[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

강의자료

(2020.09.24 목요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

[주제 2]

Heap-based Priority Queues and Heap Sort

(Review)

Heap Sort

[Horowitz 7.7]
[Neapolitan 7.6]

	1	2	3	4	5	6	7	8	9	10	
	26	5	77	1	61	11	59	15	48	19	← unordered
①	77	61	59	48	19	11	26	15	1	5	← max heap
②	1	5	11	15	19	26	48	59	61	77	← ordered

- **Method**
  ① Convert an input array of *n* unordered items into a max heap.
  ② Extract the items from the heap one at a time to build an ordered array.

주어진 정수들을 비감소 순서(non-decreasing order)대로 정렬하라.

```
void heapsort(element list[], int n)
/* perform a heapsort on the array */
{
   int i,j;
   element temp;

   for (i = n/2; i > 0; i--)
      adjust(list,i,n);
   for (i = n-1; i > 0; i--) {
      SWAP(list[1],list[i+1],temp);
      adjust(list,1,i);
   }
}
```

```
typedef struct{
  int key;
  /* other fields */
} element;
Element list[MAX_SIZE];
```

1. Make a (max) heap.

2. Extract items one by one.

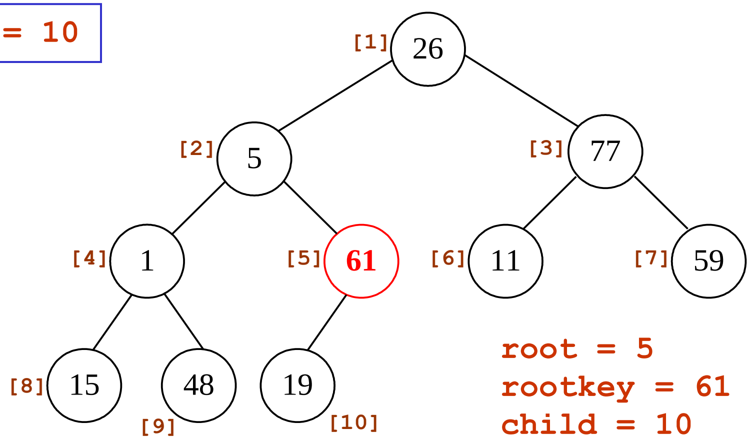
① $O(n)$ ② $O(n\log n)$ $\Rightarrow$ $O(n\log n)$

1. Make a Max Heap.

1 2 3 4 5 6 7 8 9 10
26 5 77 1 61 11 59 15 48 19 ← unordered
① 77 61 59 48 19 11 26 15 1 5 ← max heap

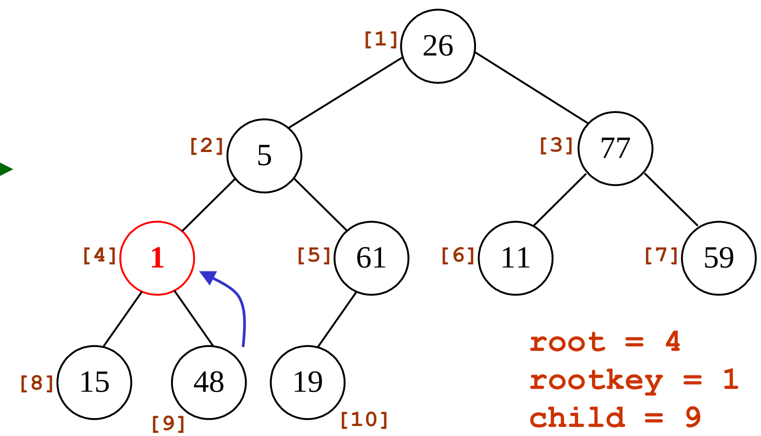
```
n = 10
```

```
adjust(list, 5, 10);
```



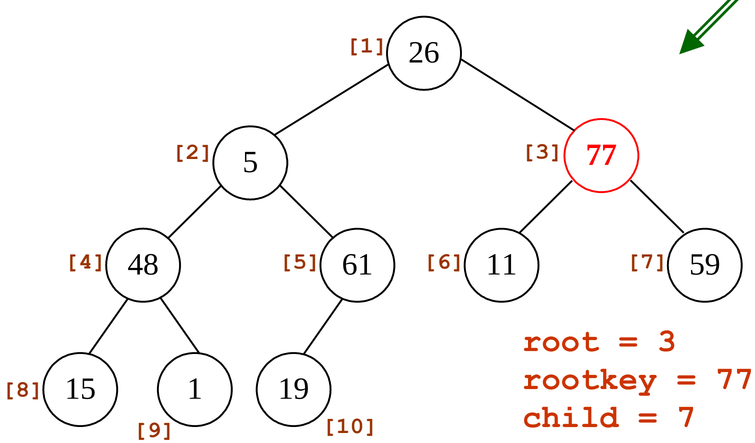

```
root = 5
rootkey = 61
child = 10
```

```
adjust(list, 4, 10);
```



