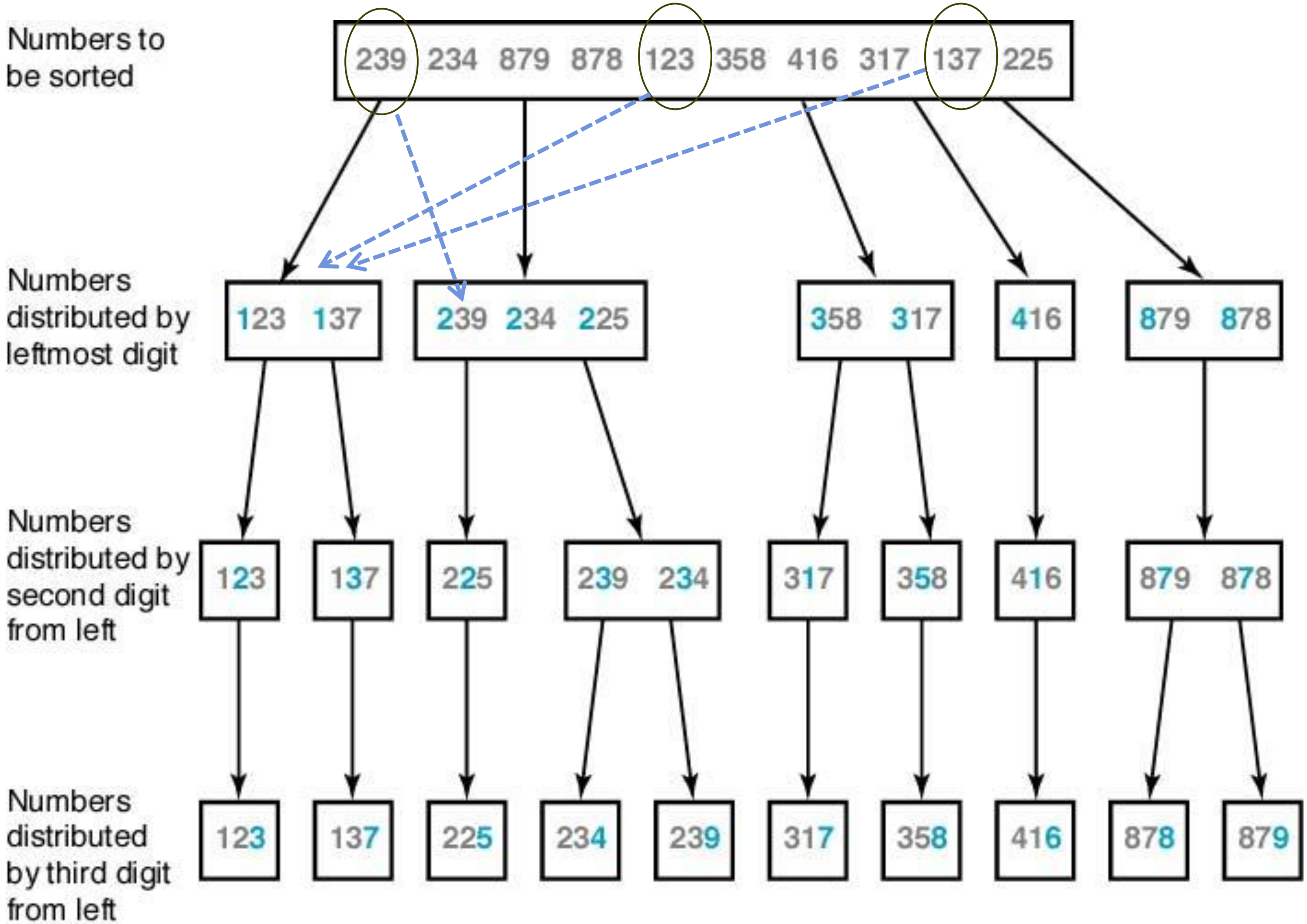
- 합병정렬의 평균의 경우 성능인 $n \lg n - 1.26n$ 은 키를 비교만 하여 정렬하는 알고리즘으로는 거의 최적임

분배에 의한 정렬: 기수정렬

- 키에 대해서 아무런 정보가 없는 경우
  - ✓ 키들을 비교하는 것 이외에는 다른 방법이 없으므로 $\Theta(n \lg n)$보다 더 좋은 알고리즘을 만드는 것은 불가능하다.

