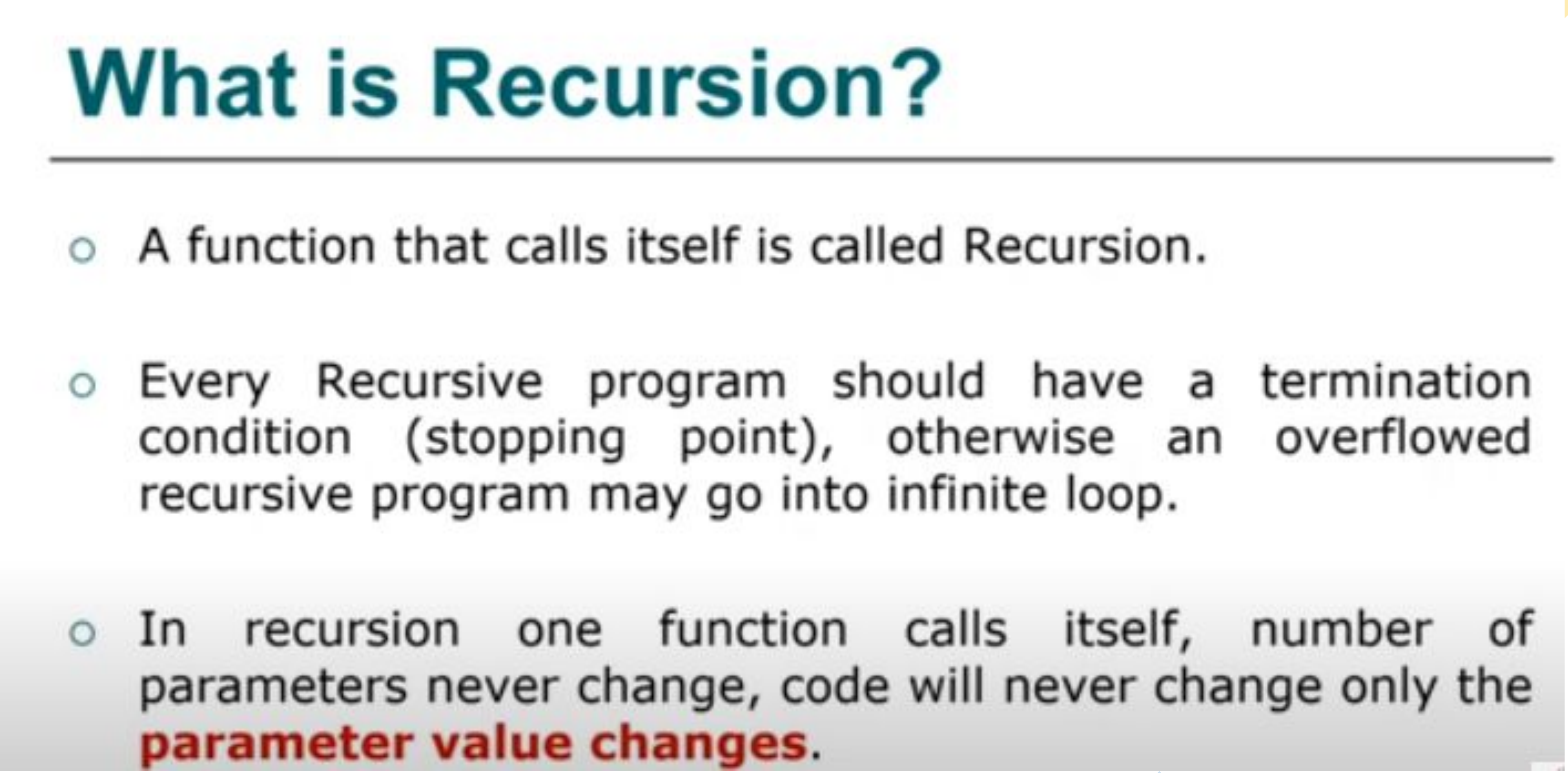
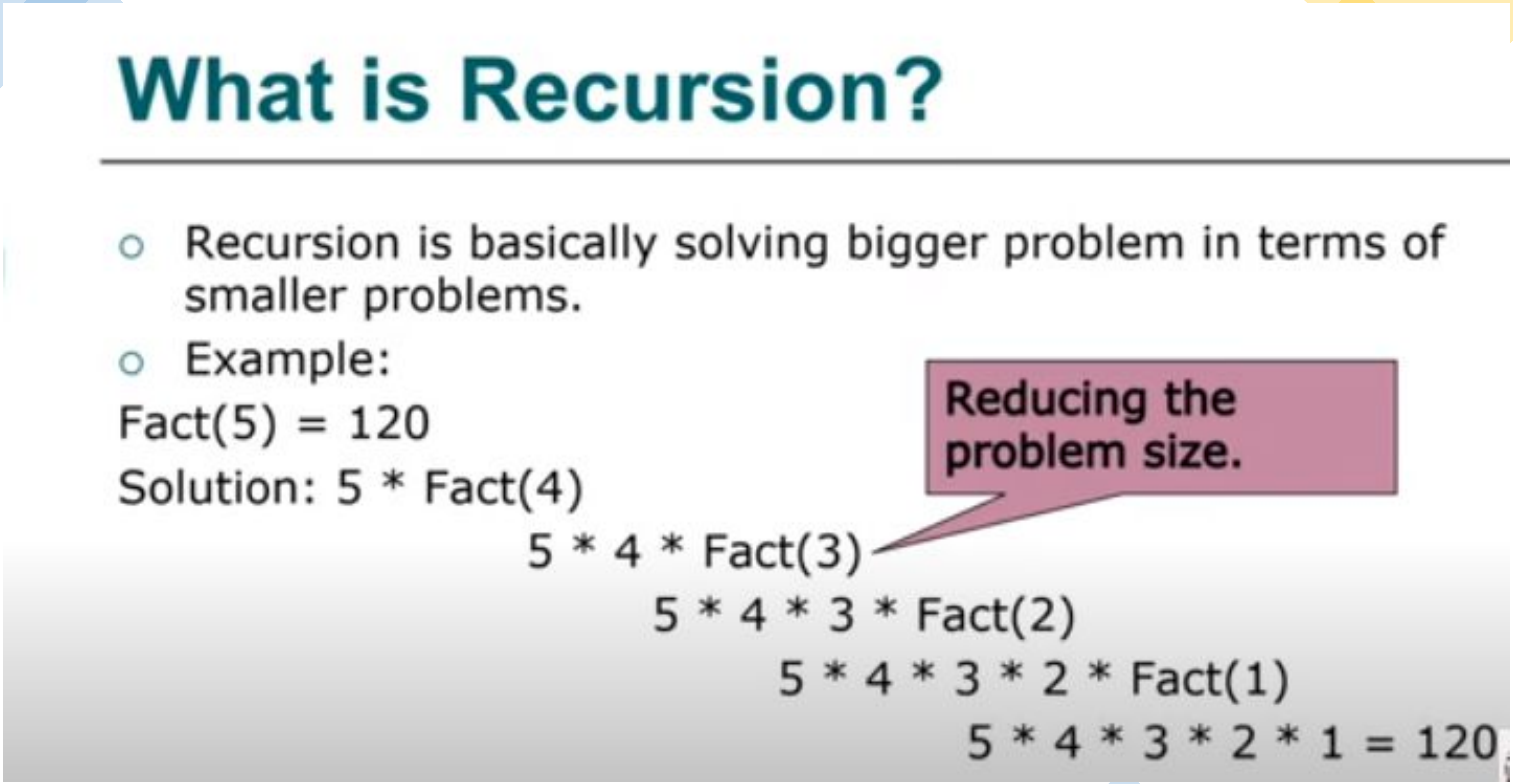
Module No.2: Recursive Algorithms



What is Recursion?

- A function that calls itself is called Recursion.
- Every Recursive program should have a termination condition (stopping point), otherwise an overflowed recursive program may go into infinite loop.
- In recursion one function calls itself, number of parameters never change, code will never change only the **parameter value changes**.



What is Recursion?

- Recursion is basically solving bigger problem in terms of smaller problems.
- Example:

Fact(5) = 120

Solution: 5 * Fact(4)

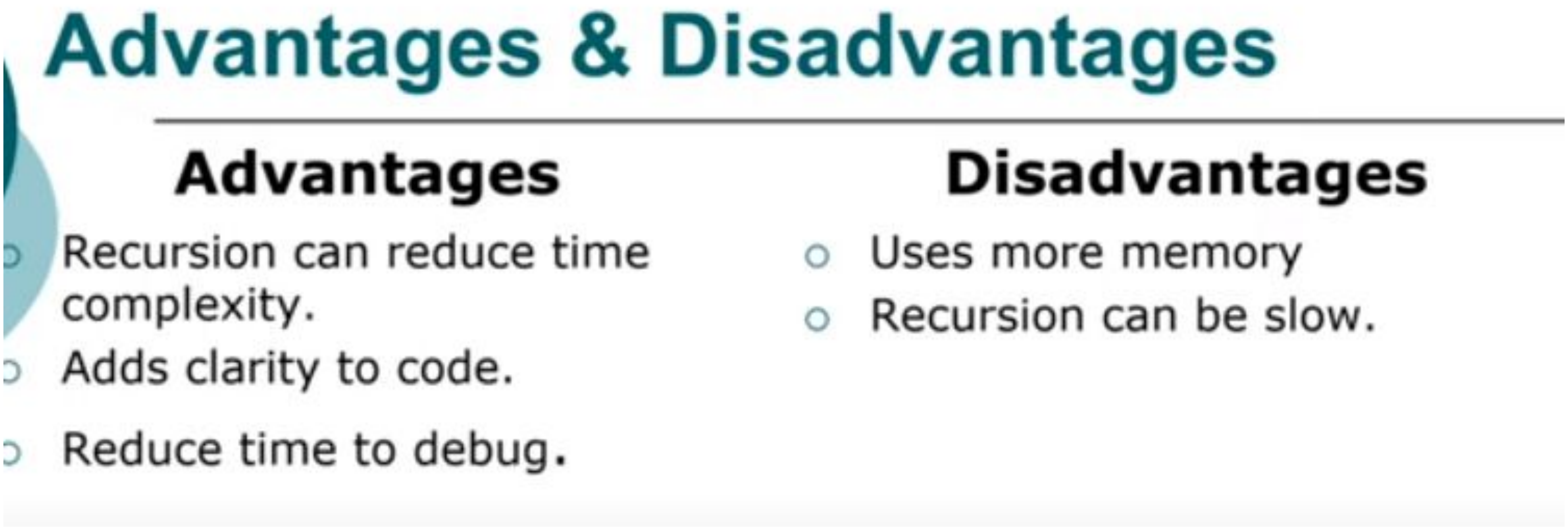
5 * 4 * Fact(3)

5 * 4 * 3 * Fact(2)

5 * 4 * 3 * 2 * Fact(1)

5 * 4 * 3 * 2 * 1 = 120

Reducing the problem size.