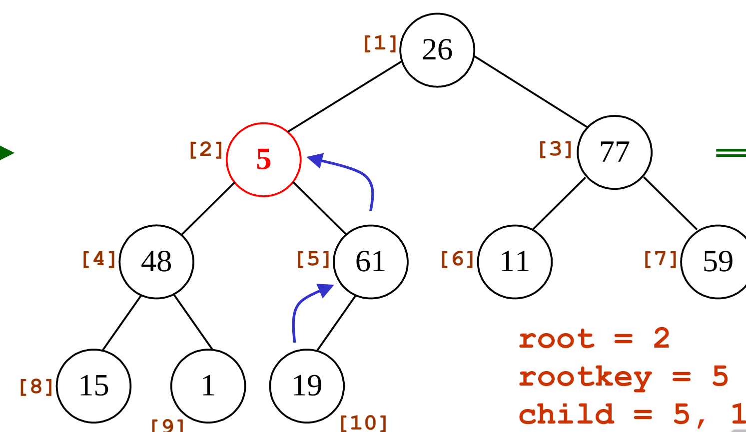

```
root = 4
rootkey = 1
child = 9
```

```
adjust(list, 3, 10);
```



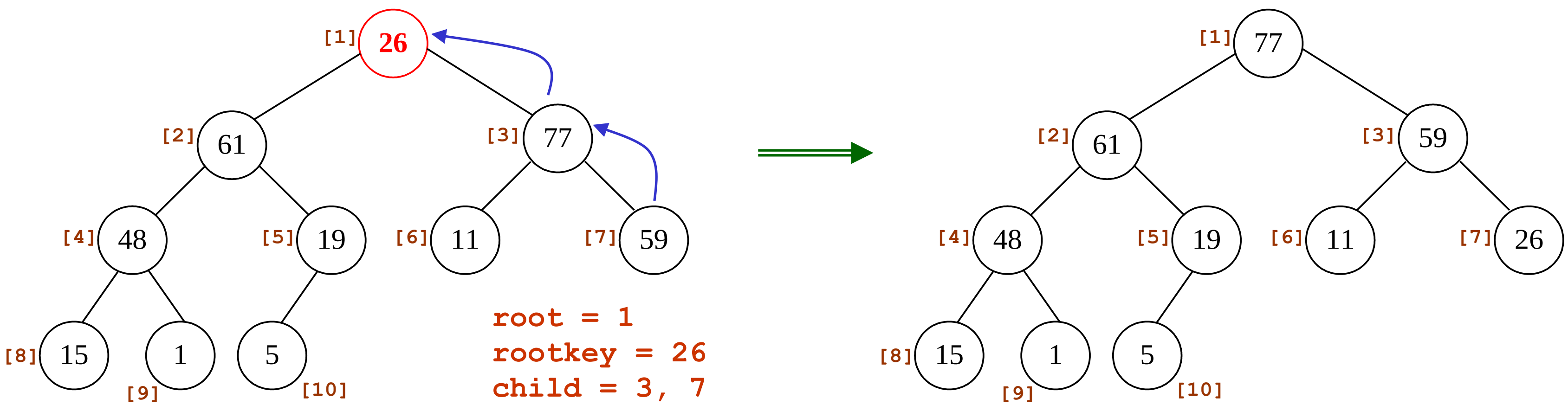

```
root = 3
rootkey = 77
child = 7
```

```
adjust(list, 2, 10);
```




```
root = 2
rootkey = 5
child = 5, 10
```

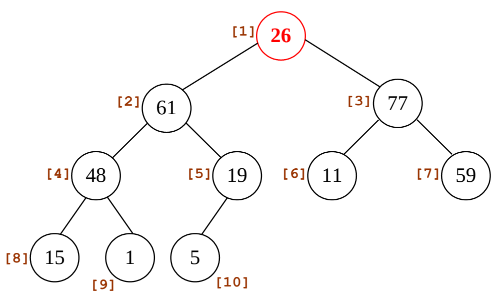
```
adjust(list, 1, 10);
```



	1	2	3	4	5	6	7	8	9	10	
	26	5	77	1	61	11	59	15	48	19	← unordered
①	77	61	59	48	19	11	26	15	1	5	← max heap
②	1	5	11	15	19	26	48	59	61	77	← ordered

