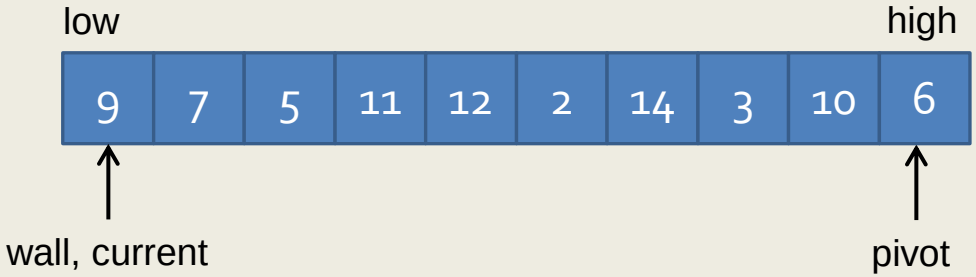
Quick Sort

- 퀵 정렬은 연속적인 분할-정복(divide-and-conquer) 알고리즘을 통해 이루어진다.
- 축(Pivot)값을 중심으로 왼쪽은 이 축 값보다 작은 값으로 오른쪽은 모두 이 축 값보다 큰 값을 배열시키는 것이다.
- 축 값의 왼쪽과 오른쪽 부분에 대해 또다시 분할 과정을 적용하여 분할의 크기가 1이 될 때까지 반복하면 전체적 정렬이 완료.
- 재귀(recursive) 알고리즘 사용
  - 배열 a를 pivot을 기준으로 2 부분으로 분할하여 최종적인 pivot의 위치를 결정한다 (예를 들면 mid)
  - 왼쪽파트에 대해 동일한 퀵 정렬 알고리즘 적용
  - 오른쪽 파트에 대한 동일한 퀵 정렬 알고리즘 적용

배열 Part 3

Divide Algorithm (with video)-Using Lomuto Partition Scheme



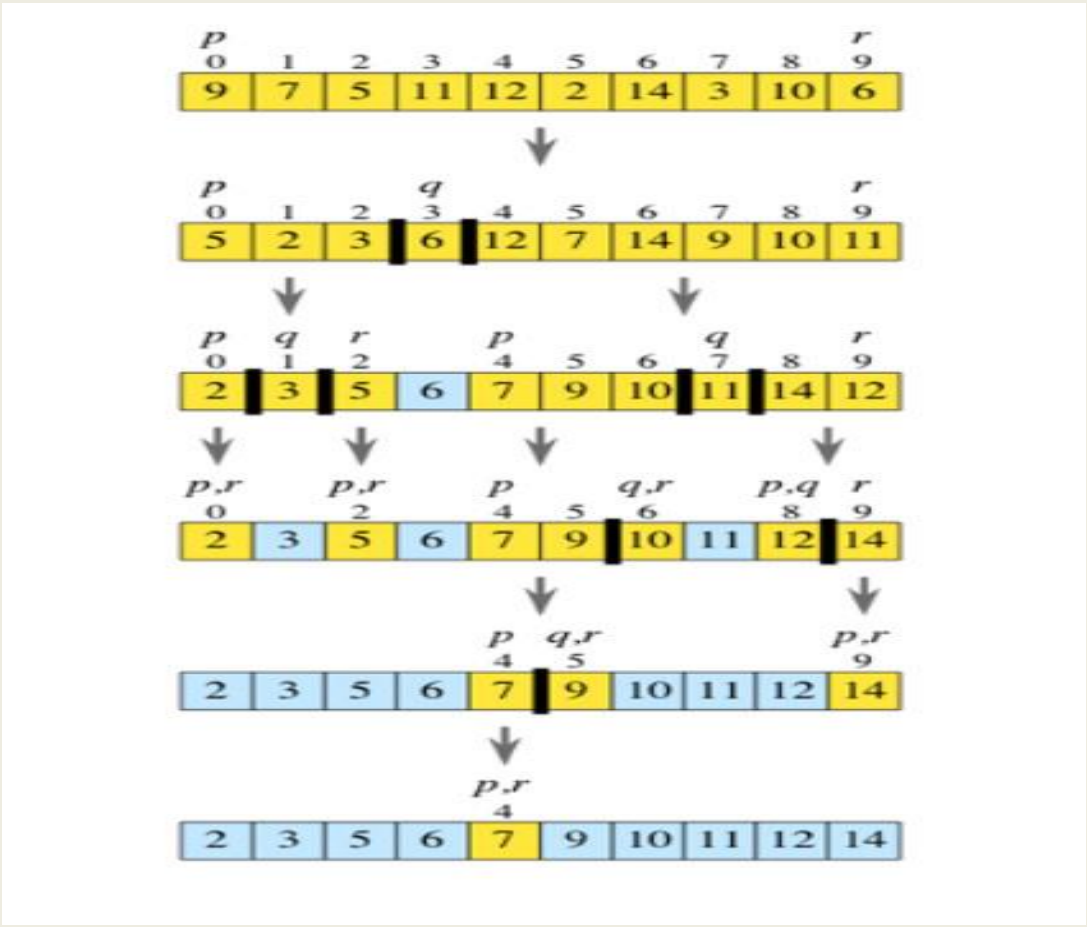

```
int Divide(int array[], int low, int high) {
        int wall, current, pivot;
        pivot = array[high];
        wall = low ;                    //place for swapping
        for (current = low; current < high; current++)
                if(array[current] <= pivot ) {
                        swap(&array[wall], &array[current]);
                        wall++;
                }
        swap(&array[wall], &array[high]);
        return wall ;
}
```

