# Design and Analysis of Algorithm

Lecture-5:

# Contents

# Complexity

$$T(n) = \begin{cases} 0 & if\ n = 1 \\ 1 & if\ n = 2 \\ 2T\left(\frac{n}{2}\right) + 2 & if\ n > 2 \end{cases}$$

$$solution\text{:}\ \frac{3n}{2} - 2$$

$$Complexity = O(n)$$

In terms of storage ,$MaxMin$ using divide and conquer is worse than the straightforward algorithm because it requires stack space for $i, j, \max, \min, max1$, and $\min 1$.

Given $n$ elements, there will be $\lfloor \log_2 n \rfloor + 1$ levels of recursion and we need to save seven values for each recursive call including $return$

# Binary search algorithm

**Input:** An array $a$ of $n$ elements and the number to be searched (*say* $x$) in the array
**Output:** Return position of $x$, if $x$ is found in the array

Let Small(P) be true if n = 1.

In this case, S(P)will take the value $i$ if $x = a[i]$,

Otherwise it will take the value 0

If $P$ has more than one element, it should be divided into a sub-problems

# Binary search algorithm

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
                  return BinSrch(a, i, mid − 1, x);
              else return BinSrch(a, mid + 1, l, x);
    }
}
```

A=[2]

X=3

A=[2]

X=2

# Binary search algorithm

**Search Key: 42**

| index | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value | -4 | 2 | 7 | 10 | 15 | 20 | 22 | 25 | 30 | 36 | 42 | 50 | 56 | 68 | 85 | 92 | 103 |

| | | | | | |
|---|---|---|---|---|---|
| *i* | *i* | *mid* | *l* | *l* | |

***Key found at index location 10***

# Complexity of binary search

Let $T(n)$ be the time required to find element x in the given array of n element using binary search

$$T(n) = \begin{cases} b & if\ n = 1 \\ T\left(\frac{n}{2}\right) + c & otherwise \end{cases}$$

$$T(n) = T(n/2) + C$$
$$= T(n/2^2) + C + C$$
$$\downarrow k \text{ time}$$
$$= T\left(\frac{n}{2^k}\right) + kC$$
$$= T(1) + \log_2 n \cdot C$$
$$= b + C \cdot \log n$$
$$\boxed{T(n) = O(\log n)}$$

# Comparison with linear search

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value | -4 | 2 | 7 | 10 | 15 | 20 | 22 | 25 | 30 | 36 | 42 | 50 | 56 | 68 | 85 | 92 | 103 |

Time complexity: O(n)

*Note: Linear search is better than binary search if data is not sorted*

# Assignment

Write an algorithm and compute its time complexity.

Input: A sorted Array of $n$ elements
Output: Find two elements a && b such that a + b=200

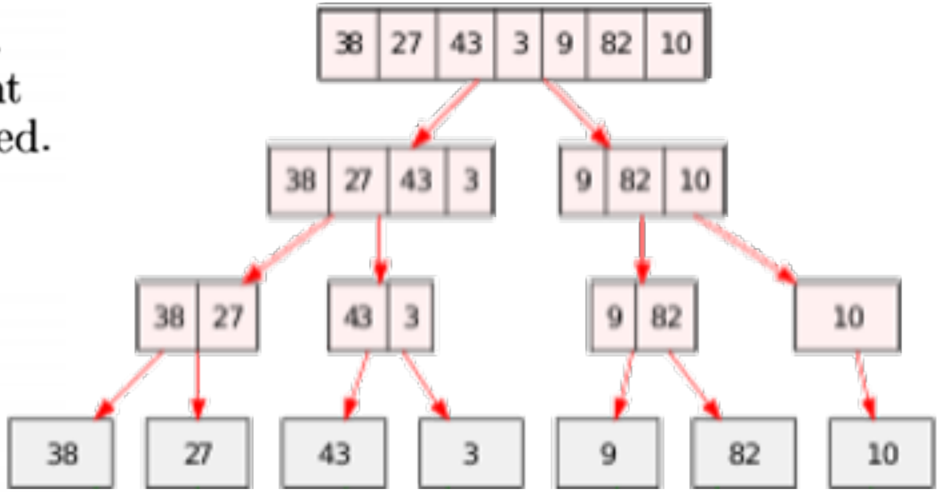
# Merge Sort

Given a sequence of n elements $a[1], \ldots, a[n]$. The general idea is to imagine them split into two sets $a[1], \ldots, a[\lfloor \frac{n}{2} \rfloor]$ and $a[\lfloor \frac{n}{2} \rfloor + 1], \ldots, a[n]$.

Each set is individually sorted , and the resulting sorted sequences are merged to produce a single sorted sequence of $n$ elements

# Merge Sort Algorithm

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```



# Merge Algorithm

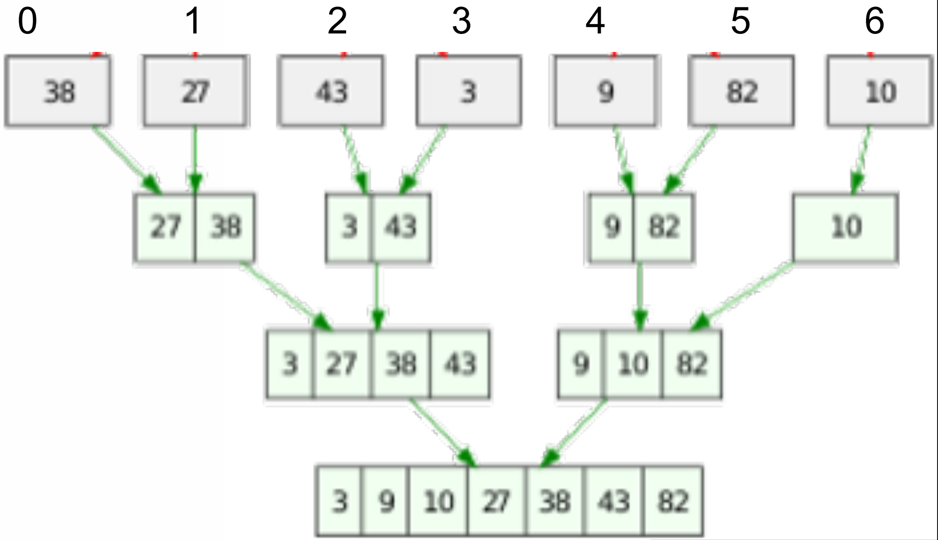
**Algorithm** Merge$(low, mid, high)$
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
    $h := low$; $i := low$; $j := mid + 1$;
    **while** $((h \leq mid)$ **and** $(j \leq high))$ **do**
    {
        **if** $(a[h] \leq a[j])$ **then**
        {
            $b[i] := a[h]$;$h := h + 1$;
        }
        **else**
        {
            $b[i] := a[j]$; $j := j + 1$;
        }
        $i := i + 1$;
    }

# Merge Algorithm