The `adjust()` function

- **Input:** a binary tree *T* whose left and right subtrees satisfy the heap property but whose root may not
- **Output:** a binary tree *T* adjusted so that the entire binary tree satisfies the heap property




```
void adjust(element list[], int root, int n)
/* adjust the binary tree to establish the heap */
{
   int child,rootkey;
   element temp;
   temp = list[root];
   rootkey = list[root].key;
   child = 2 * root;         /* left child */
   while (child <= n) {
      if ((child < n) &&
      (list[child].key < list[child+1].key))
         child++;
      if (rootkey > list[child].key) /* compare root and
                                        max. child */
         break;
      else {
         list[child / 2] = list[child]; /* move to parent */
         child *= 2;
      }
   }
   list[child/2] = temp;
}
```

Executed *d* times, where *d* is the depth of the tree with root *i*

***O*(*d*) time**

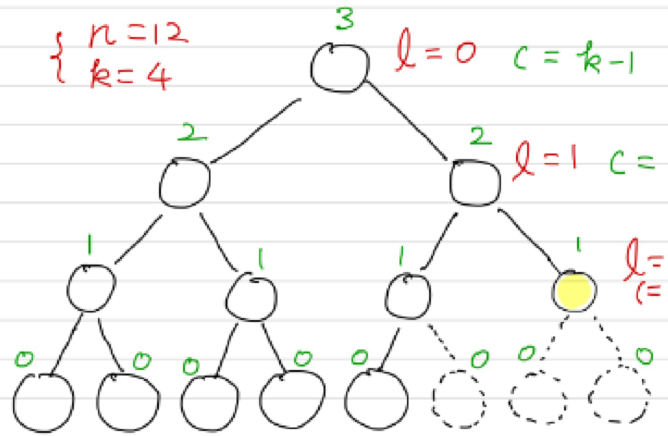
Cost of Make-Heap

Make Heap

$2^{k-1} \le n < 2^k$

$k-1 \le \log_2 n < k$

$\begin{cases} n=12 \\ k=4 \end{cases}$



$l=0 \quad c=k-1$

$l=1 \quad c=k-2$

$l=2 \quad c=1 \quad \leftarrow 2^{k-2}$

$(=k-2)$

$l=k-1 \quad c=0$

$$C_{MH} \le \underbrace{(k-1)\cdot 2^0 + (k-2)\cdot 2^1 + (k-3)\cdot 2^2 + \cdots + 1\cdot 2^{k-2}}_{I}$$

$$I = (k-1)\cdot 2^0 + (k-2)\cdot 2^1 + (k-3)\cdot 2^2 + \cdots + 1\cdot 2^{k-2}$$

$$2I = (k-1)\cdot 2^1 + (k-2)\cdot 2^2 + \quad + 2\cdot 2^{k-2} + 1\cdot 2^{k-1}$$

$$2I - I = -(k-1) + 2^1 + 2^2 + \cdots + 2^{k-1}$$

$$\therefore I = -k + \frac{1\cdot(2^k-1)}{2-1} = 2^k - k - 1$$

$$\Rightarrow C_{MH} \le 2^k - k - 1$$

$$\left.\begin{matrix} 2^k \le 2n \\ -k < -\log_2 n \end{matrix}\right\} \Rightarrow 2^k - k - 1 < 2n - \log_2 n - 1$$