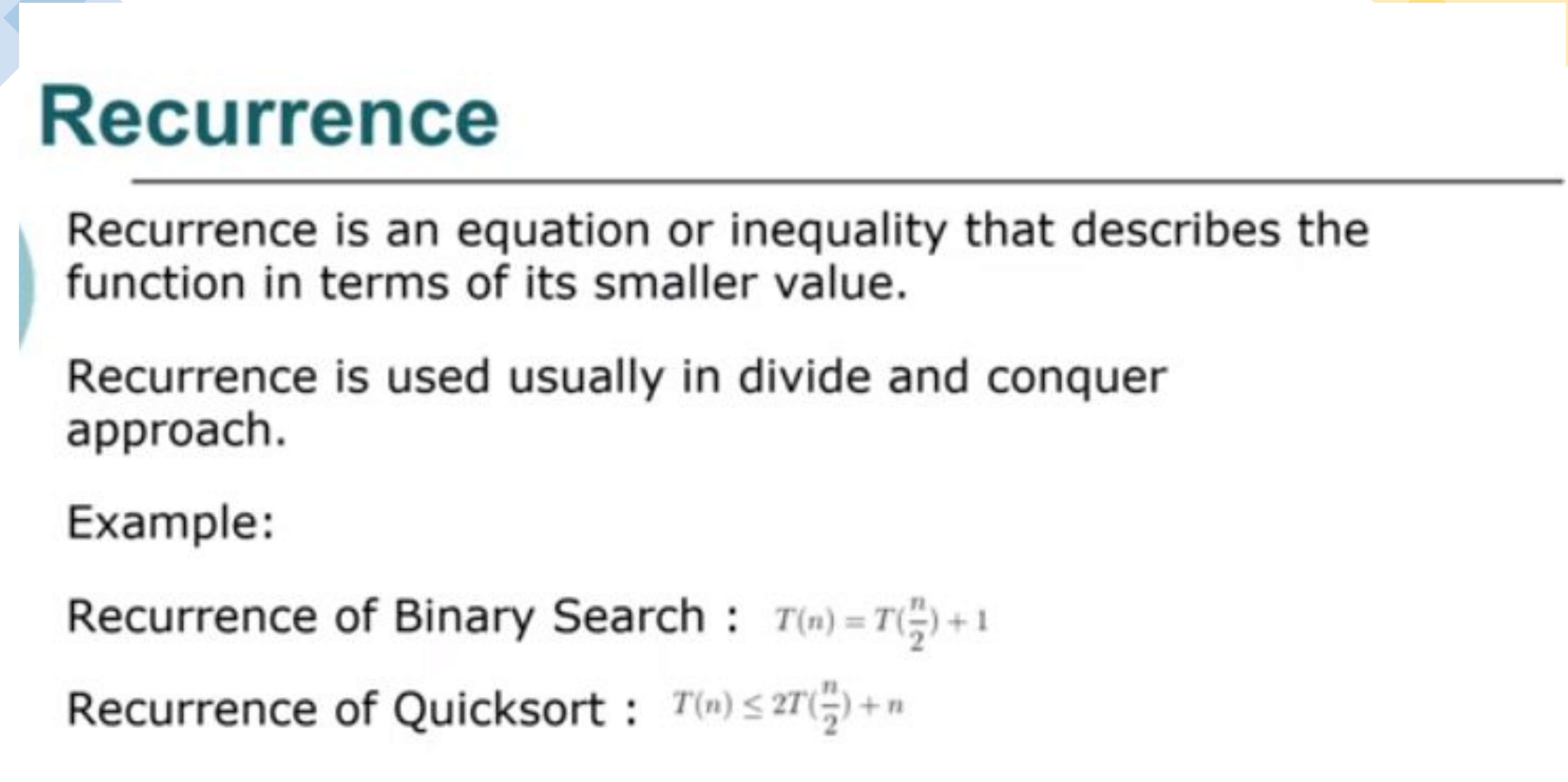
Advantages & Disadvantages

Advantages

- Recursion can reduce time complexity.
- Adds clarity to code.
- Reduce time to debug.

Disadvantages

- Uses more memory
- Recursion can be slow.



Recurrence

Recurrence is an equation or inequality that describes the function in terms of its smaller value.

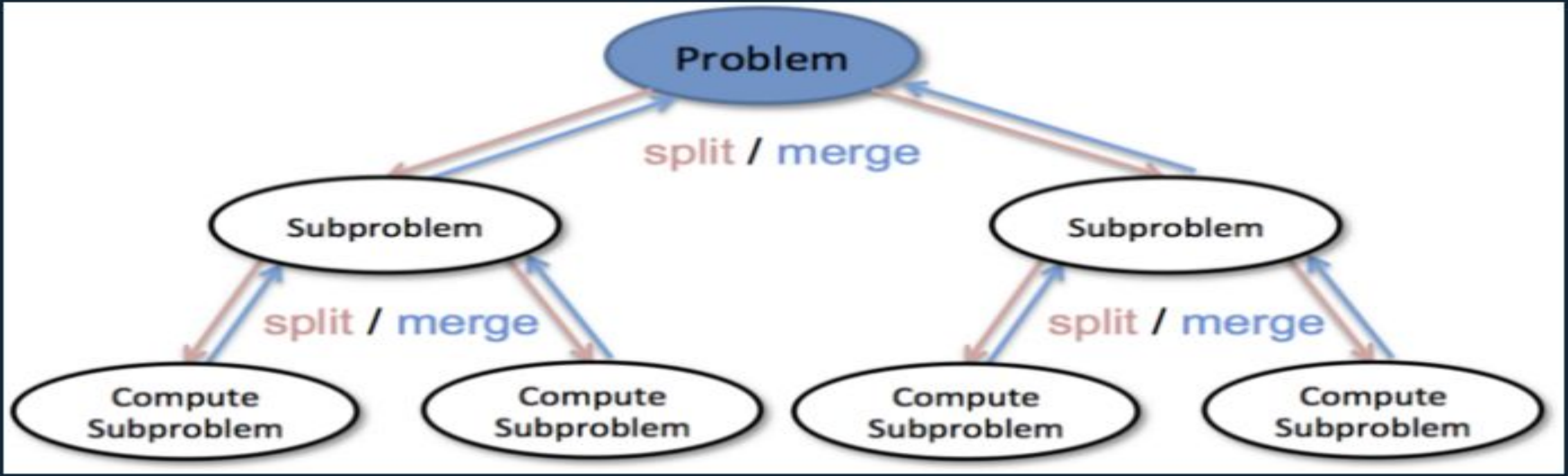
Recurrence is used usually in divide and conquer approach.

Example:

Recurrence of Binary Search : $T(n) = T(\frac{n}{2}) + 1$

Recurrence of Quicksort : $T(n) \leq 2T(\frac{n}{2}) + n$

Divide and Conquer Algorithms

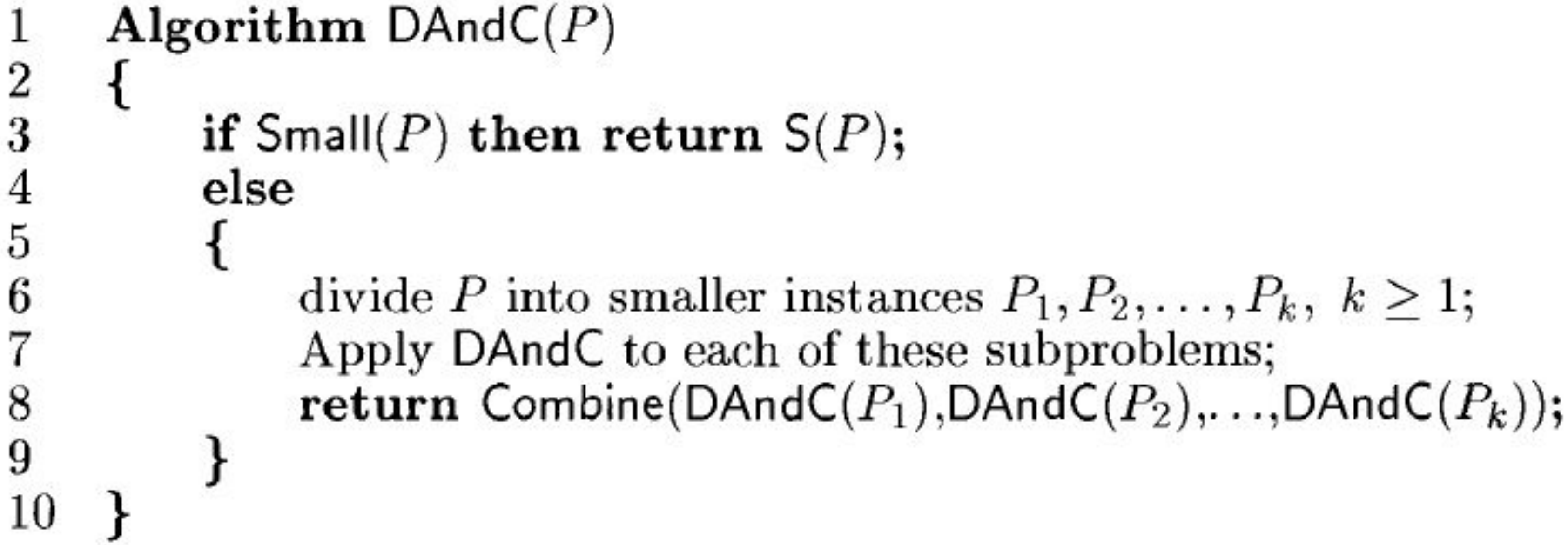


Divide:- The problem into a number of subproblems.

