Reference textbook: R. Napolitan and K. Naimipour, Foundations of Algorithms ($4^{th}$ ed.), Jones and Bartlett, 2011

# ACM Tutorial: Divide-and-Conquer


**Dr. Ukrit Watchareeruetai**

Department of Engineering and Technology,
International College, KMITL

# Contents

- Introduction to divide-and-conquer
- Binary search
- Mergesort
- When not to use divide-and-conquer

# Introduction to divide-and-conquer

# Concept of divide-and-conquer

- *Divide an instance* of a problem into two or more *smaller instances*
- The smaller instances are usually instances of the original problem.
- If the *solutions to the smaller instances* can be obtained readily, the solution to the original instance can be obtained by *combining these solutions*.

# Concept of divide-and-conquer

- If the smaller instances are still too large to be solved, they can be divide into still smaller instances.

- The process of *dividing* the instances continues *until they are so small* that a solution is readily obtainable.

# Concept of divide-and-conquer

- Divide-and-conquer is a top-down approach
  - The solution to the top-level instance of a problem is obtained by going down and obtaining solutions to the smaller instances.
  - This method is used by *recursion routines*.
  - But we can sometimes create a more efficient iterative version of the algorithm.

# Binary search

## Algorithm 1.5: Binary Search

Problem: Determine whether *x* is in the sorted array *S* of *n* keys.

Inputs: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

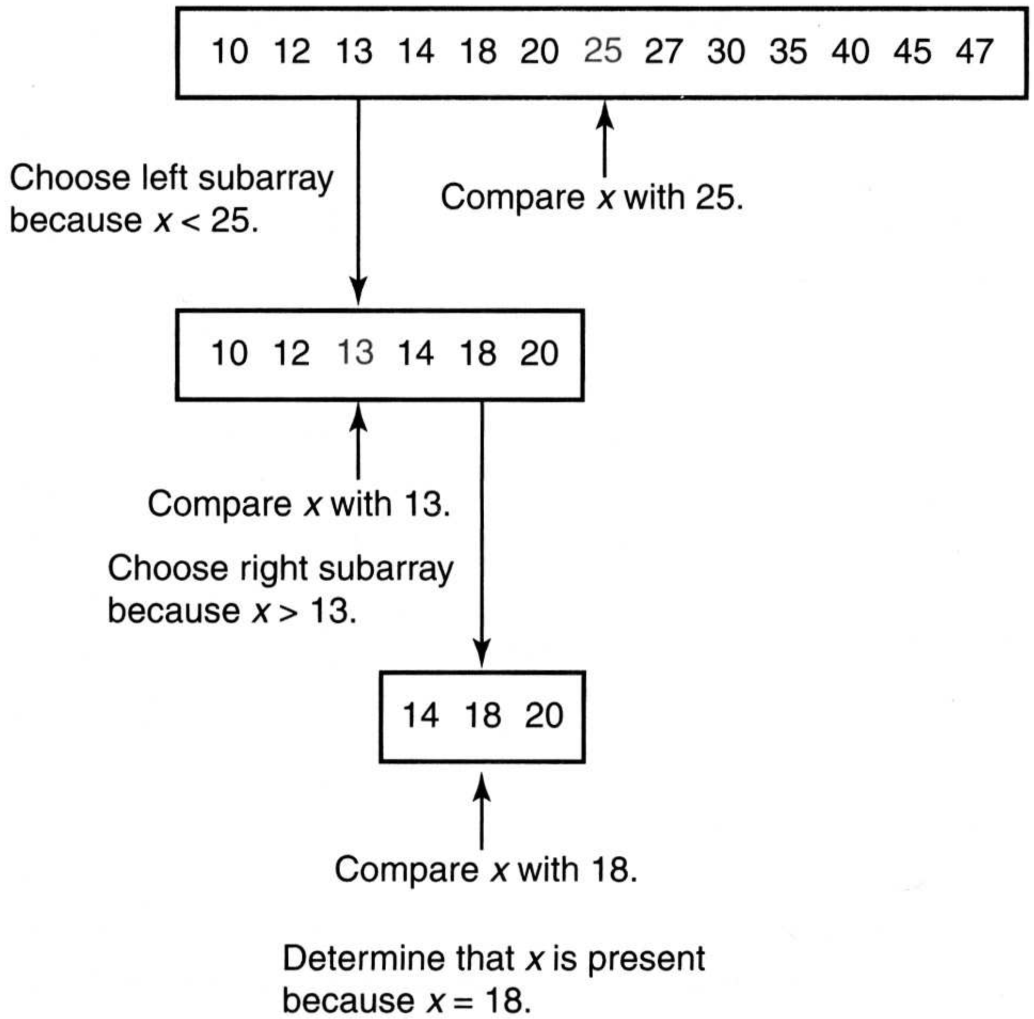
```
void binsearch (int n,
                const keytype S[],
                keytype x,
                index& location)
{
   index low, high, mid;

   low = 1; high = n;
   location = 0;
   while (low < = high &&  location = = 0){
      mid = ⌊(low + high)/2⌋;
      if (x = = S[mid])
         location = mid;
      else if (x < S[ mid ])
         high = mid - 1;
      else
         low = mid + 1;
      }
}
```