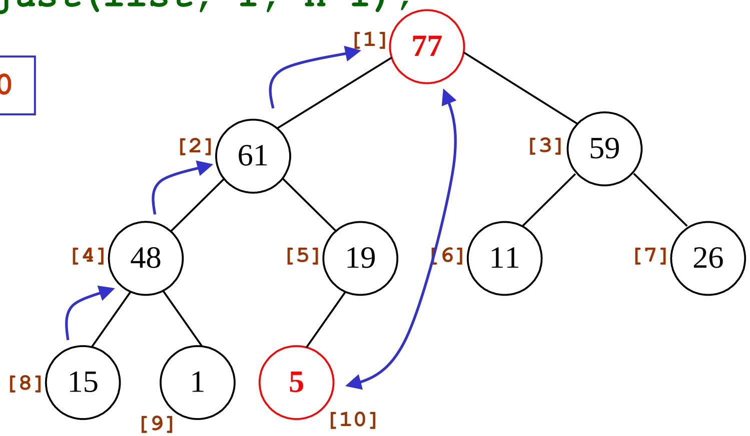
$$\Rightarrow C_{MH} = O(n)$$

2. Extract items one by one.

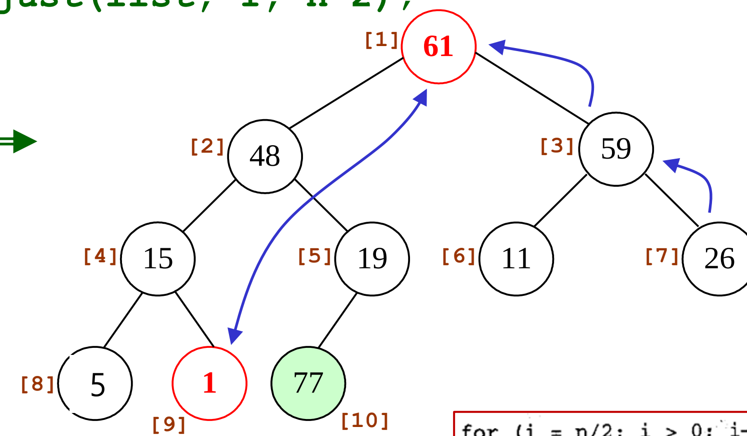
① 77 61 59 48 19 11 26 15 1 5 ← max heap
② 1 5 11 15 19 26 48 59 61 77 ← ordered

```
swap(list[1], list[(n-1)+1], temp);
adjust(list, 1, n-1);
```

n = 10

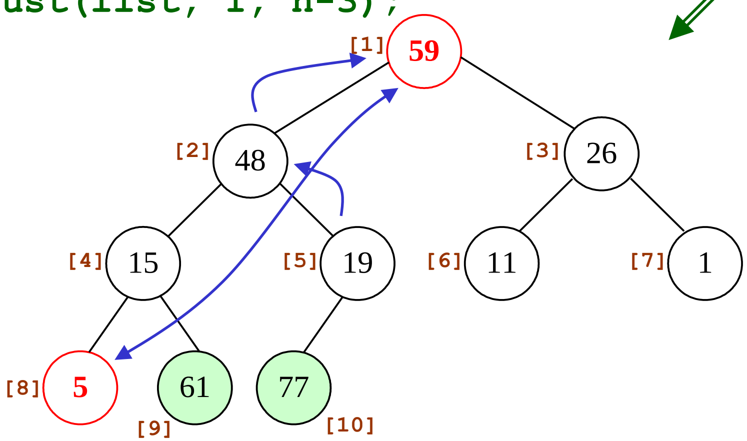


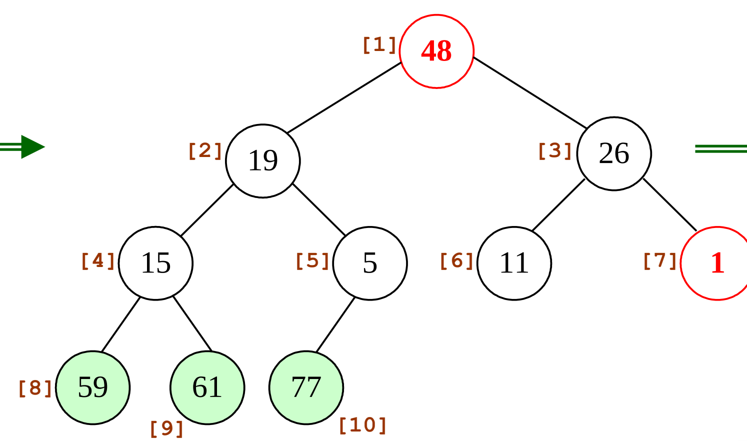

```
swap(list[1], list[(n-2)+1], temp);
adjust(list, 1, n-2);
```