Conquer:- The subproblem by solving them recursively.

Combine:- it define as the solutions to the subproblem into the solution for the original problem

Divide and Conquer



```
Algorithm DAndC($P$)
{
    if Small($P$) then return S($P$);
    else
    {
        divide $P$ into smaller instances $P_1, P_2, \ldots, P_k,\ k \geq 1$;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC($P_1$),DAndC($P_2$),. . .,DAndC($P_k$));
    }
}
```

Following, these are few standard algorithms that are Divide and Conquer algorithms.

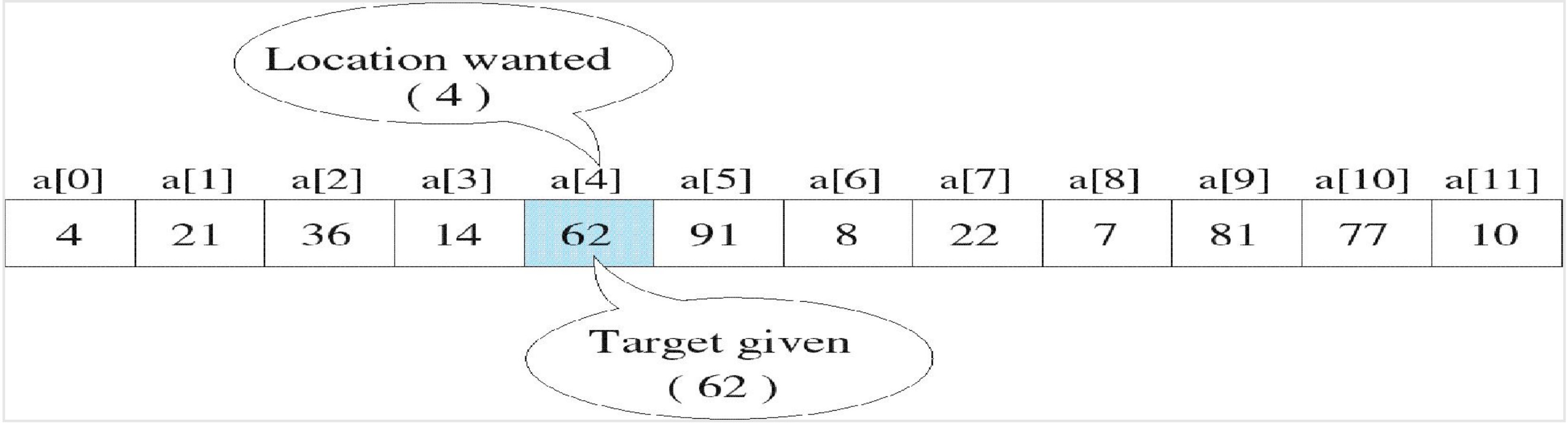
- BINARY SEARCH
- QUICK SORT
- MERGE SORT

SEARCHING:

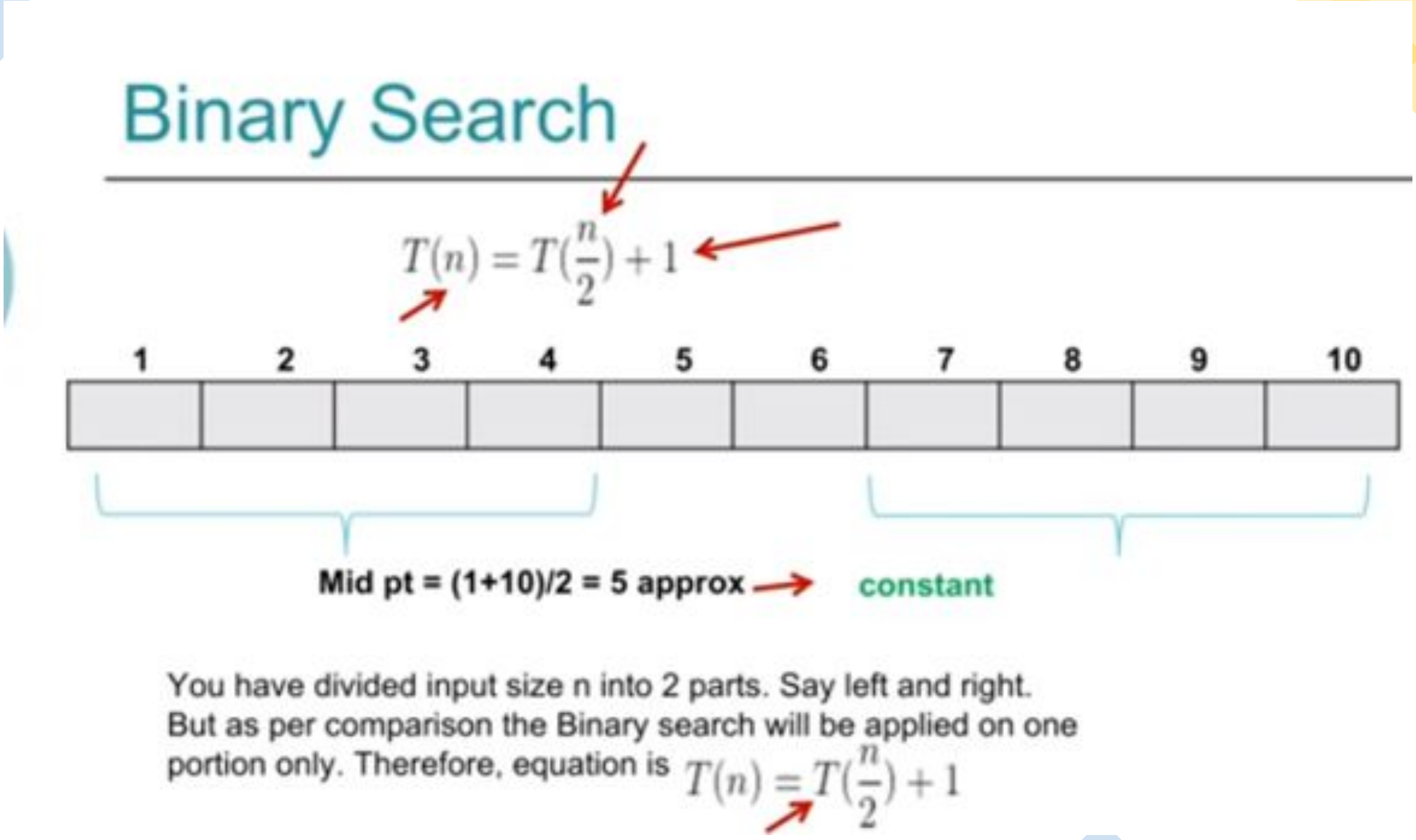
Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

Example: Searching an array finds the index of first element in an array containing that value



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	21	36	14	62	91	8	22	7	81	77	10