www.Bandicam.co.kr

QUICKSORT

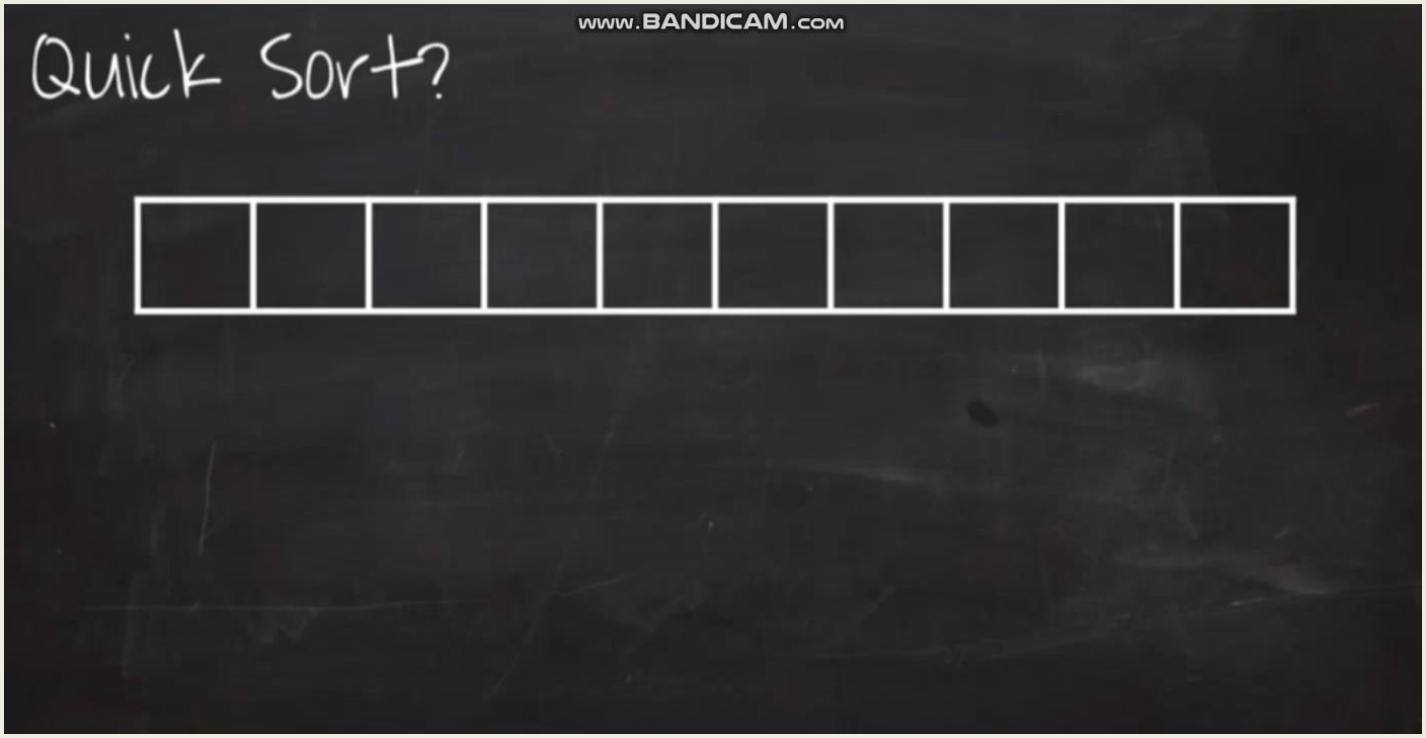
Quick Sort Using Lomuto Partition Scheme

```
void QuickSort(int array[], int low, int high) {
        int p;
        if(low < high) {
                p = divide(array, low, high);
                QuickSort(array, low, p - 1);
                QuickSort(array, p + 1, high);
        }
}

int divide(int array[], int low, int high) {
        int wall, current, pivot;
        pivot = array[high] ;
        wall = low ;                        //place for swapping
        for (current = low; current < high; current++)
                if(array[current] <= pivot ) {
                        swap(&array[wall], &array[current]);
                        wall++;
                }
        swap(&array[wall], &array[high]);
        return wall ;
}
```