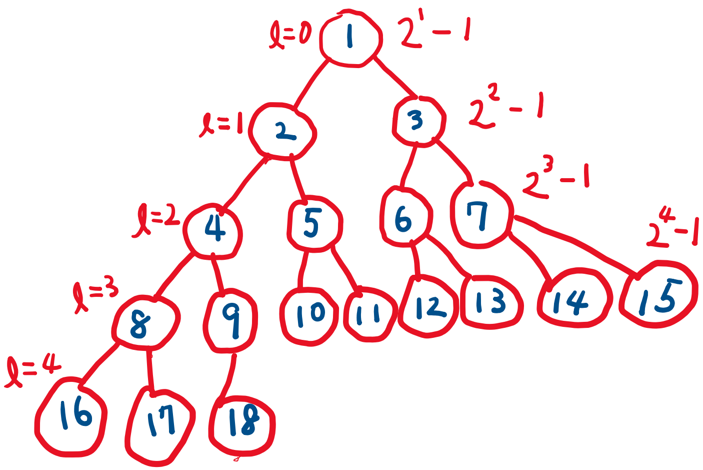

```
for (i = n/2; i > 0; i--)
  adjust(list,i,n);
for (i = n-1; i > 0; i--) {
  SWAP(list[1],list[i+1],temp);
  adjust(list,1,i);
```

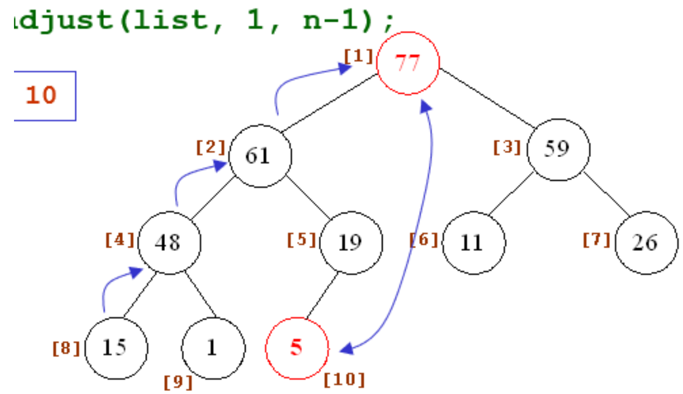
```
swap(list[1], list[(n-3)+1], temp);
adjust(list, 1, n-3);
```





Complexity of Item Extractions





For a given n, the cost is $c = \lfloor \log_2 n \rfloor$.
(depth)

$$2^c \le n < 2^{c+1} \Rightarrow c \le \log_2 n < c+1$$

$$C_{IE} = \lfloor \log n-1 \rfloor + \lfloor \log n-2 \rfloor + \lfloor \log n-3 \rfloor + \cdots + \lfloor \log 2 \rfloor + \lfloor \log 1 \rfloor$$

$$\le \log 2 + \log 3 + \cdots + \log n-1 < \sum_{i=2}^{n} \log_2 n = O(n \log n)$$

$$\text{Heap Sort : } C_{MH} + C_{IE} = O(n) + O(n \log n) = O(n \log n)$$

Priority Queue 2: Min-Max Heap

[Horowitz 9.1]

- Problem
  - The following operations must be performed as mixed in data processing:
    - Store a record with a key in an arbitrary order.
    - Fetch the record with the current largest key.
    - Fetch the record with the current smallest key.

- A solution: Design a data structure that offers the efficient implementation of the following operations (*Double-Ended Priority Queue*):
  - *Insert an element with an arbitrary key.*
  - *Delete an element with the largest key.*
  - *Delete an element with the smallest key.*