Binary Search

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

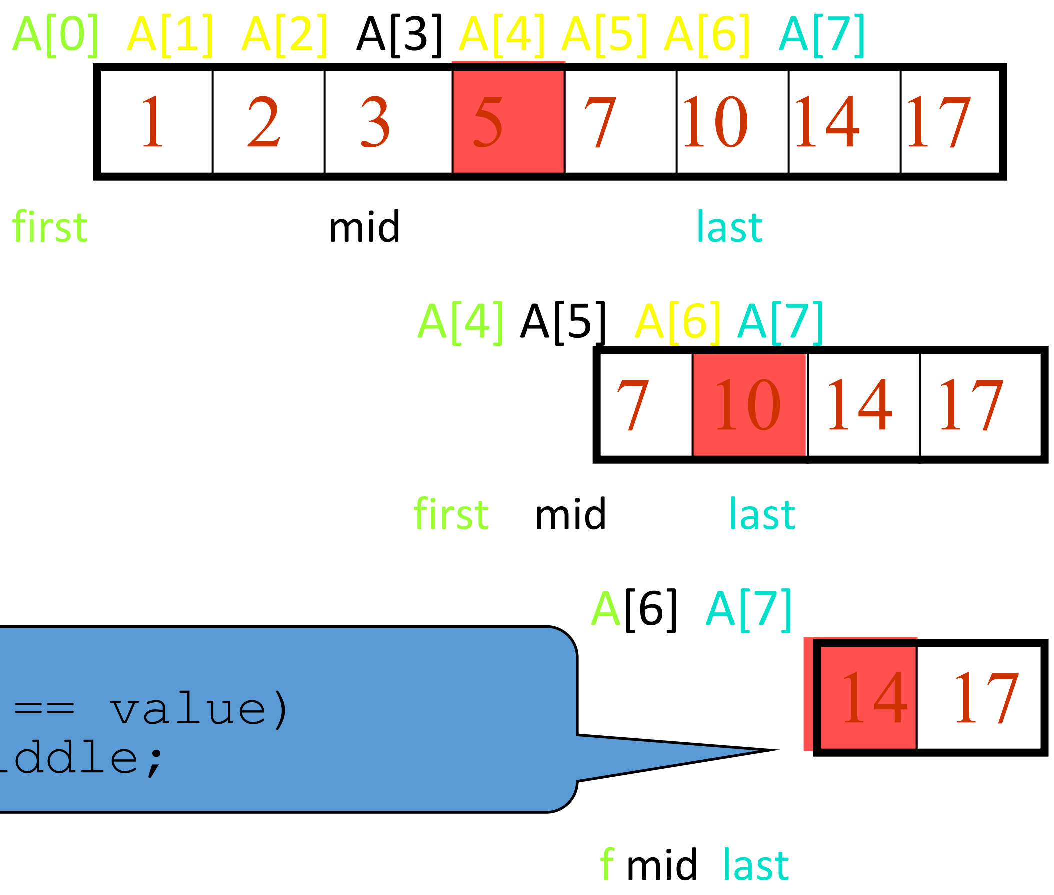
Mid pt = (1+10)/2 = 5 approx → constant

You have divided input size n into 2 parts. Say left and right. But as per comparison the Binary search will be applied on one portion only. Therefore, equation is $T(n) = T(\frac{n}{2}) + 1$

successful

Example: binary search

- 14 ?

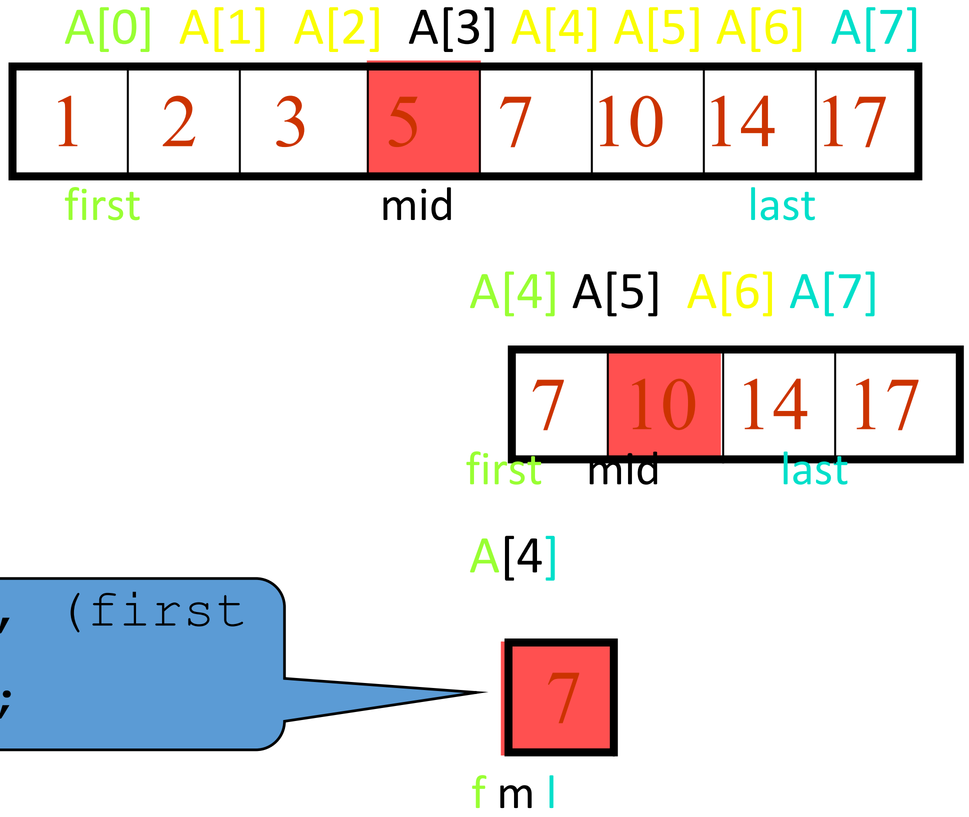


```
In this case,
(data[middle] == value)
     return middle;
```

Example: binary search

unsuccessful

- 8 ?



A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

1	2	3	5	7	10	14	17

first mid last

A[4] A[5] A[6] A[7]

7	10	14	17

first mid last

A[4]

7

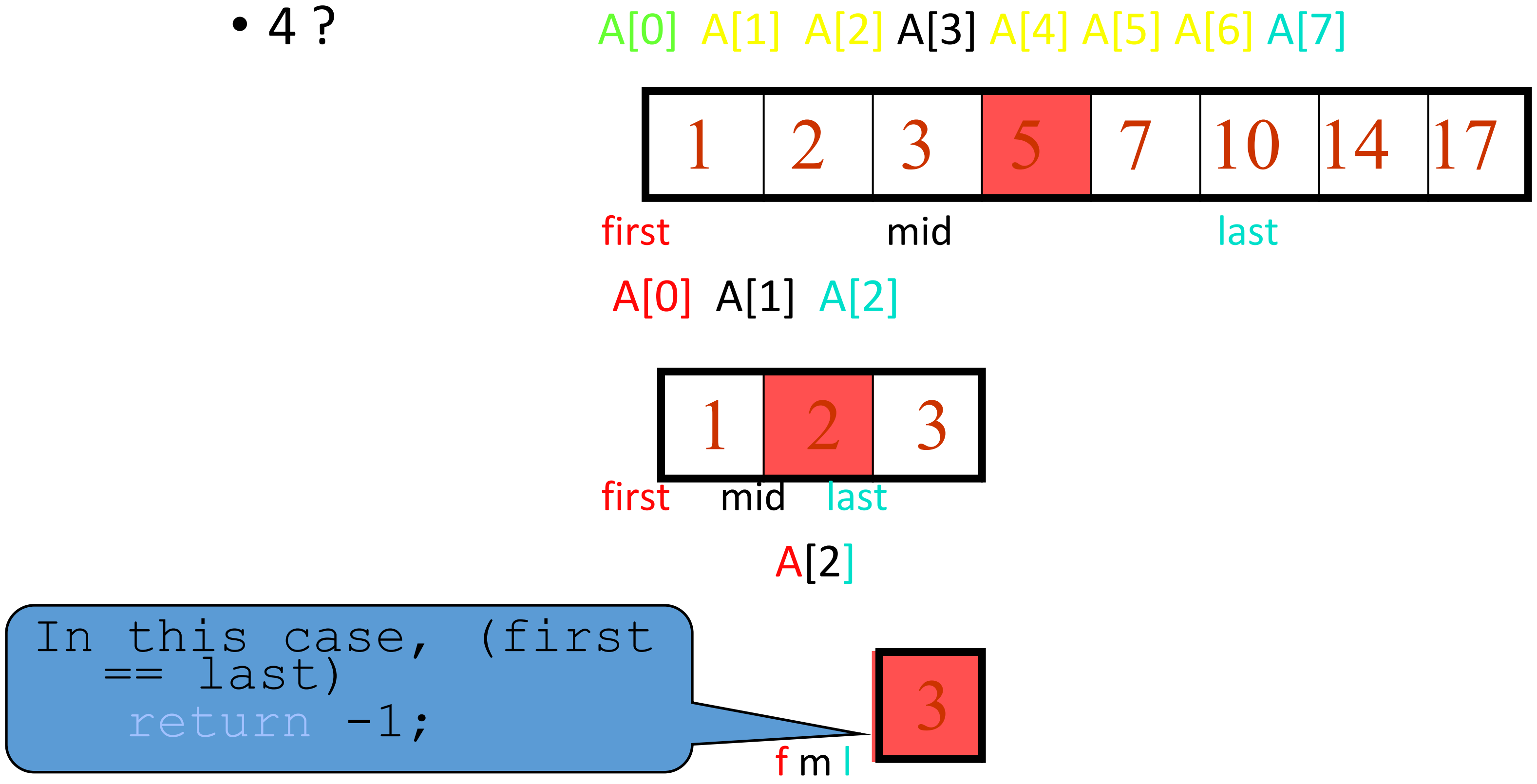
f m l

```
In this case, (first
   == last)
    return -1;
```

Example: binary search

unsuccessful

- 4 ?