p r
0 1 2 3 4 5 6 7 8 9
9 7 5 11 12 2 14 3 10 6
p q r
0 1 2 3 4 5 6 7 8 9
5 2 3 6 12 7 14 9 10 11
p q r p q r
0 1 2 4 5 6 7 8 9
2 3 5 6 7 9 10 11 14 12
p,r p,r p q,r p,q r
0 2 4 5 6 8 9
2 3 5 6 7 9 10 11 12 14
p q,r p,r
4 5 9
2 3 5 6 7 9 10 11 12 14
p,r
4
2 3 5 6 7 9 10 11 12 14

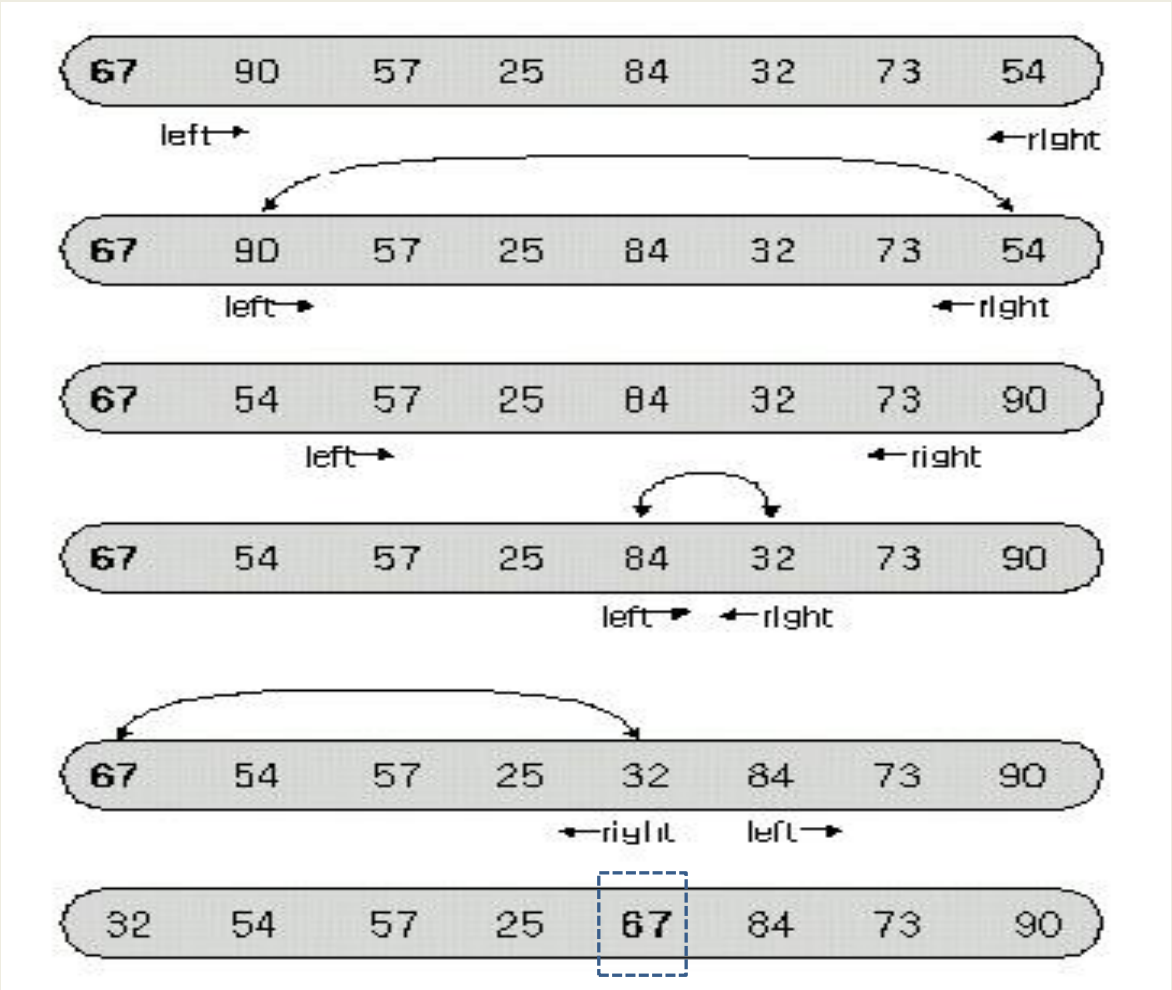
Quick Sort Using Hoare Partition Scheme

설명영상

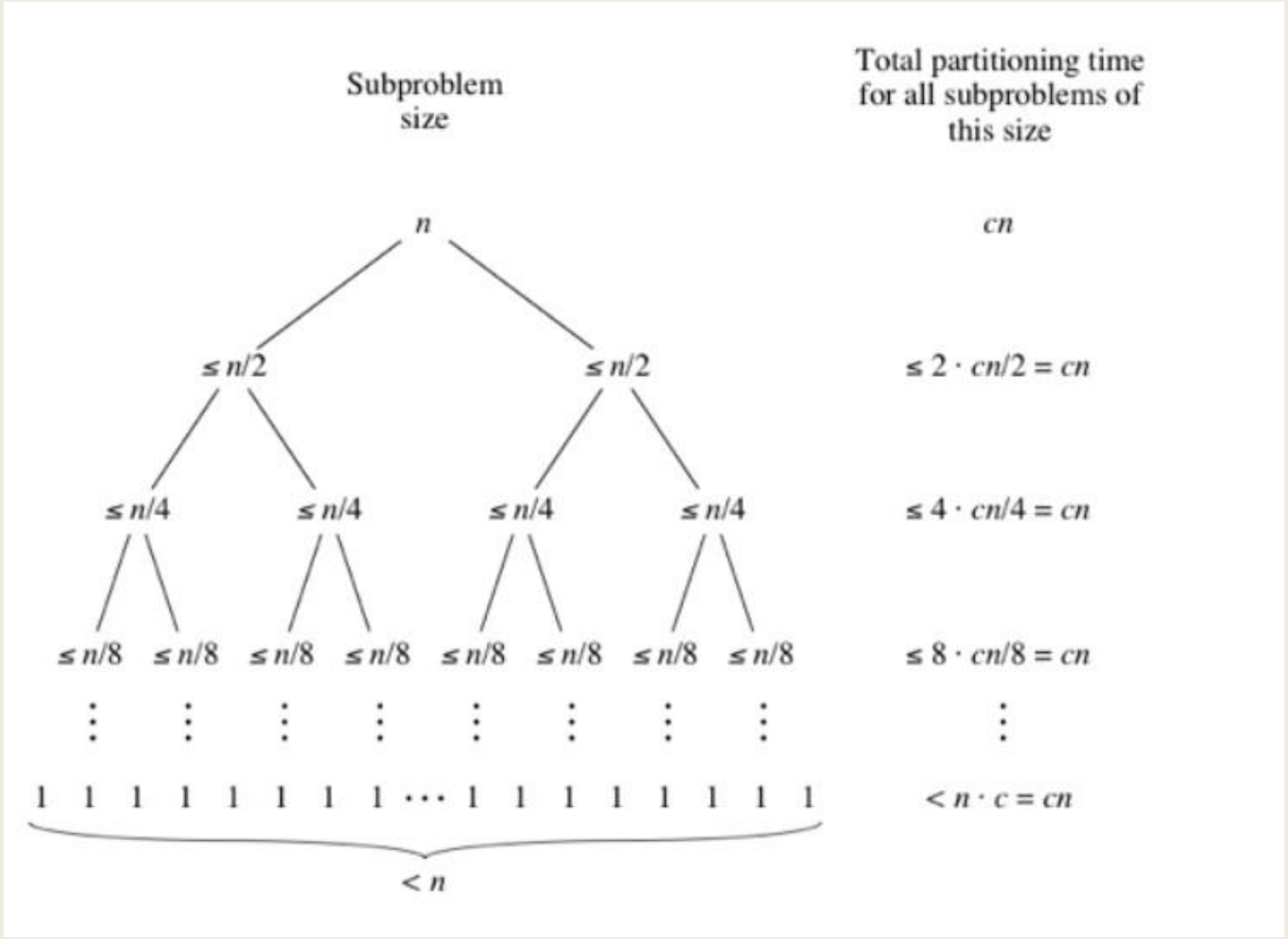


Quick Sort Using Hoare Partition Scheme

```
void QuickSort(int array[], int low, int high) {
        int p;
        if(low < high) {
                p = divide(array, low, high);
                QuickSort(array, low, p);
                QuickSort(array, p + 1, high);
}