- 키에 대한 어느 정도의 정보를 알고 있는 경우
  - ✓ 디지트(digit)의 개수가 모두 같다면, 첫 번째 디지트가 같은 수끼리 따로 모으고, 그 중에서 두 번째 디지트가 같은 수끼리 따로 모으고, 마지막 디지트 까지 이런 식으로 계속 모으는 방법으로 각 디지트를 한번씩만 조사를 하면 정렬을 완료할 수 있다.
  - ✓ "분배에 의한 정렬(sorting by distribution)" - 기수정렬(radix sort)
  - ✓ 기수(radix, base)를 사용

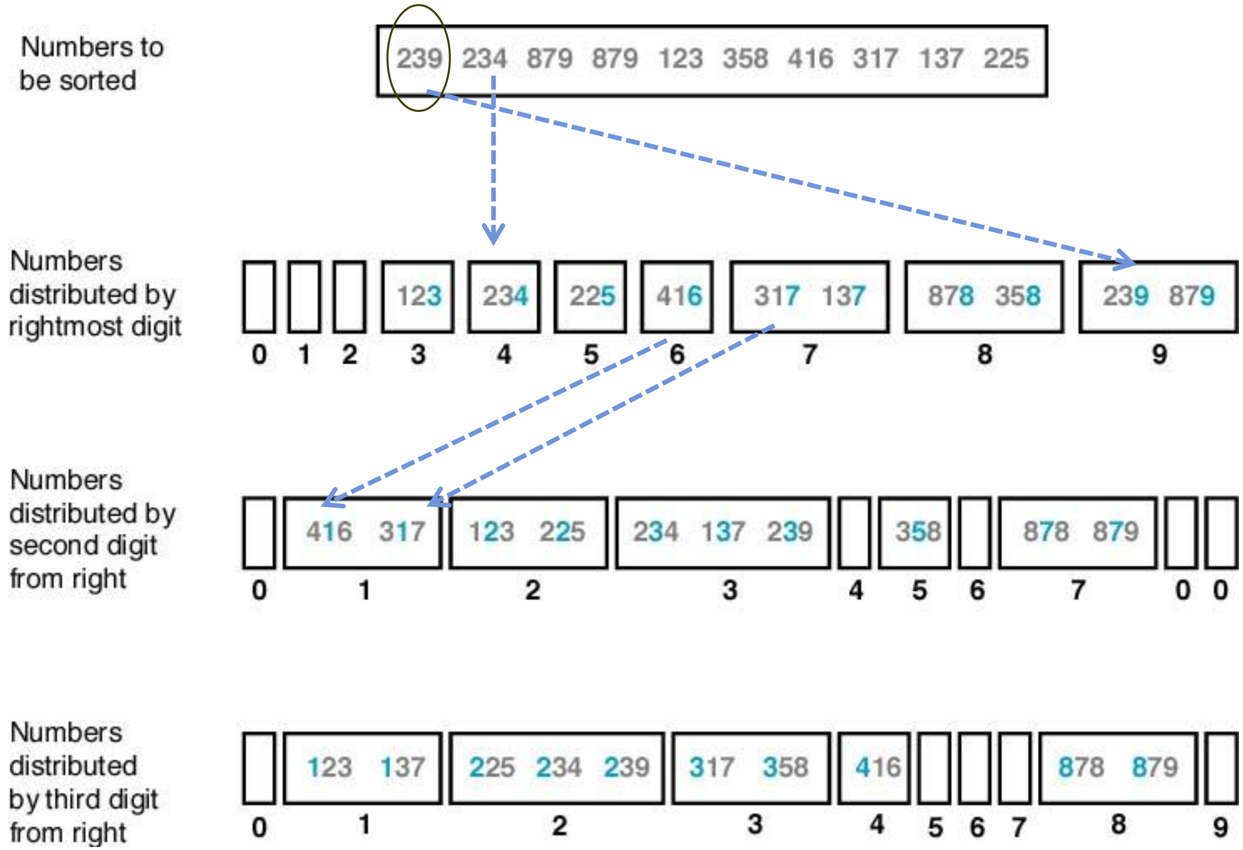
기수정렬(왼쪽에서 오른쪽 자리순으로)



The number of piles is not constant. Hard to operate it.