A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

1	2	3	5	7	10	14	17

first mid last

A[0] A[1] A[2]

1	2	3

first mid last

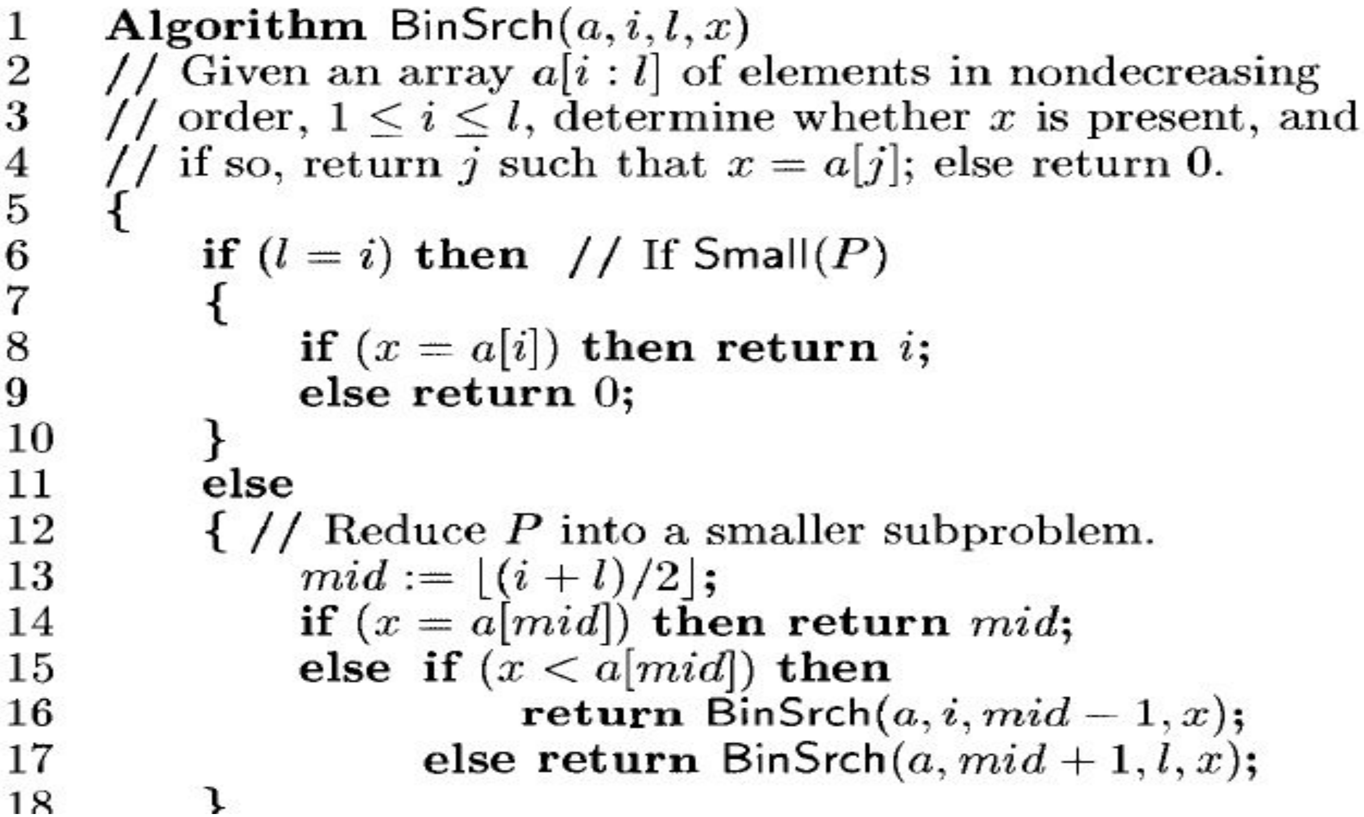
A[2]

3

f m l

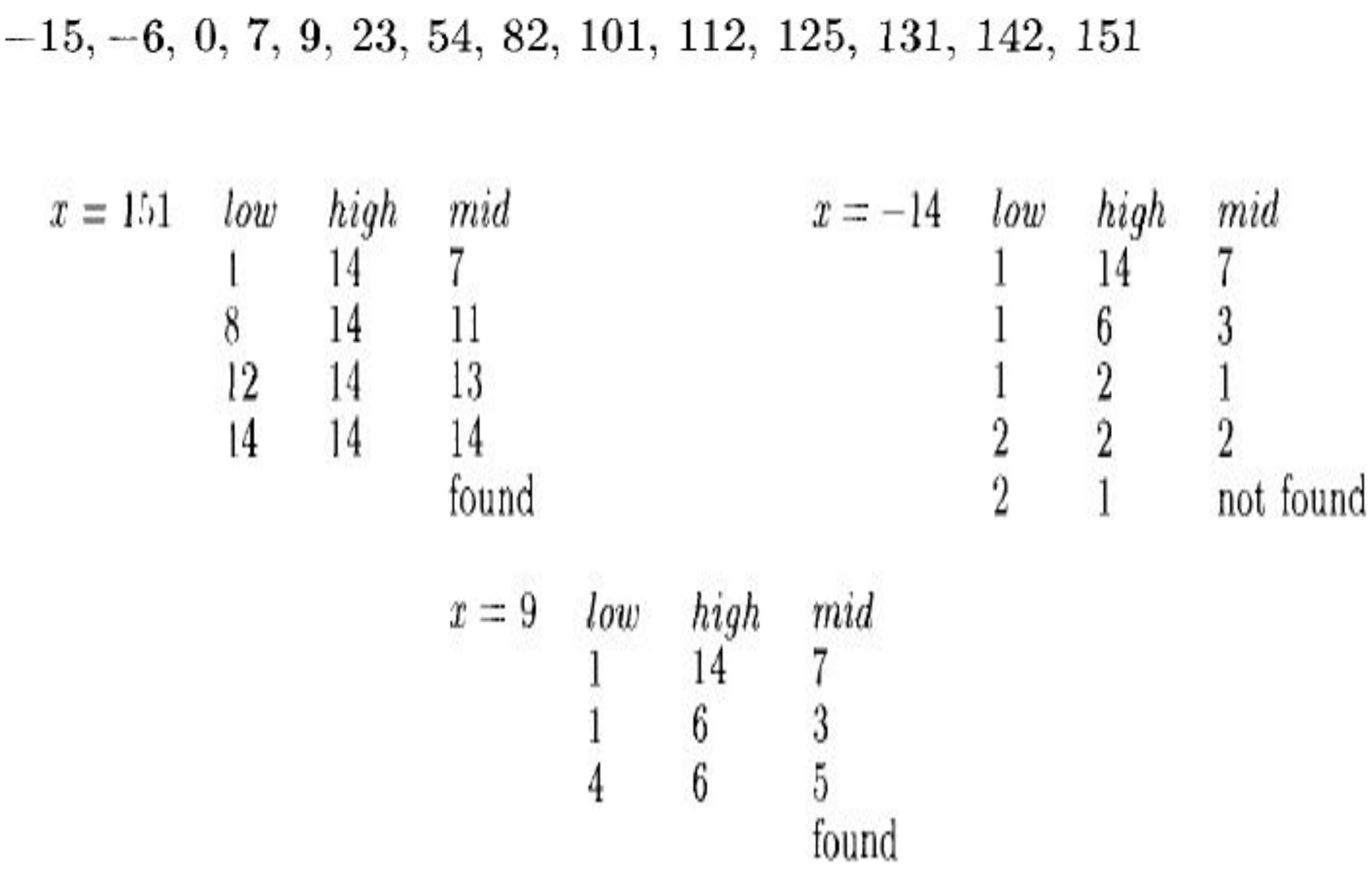
```
In this case, (first
   == last)
   return -1;
```

Recursive Algorithm:



```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, 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.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                  return BinSrch($a, i, mid - 1, x$);
             else return BinSrch($a, mid + 1, l, x$);
    }
```

Example:



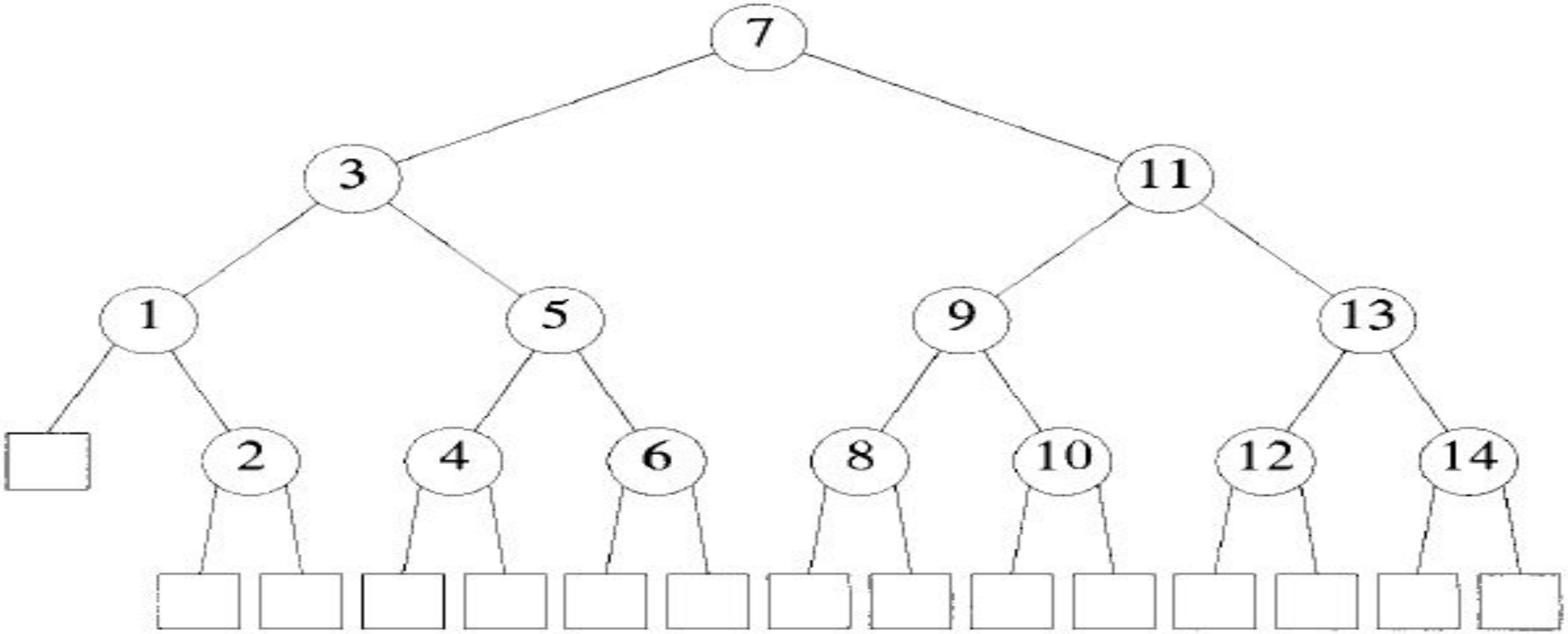
$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$