int devide(int array[], int low, int high) {
        int pivot, left, right;
        pivot = array[low] ;
        left = low - 1 ;
        right = high + 1 ;
        while(1) {
                do right-- ;
                while(array[right] > pivot)
                do left++ ;
                while(array[left] < pivot)
                if(left < right)
                        swap(&array[left], &array[right]);
                else
                        return right;
        }
}
```


67 90 57 25 84 32 73 54
left
right
67 90 57 25 84 32 73 54
left
right
67 54 57 25 84 32 73 90
left
right
67 54 57 25 84 32 73 90
left
right
67 54 57 25 32 84 73 90
right
left
32 54 57 25 67 84 73 90

Best-case running time : O(nlogn)



C 언어가 제공하는 qsort()_

```
#include <stdio.h>
#include <search.h>
#include <string.h>

void main(void) {

    // qsort 라이브러리를 사용
    int array[] = { 3, 5, 6, 3, 1, 2, 7, 6, 7, 4, 8, 9, 3 };
    qsort(array, 13, sizeof(int), strcmp);

    for (i = 0; i < 13; i++){
        printf("%d₩t", array[i]);
    }

    return 0;
}
```

Merge Sort(병합정렬)

```
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is ₩n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("₩nSorted array is ₩n");
    printArray(arr, arr_size);
    return 0;
}
```

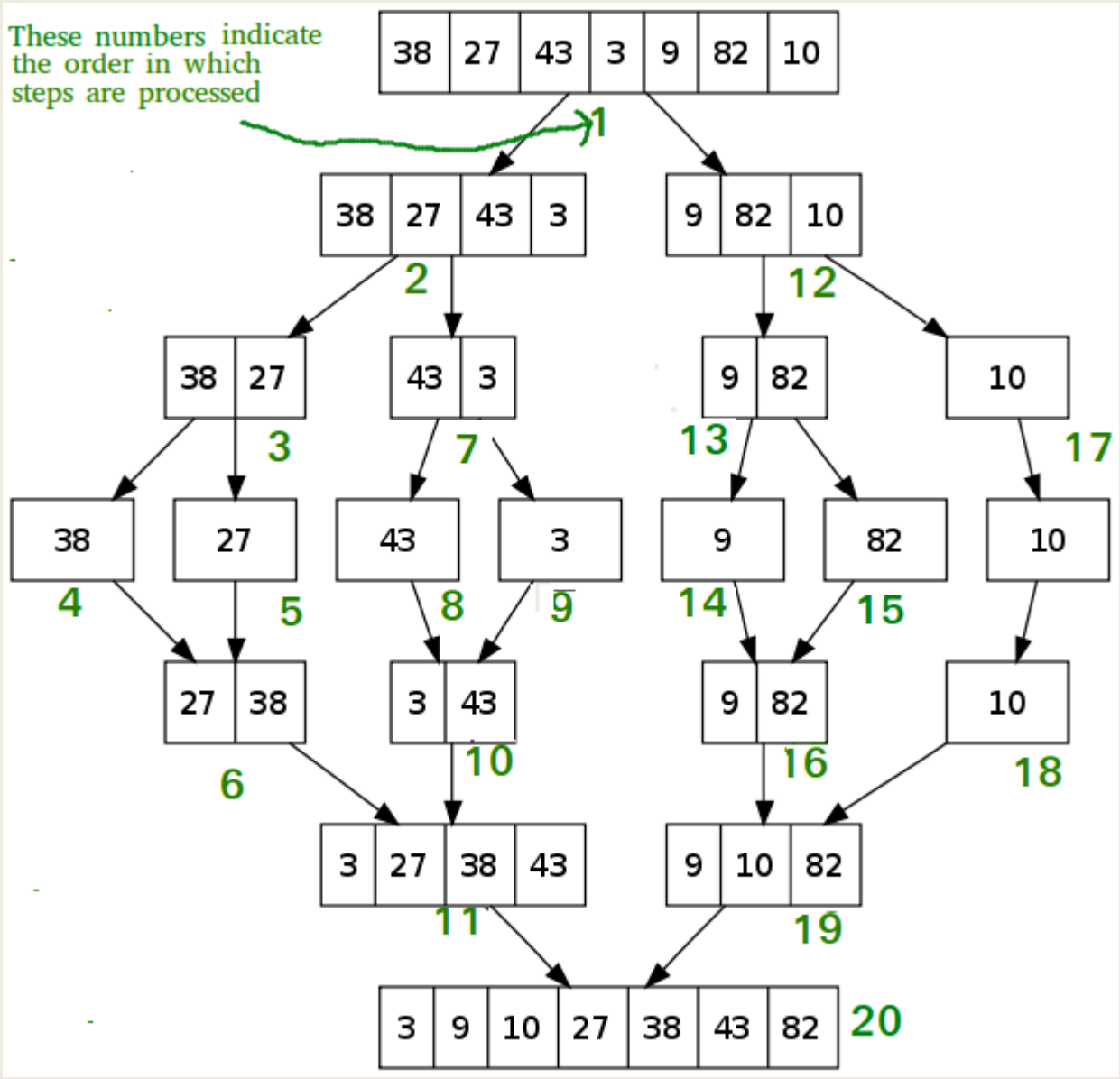
```
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
```