[주제 3]
Divide-and-Conquer Techniques
and
Sorting Techniques

Algorithm Design Techniques

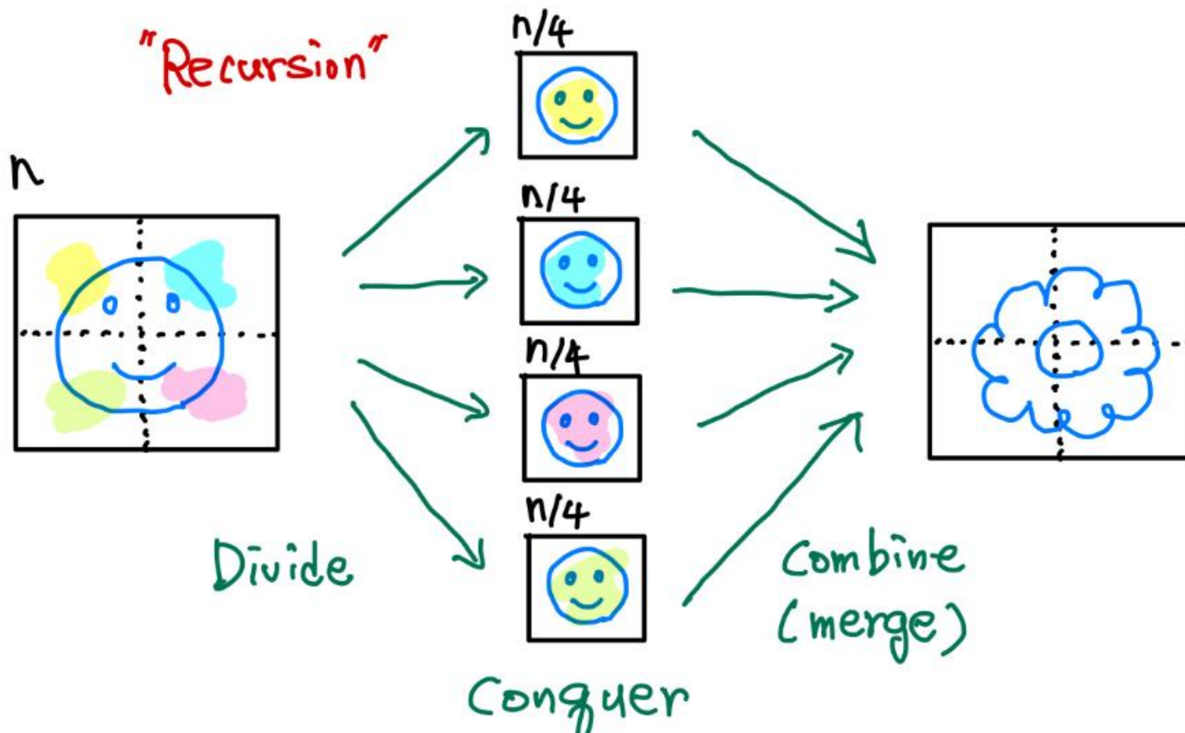
- **Divide-and-Conquer Method**
- Dynamic Programming Method
- Greedy Method
- Backtracking Method
- Local Search Method
- Branch-and-Bound Method
- Etc.

The Divide-and-Conquer Approach

① **Divide** an instance of a problem into one or more smaller instances.

② **Conquer** (Solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.

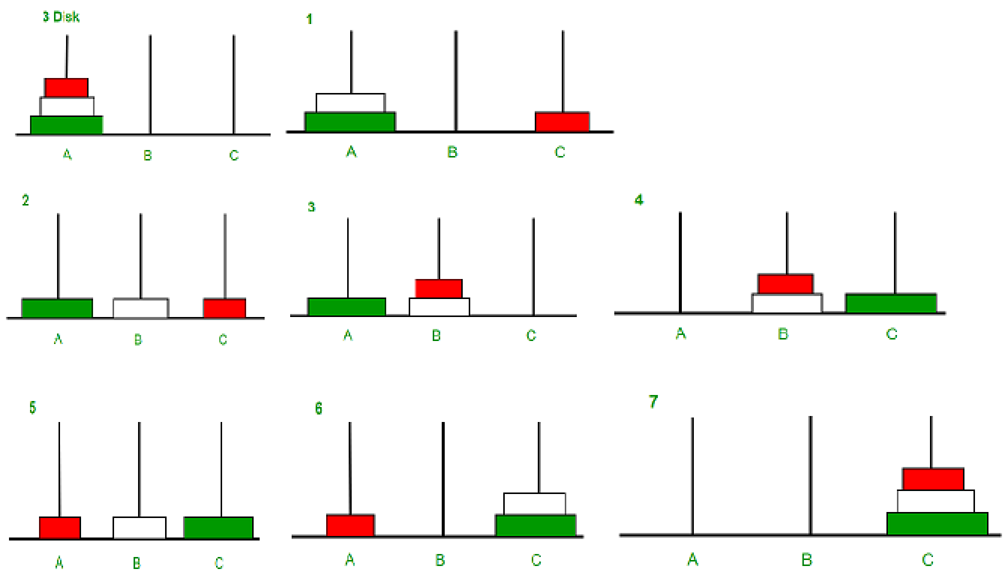
③ If necessary, **combine** the solutions to the smaller instances to obtain the solution to the original instance.