Iterative version of binary search

# Binary search

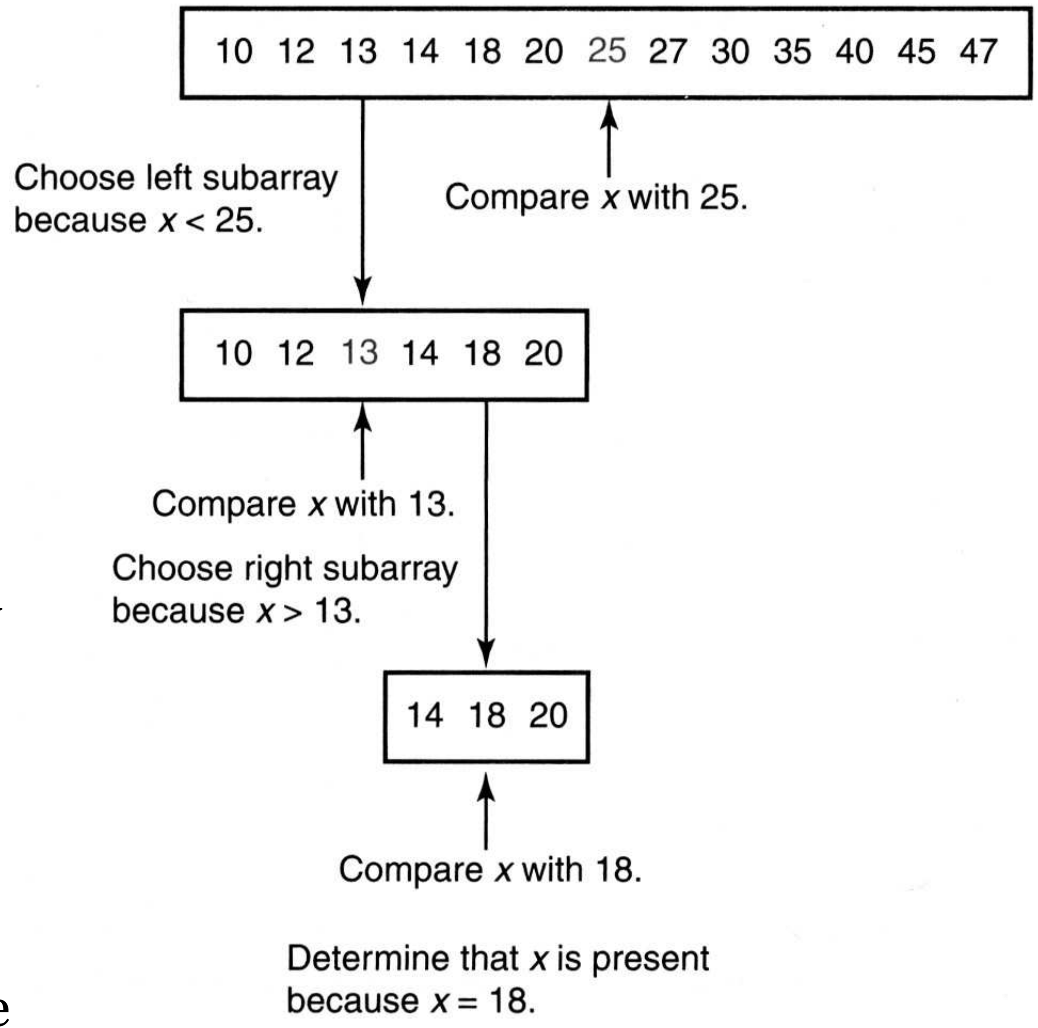
- Array must be sorted.
- Compare a query value $x$ with the middle item $m$
  - If $x = m$, the solution is found.
  - If not, the array is divided into subarrays
    - If $x$ is less than $m$, ignore the right subarray.
    - If $x$ is greater than $m$, ignore the left subarray.
- Repeat this process until $x$ is found or all data exhaust



# Binary search

- If $x$ equals the middle item, quit. Otherwise:

1. *Divide* the array into subarrays. If $x$ is smaller, choose the left subarray. If $x$ is larger, choose the right subarray.
2. *Conquer* (solve) the subarray by determining whether $x$ is in that subarray. Unless the subarray is sufficient small, use recursion.
3. *Obtain* the solution to the array from the solution to the subarray



# Binary search

Suppose $x$ = 18 and we have the following array:

10 12 13 14 18 20 25 27 30 35 40 45 47.