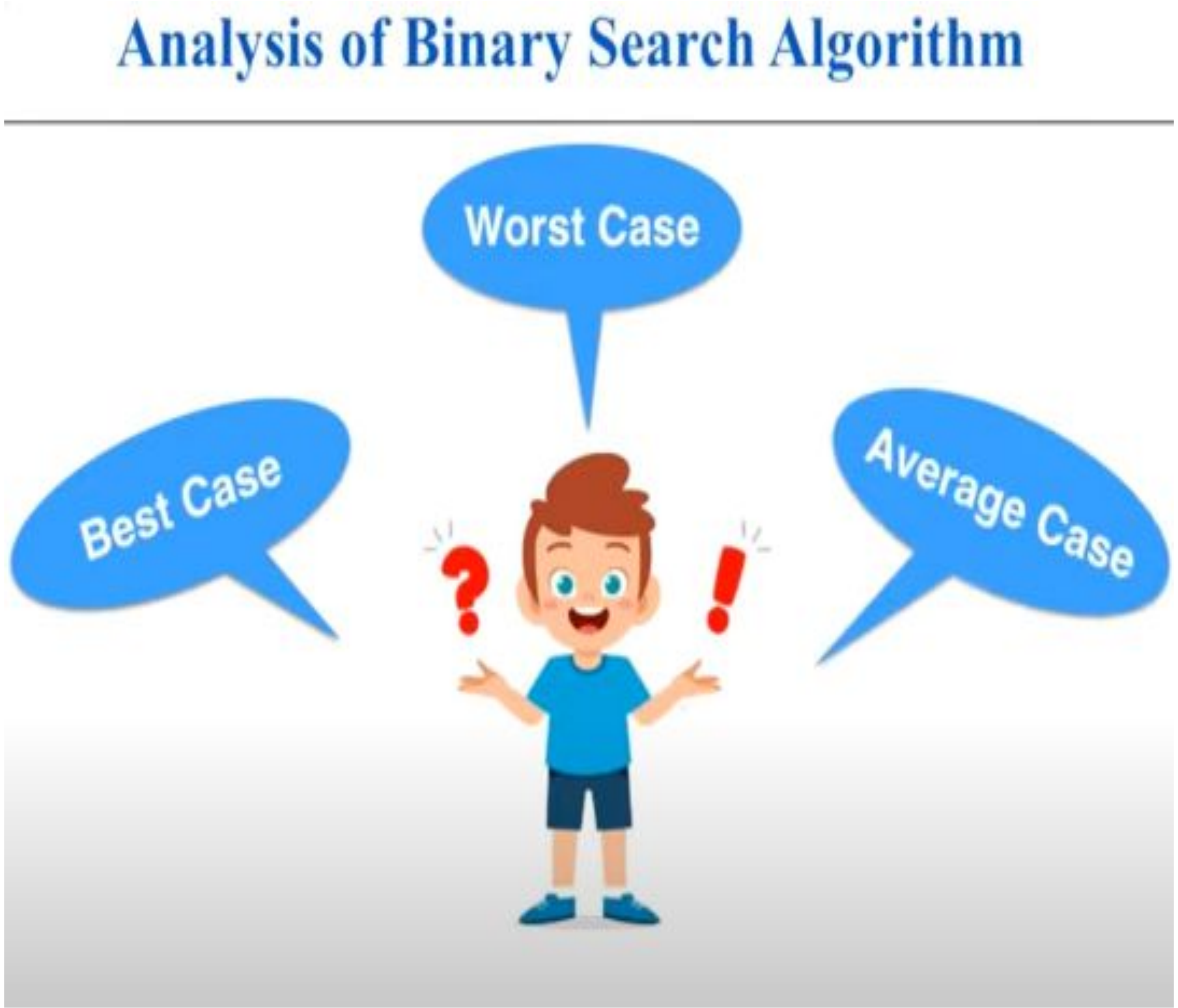
$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

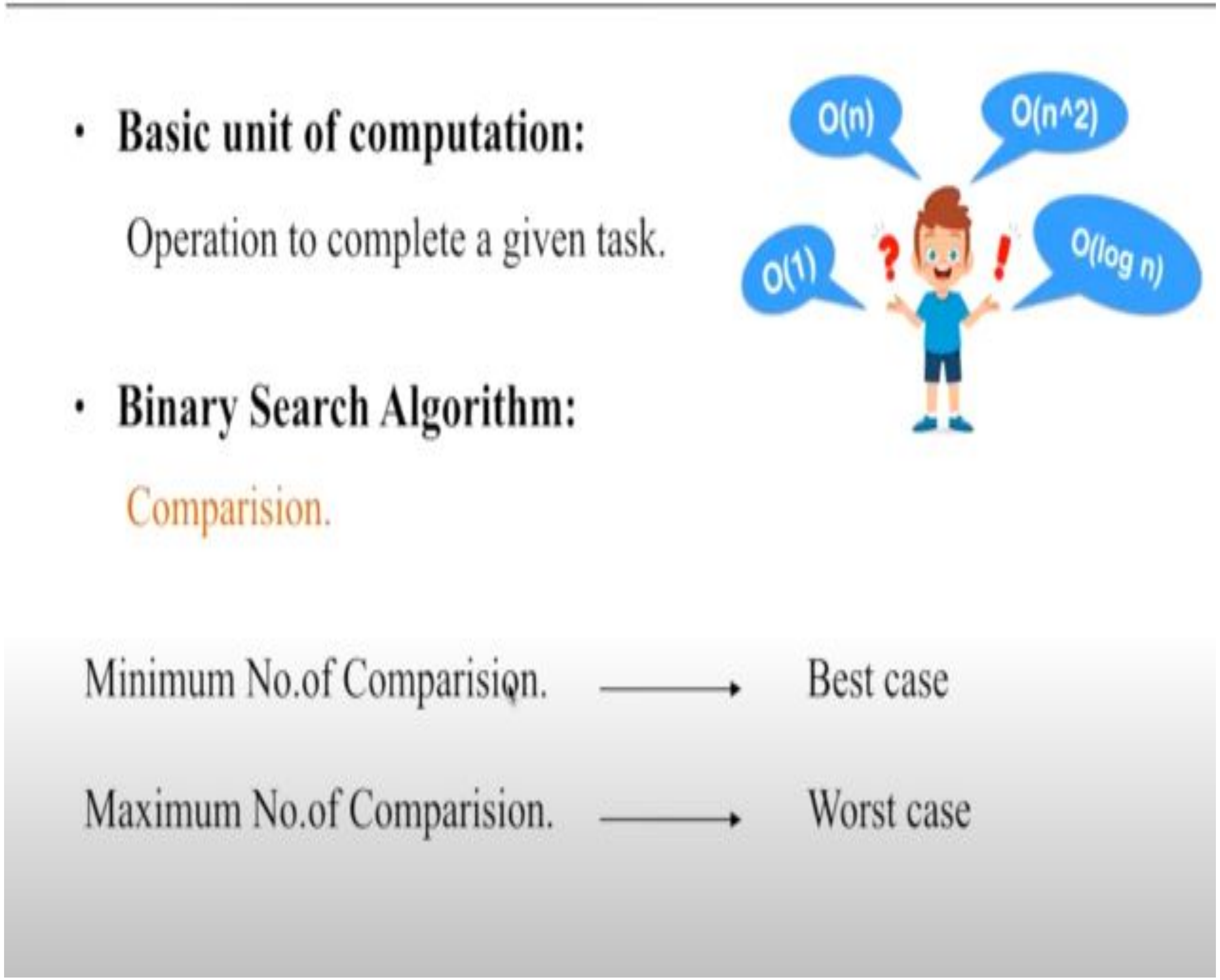
$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4