기수정렬(오른쪽에서 왼쪽 자리순으로)

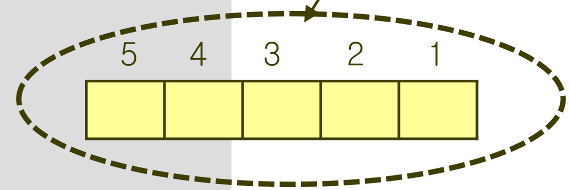
- 왼쪽에서 오른쪽순으로 하는 경우 뭉치(pile)를 구성하는 개수가 항상 일정하지 않으므로 관리하기가 쉽지 않다. 이를 해결하기 위해서는 다음 예와 같이 끝에 있는 디지트부터 먼저 조사를 시작하면 된다.



기수정렬

if numdigits=5

하나의 정수




```
void radixsort (node_pointer& masterlist, int numdigits){
   index i;
   node_pointer list[0..9];

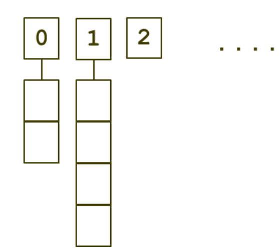
   for(i=1; i<= numdigits;i++){
       distribute(masterlist,i);
       coalesce(masterlist);
   }
 }
```

numdigits

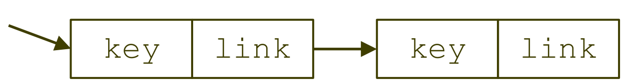
```
void distribute (node_pointer& masterlist, index i){
  index j;
  node_pointer p;
  for(j=0;j<=9;j++)
      list[j]=NULL;
  p = masterlist;
  while (p!=NULL){
    j=p->key에서(오른쪽에서) i번째 숫자의 값;
    p를 list[j]의 끝에 링크;
    p = p->link;
  }
}
```

n

list[j]

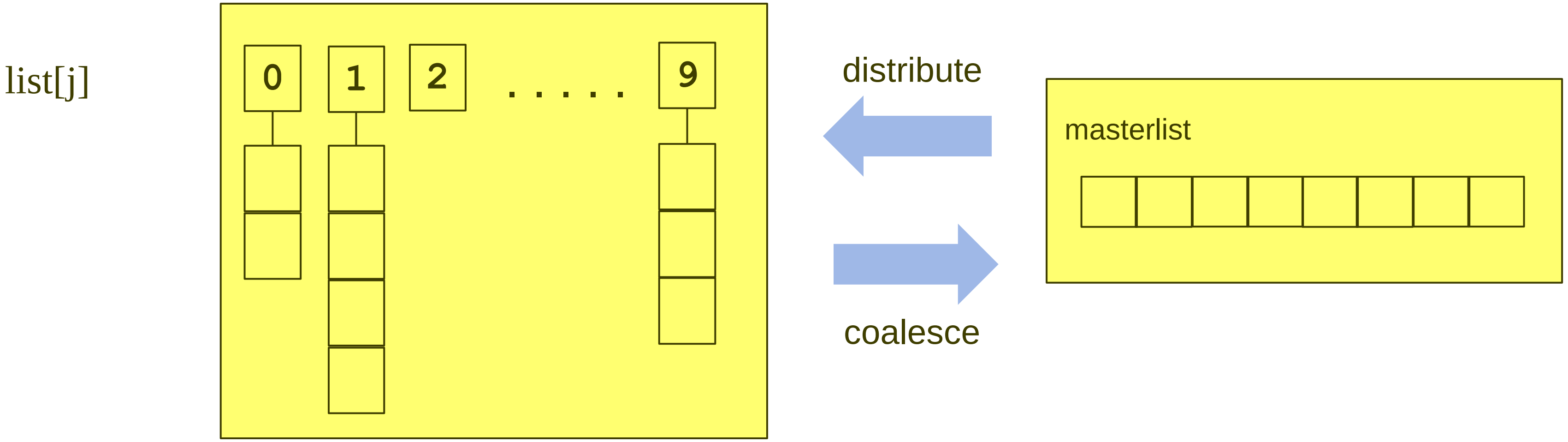


masterlist



```
void coalesce(node_pointer& masterlist){
    index j;
    masterlist = NULL;
    for(j=0; j<= 9; j++)
       list[j]에 있는 마디들을 masterlist의 끝에 링크
 }
```