```
if (h > mid) then
    for k := j to high do
    {
        b[i] := a[k]; i := i + 1;
    }
else
    for k := h to mid do
    {
        b[i] := a[k]; i := i + 1;
    }
for k := low to high do a[k] := b[k];
}
```

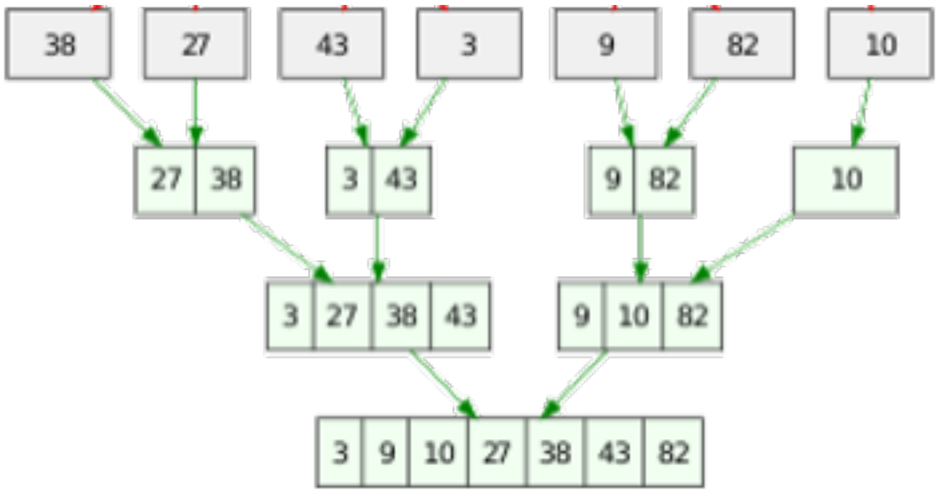
# Merge Algorithm

**Algorithm** Merge$(low, mid, high)$
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
  $h := low$; $i := low$; $j := mid + 1$;
  **while** $((h \le mid)$ **and** $(j \le high))$ **do**
  {
    **if** $(a[h] \le a[j])$ **then**
    {
      $b[i] := a[h]; h := h + 1;$
    }
    **else**
    {
      $b[i] := a[j]; j := j + 1;$
    }
    $i := i + 1;$
  }



# Merge Algorithm

```
if (h > mid) then
        for k := j to high do
        {
                b[i] := a[k]; i := i + 1;
        }
else
        for k := h to mid do
        {
                b[i] := a[k]; i := i + 1;
        }
for k := low to high do a[k] := b[k];
}
```



# Complexity

$$T(n) = \begin{cases} b & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & otherwise \end{cases}$$

$$T(n) = O(nlog_2n)$$

***<u>Note:</u>***

1. Merge sort is good for large sized array
2. It is not in-place sorting.
3. It is the stable sorting algorithm