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {

        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
/* create temp arrays */
    int L[n1], R[n2];
/* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

/* Merge the temp arrays back into arr[l..r]*/
    i = 0;     // Initial index of first subarray
    j = 0;     // Initial index of second subarray
    k = l;     // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

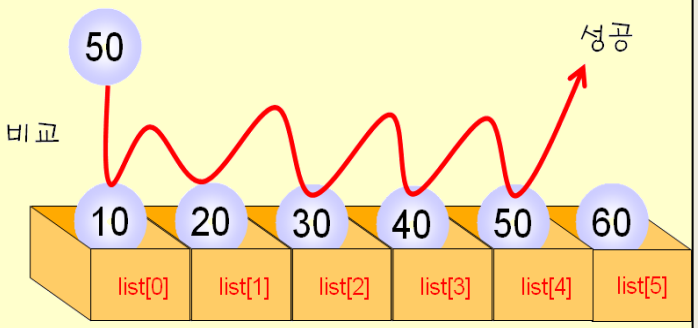
```
/* Copy the remaining elements of L[], if there are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

/* Copy the remaining elements of R[], if there are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
```


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

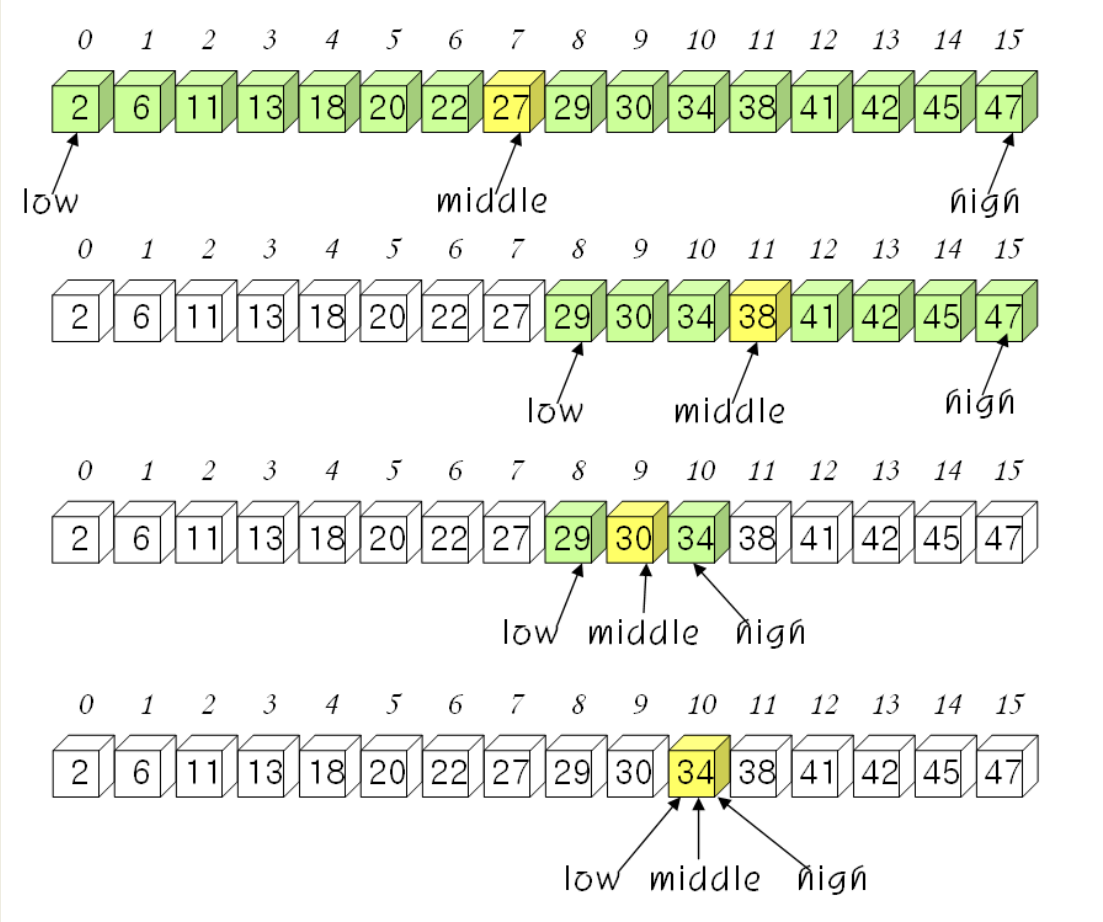
순차 탐색(Sequential Search)

```c
#include <stdio.h>
#define SIZE 6

int seq_search(int list[], int n, int key);

int main(void)
{
        int key;
        int grade[SIZE] = { 10, 20, 30, 40, 50, 60 };

        printf("탐색할 값을 입력하시오:");
        scanf("%d", &key);
        printf("탐색 결과 = %d₩n", seq_search(grade, SIZE, key));

        return 0;
}
int seq_search(int list[], int n, int key)
{
        int i;

        for(i = 0; i < SIZE; i++)
                if(list[i] == key)
                        return i;     // 탐색이 성공하면 인덱스 반환