10



기수정렬 알고리즘의 분석

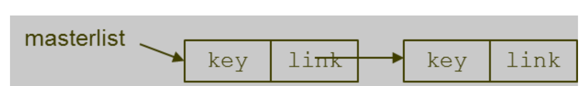
- 단위연산: 뭉치에 수를 추가하는 연산
- 입력크기: 정렬하는 정수의 개수 = n, 각 정수를 이루는 디지트의 최대 개수 = *numdigits*
- 모든 경우 시간복잡도 분석:

distribute coalesce

$$T(n) = numdigits \times (n + 10) \in \Theta(numdigits \times n)$$

따라서 *numdigits*가 n과 같으면, 시간복잡도는 $\Theta(n^2)$가 된다. 그러나 일반적으로 서로 다른 n개의 수가 있을 때 그것을 표현하는데 필요한 디지트의 수는 $\lg n$으로 볼 수 있다. 예: 주민등록번호는 13개의 디지트로 되어 있는데, 표현할 수 있는 개수는 10,000,000,000,000개 이다. 이 10조개의 번호를 기수정렬하는데 걸리는 시간은 $10{,}000{,}000{,}000{,}000 \times \log_{10} 10{,}000{,}000{,}000{,}000 =$ 130조

- 공간복잡도 분석
  - ✓ 추가적으로 필요한 공간은 키를 연결된 리스트로 만드는데 필요한 공간(link의 공간), 즉, $\Theta(n)$



기수정렬 알고리즘의 분석

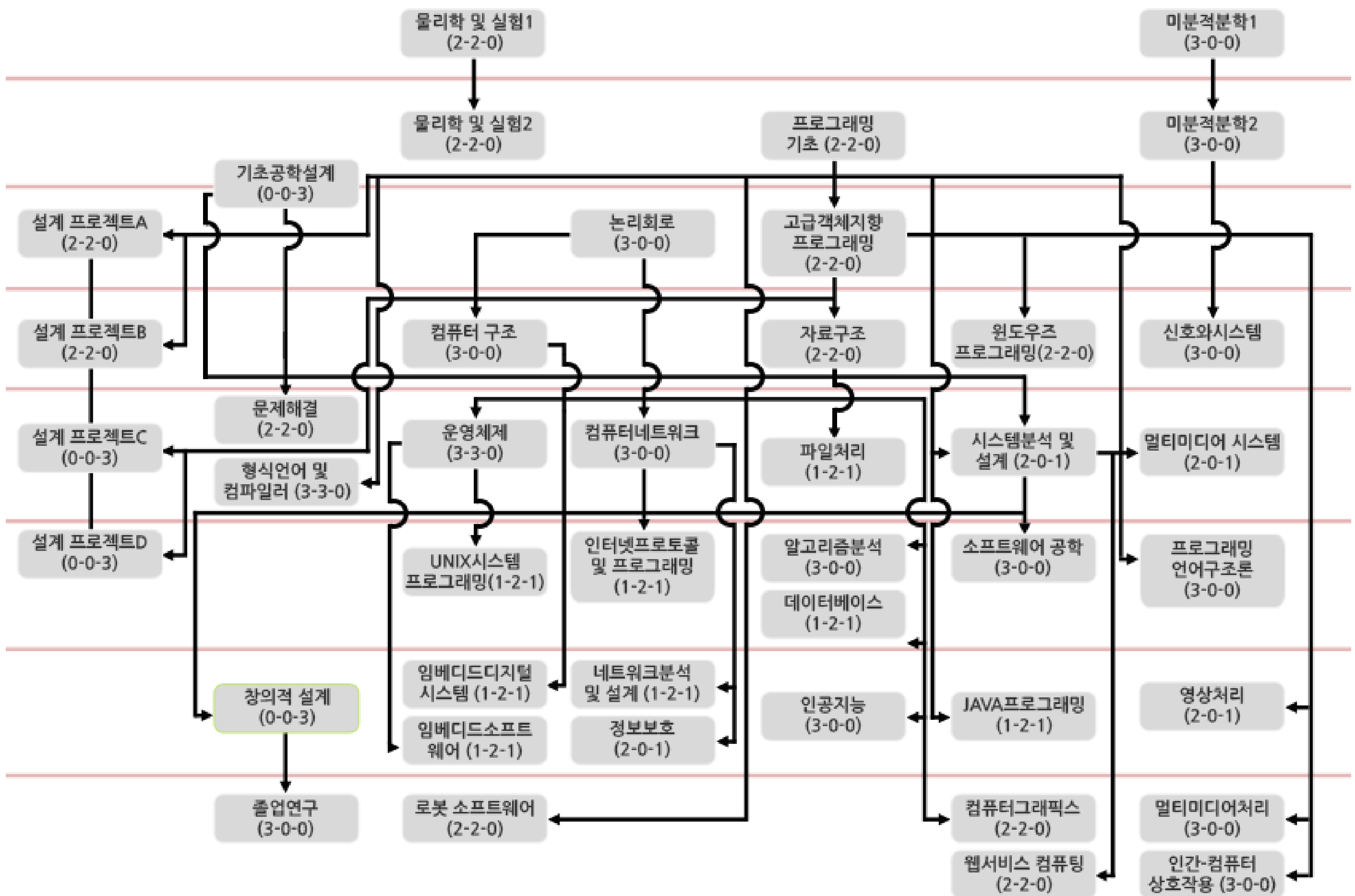
- 일반적인 기수 정렬의 시간복잡도:

$$T(n) = d \times (n + k)$$

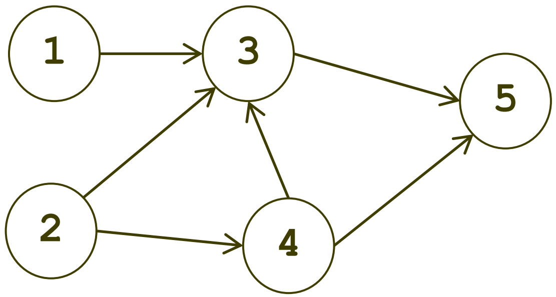
$$\begin{cases} d : \text{digit numbers (자리수)} \\ n : \text{정렬할 data 수} \\ k : \text{각 자리의 범위(2진수 : 2 (0 or 1), 10진수 : 10 (0 ~ 9))} \end{cases}$$

$$\therefore T(n) \in \Theta(dn + dk)$$

선수과목 순서를 일렬로 나열하라.



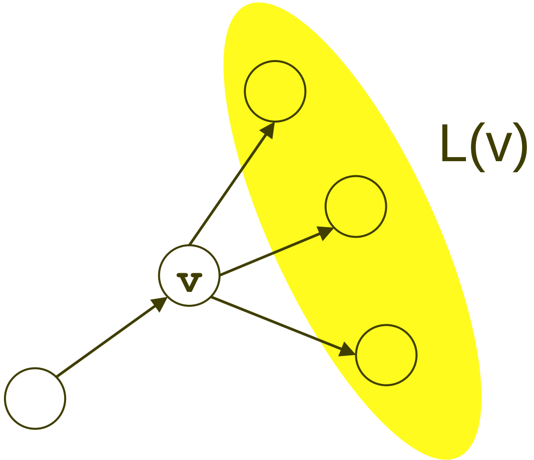
Topological Sort



- ✓ 정의: *i* 에서 *j*로 가는 arc가 있을 때 *i*가 *j*보다 먼저 오는 정렬 방법
- ✓ 물리적인 위치가 아니라 노드는 하나의 작업이라고 간주
- ✓ [예] 토지구입 후 인허가 과정을 두 노드의 관계로 표시

(예) 1 - 2 - 4 - 3 - 5

```
proc topological_sort
for v=1 to n
      mark[v]=unvisited
for v=1 to n
      if mark[v] unvisited
            dfs(v)
```




```
void dfs (vertex v){
  stack S
  push(v, S)
  mark[v]=visited;
  for each vertex w on L[v] do  // adjacent nodes
        if mark[w]= unvisited then
              dfs(w)
  print Top(S)
  Pop(S)
  }
```

original dfs

- L[*v*]: *v*의 neighbors
- topological sort의 역순으로 출력


7장 끝
수고하셨습니다.