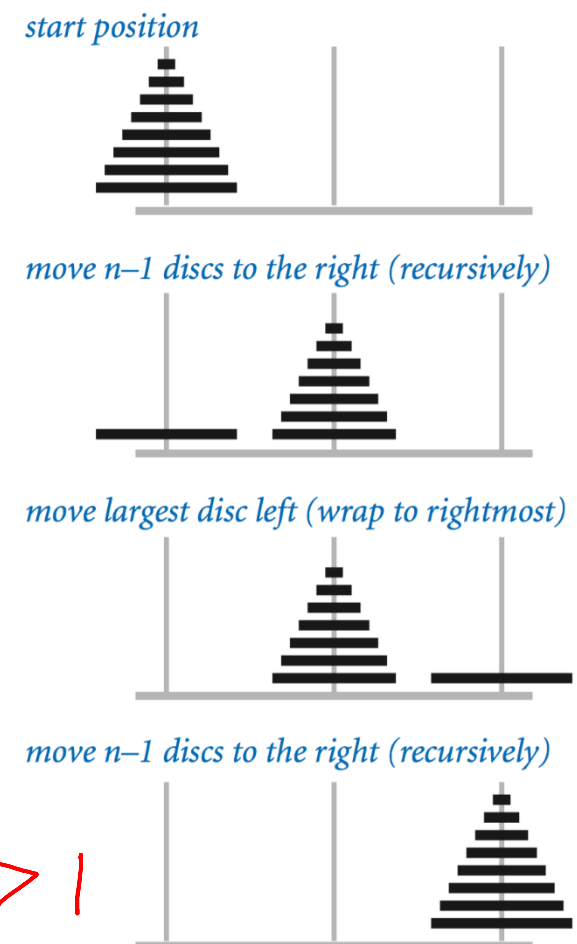
Recursion

- **Tower of Hanoi**

Recursive thinking!



https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/



https://introcs.cs.princeton.edu/java/23recursion/

$$T(n) = 2T(n-1) + 1, \quad n > 1$$

$$T(1) = 1$$

Sorting

A sorting algorithm is said to be **stable** if two items with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

- Problem: Given a list of *n* items, arrange them **in a certain order**.
  - Ex: non-increasing, non-decreasing, or etc.

- Some criteria for choosing a sorting algorithm
  - How many items will you be sorting?
  - Will there be duplicate items in the data?
  - What do you know about the data?
    - Is the data already partially sorted?
    - Do you know the distribution of the items?
    - Are the keys of items very long or hard to compare?
    - Is the range of possible keys very small?
  - Do you have to worry about disk accesses?
  - Do you need a **stable** sorting algorithm?
  - How much time do you have to write and debug your routine?

- `http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK4/NODE148.HTM`을 읽을 것.

A Formal Definition of Sorting

- A **partial order** on a set *S* is a relation *R* such that for each *a*, *b*, and *c* in *S*:
  - *aRa* is true (*R* is reflexive).
  - *aRb* and *bRc* imply *aRc* (*R* is transitive), and
  - *aRb* and *bRa* imply *a* = *b* (*R* is antisymmetric).
- A **linear order** or **total order** on a set *S* is a partial order *R* on *S* such that for every pair of elements *a*, *b*, either *aRb* or *bRa*.

- **The sorting problem**:

  Given a sequence of n elements a_1, a_2, $\cdots$, a_n drawn from a set having a linear order $\preccurlyeq$, find a permutation $\Pi = (\pi_1, \pi_2, \cdots, \pi_n)$ of $(1, 2, \cdots, n)$ that will map the sequence into a nondecreasing sequence $a_{\pi_1}, a_{\pi_2}, \cdots, a_{\pi_n}$ such that $a_{\pi_i} \preccurlyeq a_{\pi_{i+1}}$ for $1 \le i < n$.

- Ex: $\le$ on integer, $\subseteq$ on sets
- Sorting on data with partial order?

Merge Sort

Problem: Sort *n* keys in nondecreasing sequence.