↑

Middle item

1. Divide the array: Because $x$ <25, we need to search

   10 12 13 14 18 20.

2. Conquer the subarray by determining whether $x$ is in the subarray. This is accomplished by recursively dividing the subarray. The solution is:

   Yes, $x$ is in the subarray.

3. Obtain the solution to the array from the solution to the subarray:

   Yes, $x$ is in the array.

# Binary search

- When developing a recursive algorithm, we need to:
  - Develop a way to obtain the solution to an instance from the solution to one or more smaller instances.
  - Determine the terminal condition(s) that the smaller instance(s) is (are) approaching.
  - Determine the solution in the case of the terminal condition(s).

## Algorithm 2.1: Binary Search (Recursive)

Problem: Determine whether *x* is in the sorted array *S* of size *n*.

Inputs: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

```
index location (index low, index high)
   {
  index mid;
  if (low > high)
       return 0;
  else {
       mid = ⌊(low + high)/2⌋;
       if (x == S[mid])
            return mid
       else if (x <  S[mid])
            return location(low, mid - 1);
       else
            return location(mid + 1, high);
     }
   }
```

Recursive version

# Mergesort

# Mergesort

- Perform sorting by using two-way merging
- For example, to sort an array of 16 items
  - Divide it into two subarrays, each of size 8. Sort them and then merge them to produce sorted array.
  - Again, to sort each subarray of size 8, we can divide them into subarrays of size 4, sort them, and then merge them to produce sorted subarrays of size 8.
  - Eventually, the size of subarrays will become 1 and it is trivially sorted.

# Mergesort

- Given an array of size n, merge sort involves the following steps:
  1. *Divide* the array into two subarrays each with n/2 items.
  2. *Conquer* (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
  3. *Combine* the solutions to the subarrays by merging them into a single sorted array.

# Mergesort (Example)

- Suppose the array contains these numbers in sequence: 27 10 12 20 25 13 15 22.

1. Divide the array:

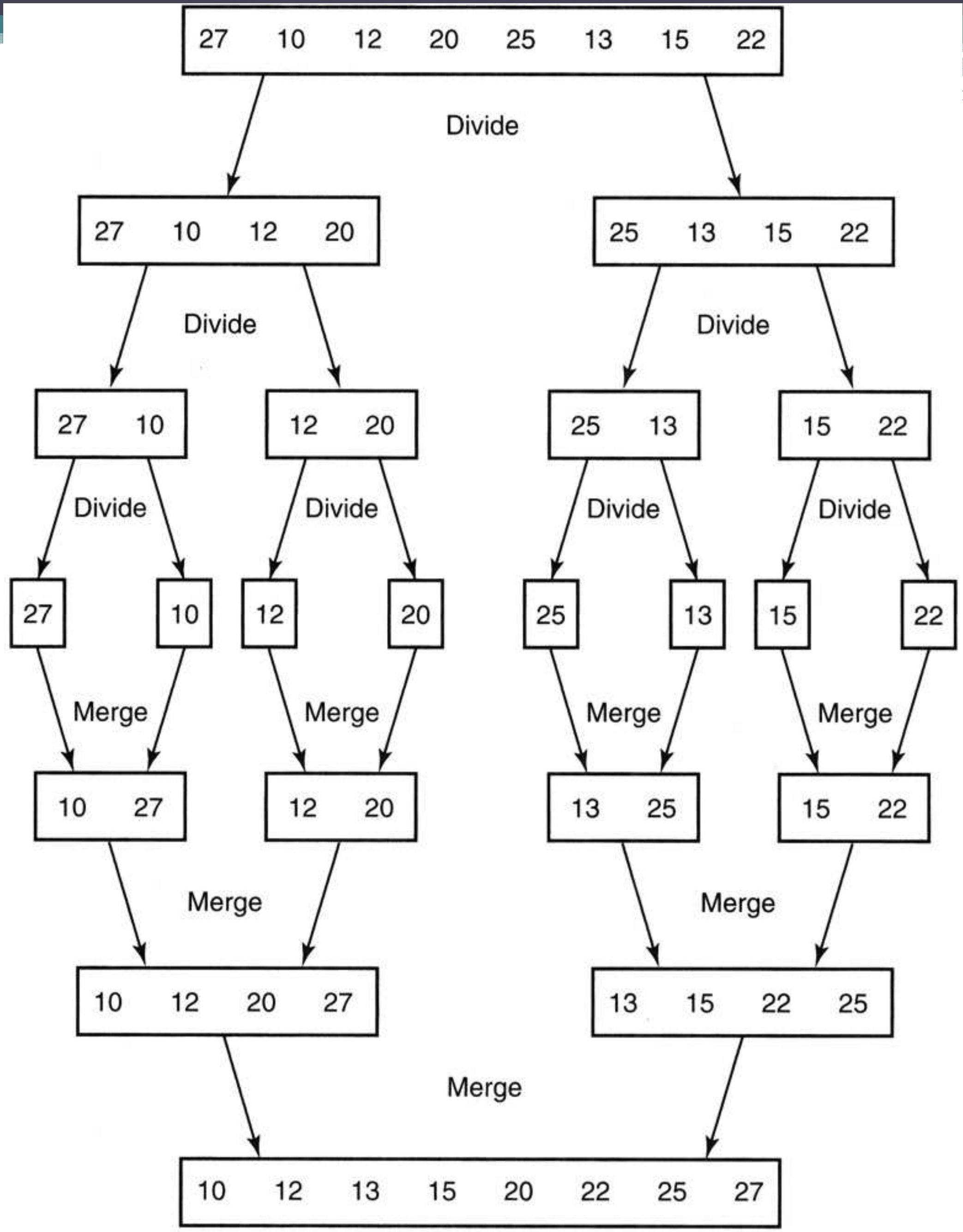
   27 10 12 20 and 25 13 15 22.