Analysis of Binary Search Algorithm



- **Basic unit of computation:**

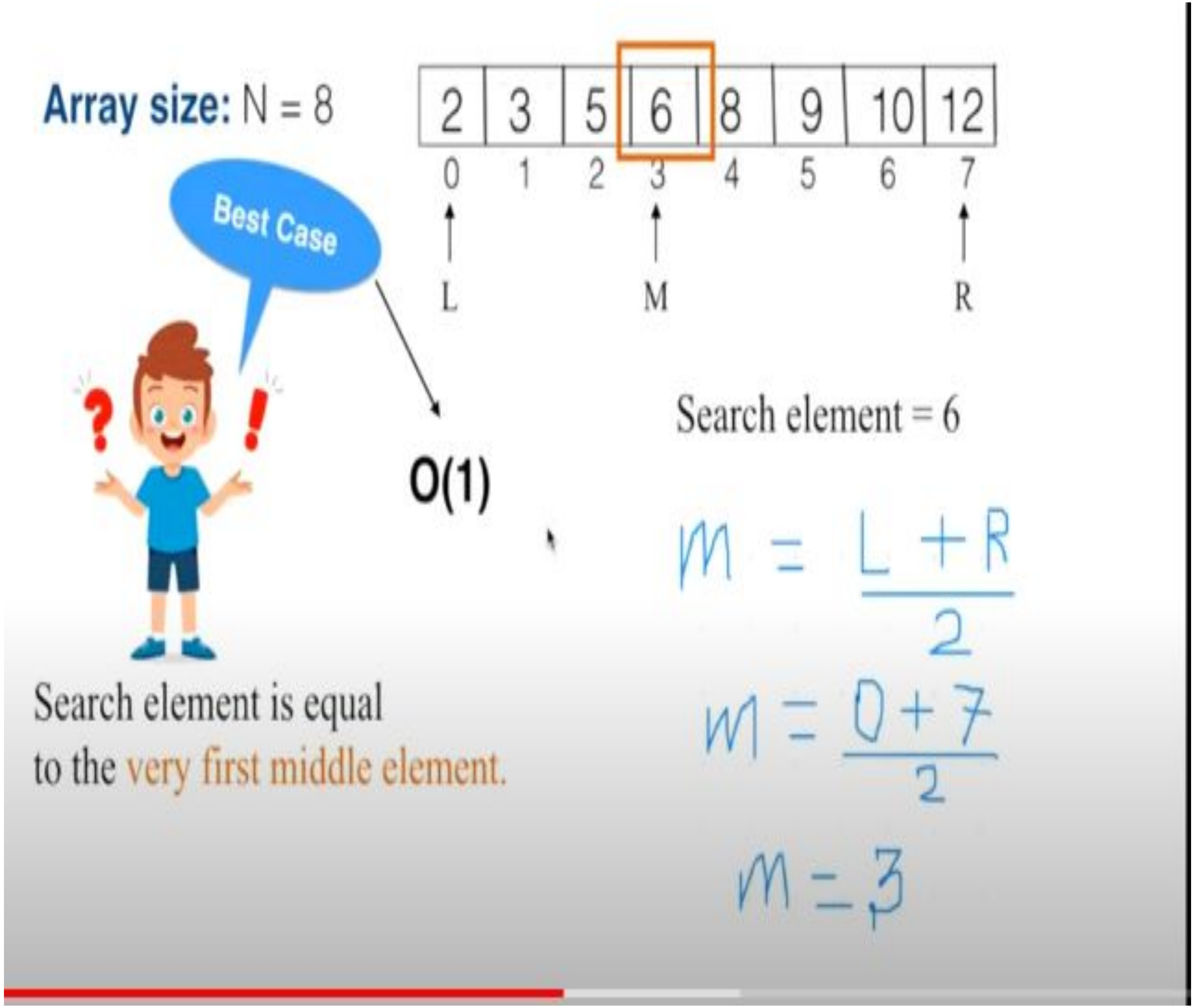
  Operation to complete a given task.

- **Binary Search Algorithm:**

  Comparision.

Minimum No.of Comparision. ⟶ Best case

Maximum No.of Comparision. ⟶ Worst case



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

O(1)

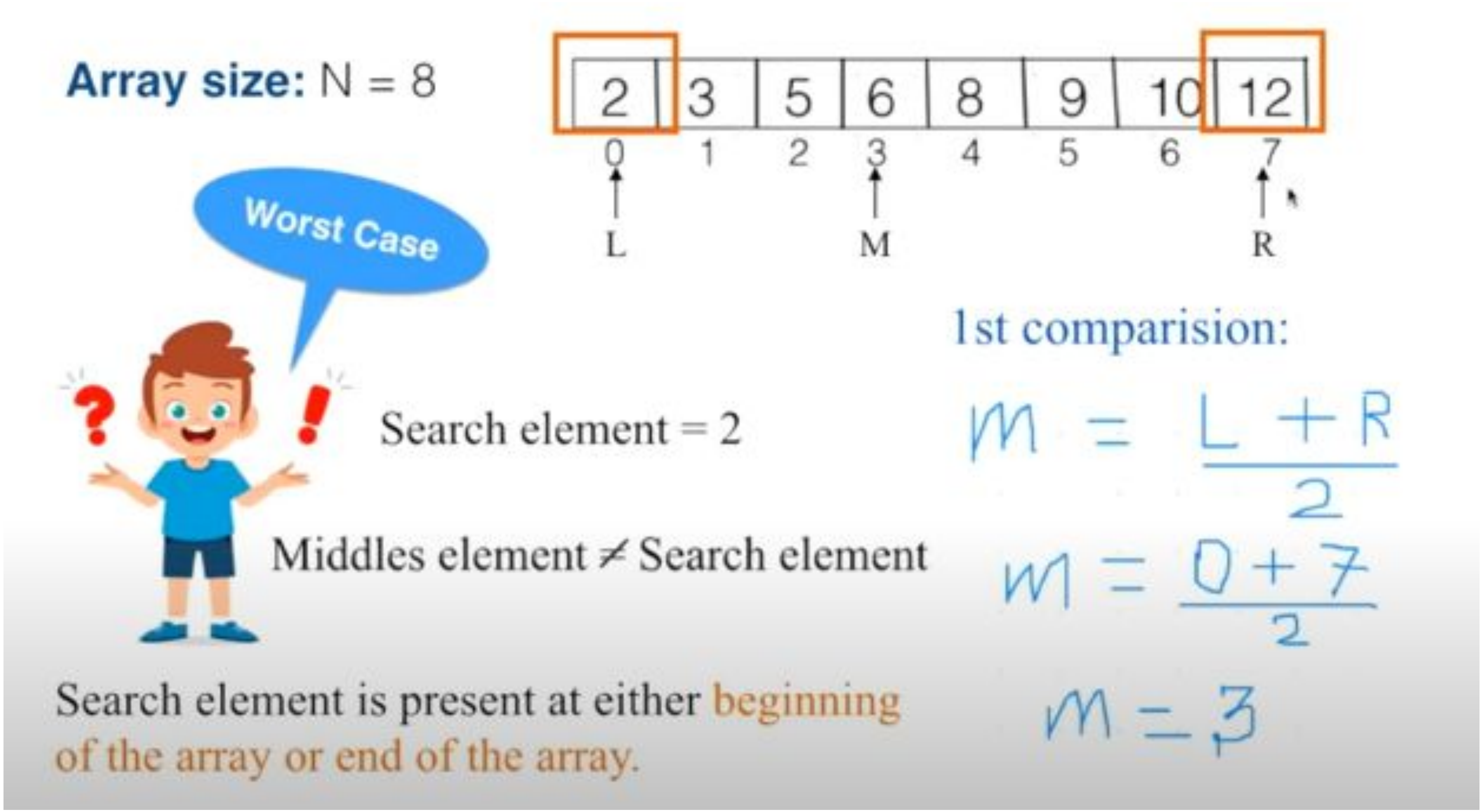
Search element = 6

$$m = \frac{L + R}{2}$$

$$m = \frac{0 + 7}{2}$$

$$m = 3$$

Search element is equal to the very first middle element.



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

Search element = 2

Middles element ≠ Search element

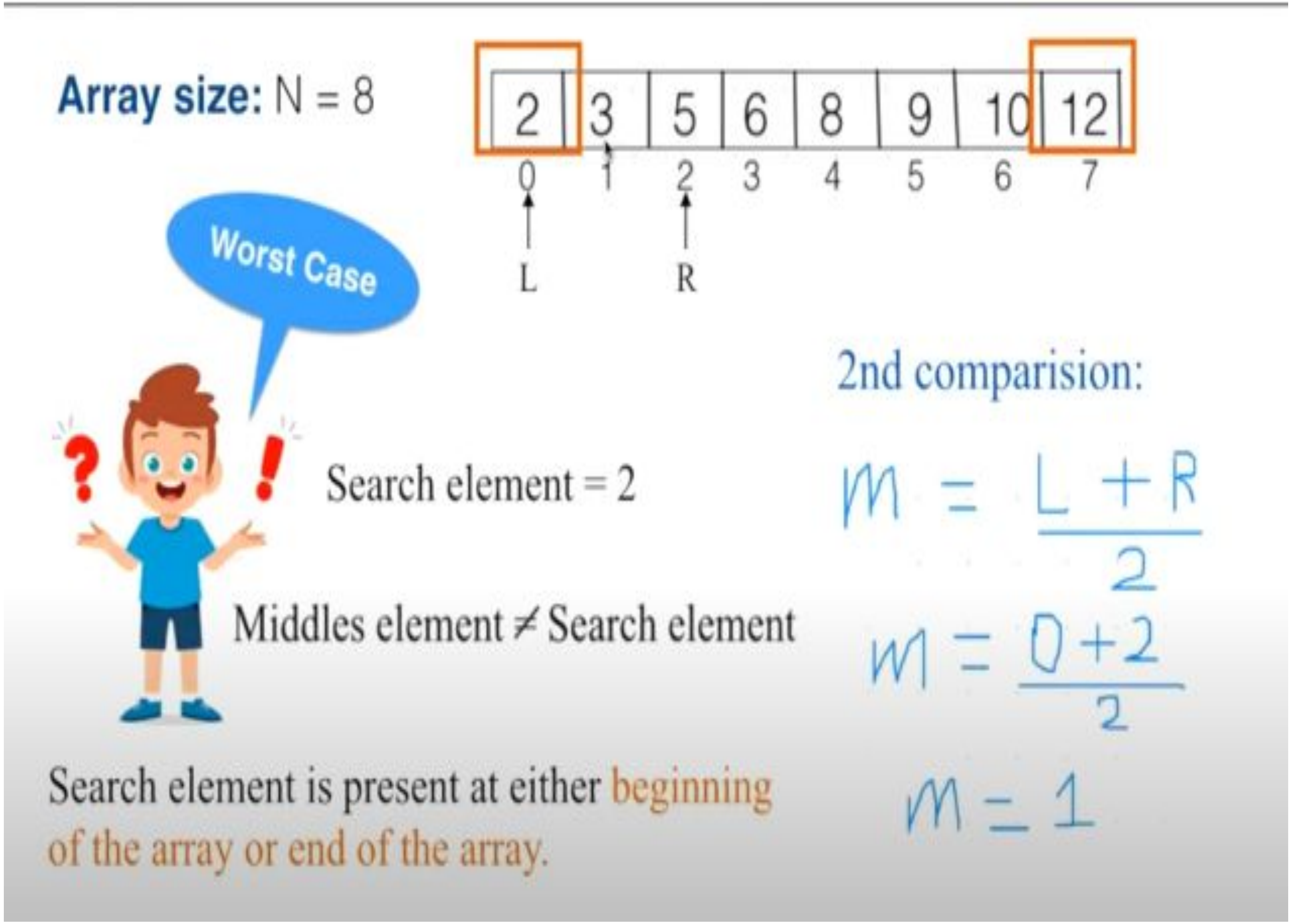
Search element is present at either beginning of the array or end of the array.