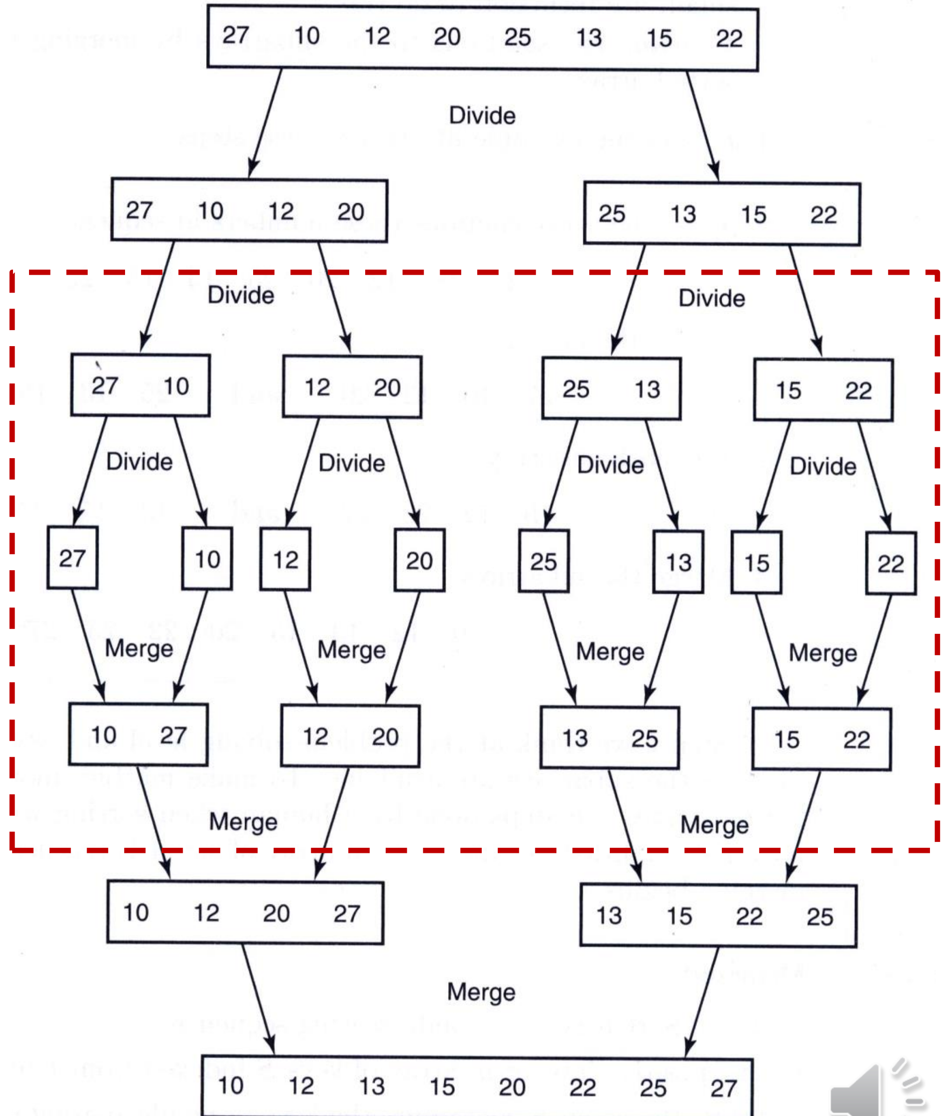
Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.

Outputs: the array *S* containing the keys in nondecreasing order.

① **Divide** the array into two subarrays each with ~n/2 items.

② **Conquer** each subarray by sorting it recursively.

③ **Combine** the solutions to the subarrays by merging them into a single sorted array.



- A **simple** implementation

```
// Sort a list from A[left] to A[right].
// Should be optimized for higher efficiency!!!

void merge_sort(item_type *A, int left, int right) {
  int middle;

  if (left < right) {
    middle = (left + right)/2;

    merge_sort(A, left, middle);
    merge_sort(A, middle + 1, right);

    merge(A, left, middle, right);
  }
}
```

– An example of merging two arrays

k	left	right	merged
1	10 12 20 27	13 15 22 25	10
2	10 12 20 27	13 15 22 25	10 12
3	10 12 20 27	13 15 22 25	10 12 13
4	10 12 20 27	13 15 22 25	10 12 13 15
5	10 12 20 27	13 15 22 25	10 12 13 15 20
6	10 12 20 27	13 15 22 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 25	10 12 13 15 20 22 25
-			10 12 13 15 20 22 25 27

```c
item_type *buffer; // extra space for merge sort, allocated beforehand

void merge(item_type *A, int left, int middle, int right) {
  int i, i_left, i_right;

  memcpy(buffer + left, A + left, sizeof(item_type)*(right - left + 1));

  i_left = left;
  i_right = middle + 1;
  i = left;

  while ((i_left <= middle) && (i_right <= right)) {
    if (buffer[i_left] < buffer[i_right])
      A[i++] = buffer[i_left++];
    else
      A[i++] = buffer[i_right++];
  }

  while (i_left <= middle)
    A[i++] = buffer[i_left++];
  while (i_right <= right)
    A[i++] = buffer[i_right++];
}
```