2. Sort each subarray:

   10 12 20 27 and 13 15 22 25.

3. Merge the subarrays:

   10 12 13 15 20 22 25 27.


27 10 12 20 25 13 15 22
Divide
27 10 12 20
25 13 15 22
Divide
Divide
27 10
12 20
25 13
15 22
Divide
Divide
Divide
Divide
27
10
12
20
25
13
15
22
Merge
Merge
Merge
Merge
10 27
12 20
13 25
15 22
Merge
Merge
10 12 20 27
13 15 22 25
Merge
10 12 13 15 20 22 25 27

## Algorithm 2.2: Mergesort

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.

Floor function

```
void mergesort (int n, keytype S[])
{
      if (n>1) {
            const int h=⌊n/2⌋, m = n - h;
            keytype U[1 ..h], V[1 ..m];
            copy S[1] through S[h] to U[1] through U[h];
            copy S[h+1] through S[n] to V[1] through V[m];
            mergesort(h, U);
            mergesort(m, V);
            merge (h, m, U, V, S);
      }
}
```

## Algorithm 2.3: Merge

Problem: Merge two sorted arrays into one sorted array.

Inputs: positive integers *h* and *m* , array of sorted keys *U* indexed from 1 to *h* , array of sorted keys *V* indexed from 1 to *m*.

Outputs: an array *S* indexed from 1 to *h* + *m* containing the keys in *U* and *V* in a single sorted array.

```
void merge (int h, int m, const keytype U[],
            const keytype V[],
            keytype S[])
{
  index i, j, k;

  i = 1; j = 1; k = 1;
  while (i <=h && j <= m) {
     if (U[i] < V[j]) {
              S[k] = U[i];
              i++;
        }
        else{
              S[k] = V[j];
              j++;
        }
        k++;
        }
  if (i>h)
        copy V[j] through V[m] to S[k] through S[h+m];
  else
        copy U[i] through U[h] to S[k] through S[h+m];
}
```

**Table 2.1: An example of merging two arrays *U* and *V* into one array *S* [* ]**

| *k* | *U* | *V* | *S* (Resutl) |
|---|---|---|---|
| 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 20 27 | 13 15 22 25 | 10 12 13 15 20 22 25 27 ← Final values |

[* ] Items compared are in boldface. Items just exchanged appear in squares.

# When not to use divide-and-conquer

# When not to use divide-and-conquer

- If possible, we should avoid divide-and-conquer in the following two cases:
  - An instance of size $n$ is divided into two or more instances each almost of size $n$.
    - Lead to exponential-time algorithm
  - An instance of size $n$ is divided into almost $n$ instances of size $n/c$, where $c$ is a constant.
    - Lead to a $n^{\Theta(\lg n)}$ algorithm