        return -1;                    // 탐색이 실패하면 -1 반환
}
```



이진 탐색(Binary Search)

- 이진 탐색(binary search): 정렬된 배열의 중앙에 위치한 원소와 비교 되
  풀이-그러므로 **먼저 정렬이 되어 있어야 한다**



- **이진 탐색 알고리즘**(binary search algorithm)
  - 정렬된 리스트에서 특정한 값의 위치를 찾는 알고리즘
  - 정렬된 리스트에만 사용할 수 있다는 단점이 있지만, 검색이 반복될 때마다 목표 값을 찾을 확률은 두 배가 되므로 속도가 빠르다는 장점이 있다
  - 분할 정복 알고리즘

반복문 사용

```
int binary_search(int list[], int n, int key)
{
        int low, high, middle;

       low = 0;
       high = n-1;

       while( low <= high ){        // 아직 숫자들이 남아있으면
                middle = (low + high)/2;    // 중간 요소 결정
                if( key == list[middle] )  // 일치하면 탐색 성공
                        return middle;
                else if( key > list[middle] )// 중간 원소보다 크다면
                        low = middle + 1;    // 새로운 값으로 low 설정
                else
                        high = middle - 1;   // 새로운 값으로 high 설정
        }
        return -1;
}
```

재귀함수 이용

```
BinarySearch(list[], key, low, high) {

   if (high < low)
            return -1 ;                    // not found
   mid = (low + high) / 2 ;

   if (list[mid] > key)
            return BinarySearch(list, key, low, mid-1) ;

   else if (list[mid] < key)
            return BinarySearch(list, key, mid+1, high) ;

   else
            return mid ;                   // found
}
```