1st comparision:

$$m = \frac{L + R}{2}$$

$$m = \frac{0 + 7}{2}$$

$$m = 3$$



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

Search element = 2

Middles element ≠ Search element

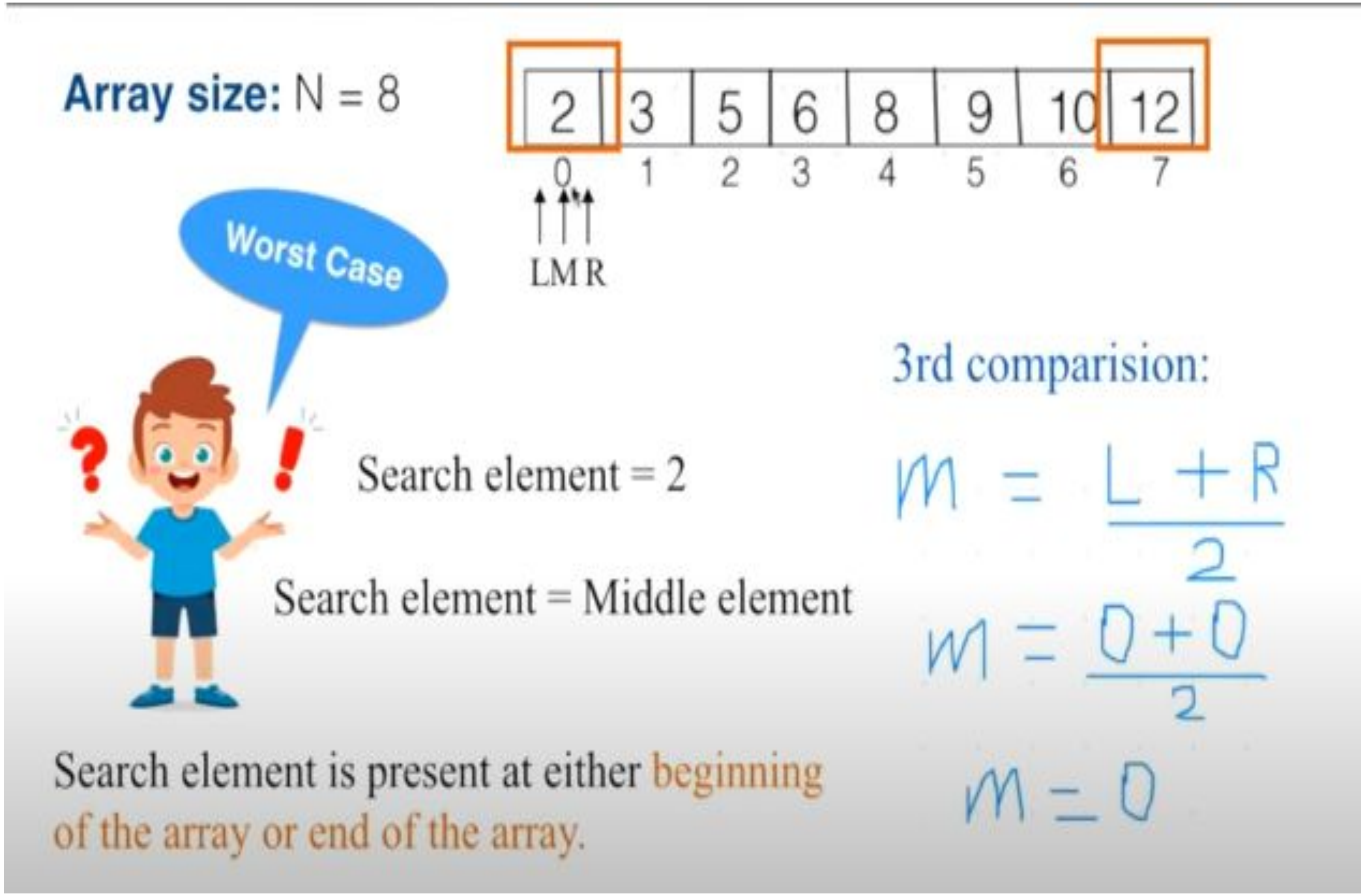
Search element is present at either beginning of the array or end of the array.

2nd comparision:

$$m = \frac{L + R}{2}$$

$$m = \frac{0+2}{2}$$

$$m = 1$$



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

Search element = 2

Search element = Middle element

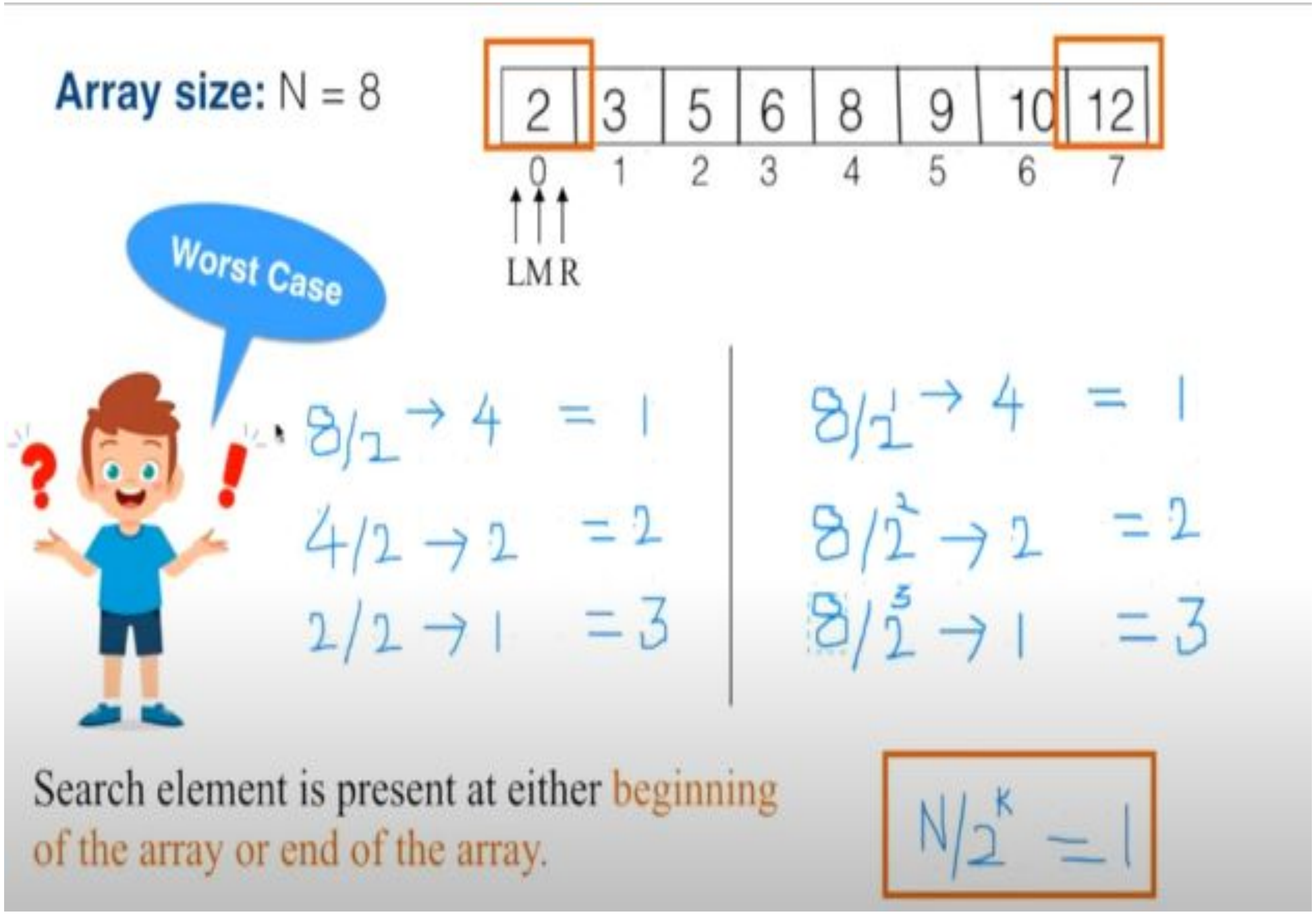
Search element is present at either beginning of the array or end of the array.

3rd comparision:

$$m = \frac{L + R}{2}$$

$$m = \frac{0 + 0}{2}$$

$$m = 0$$



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

$8/2 \rightarrow 4 = 1$

$4/2 \rightarrow 2 = 2$

$2/2 \rightarrow 1 = 3$

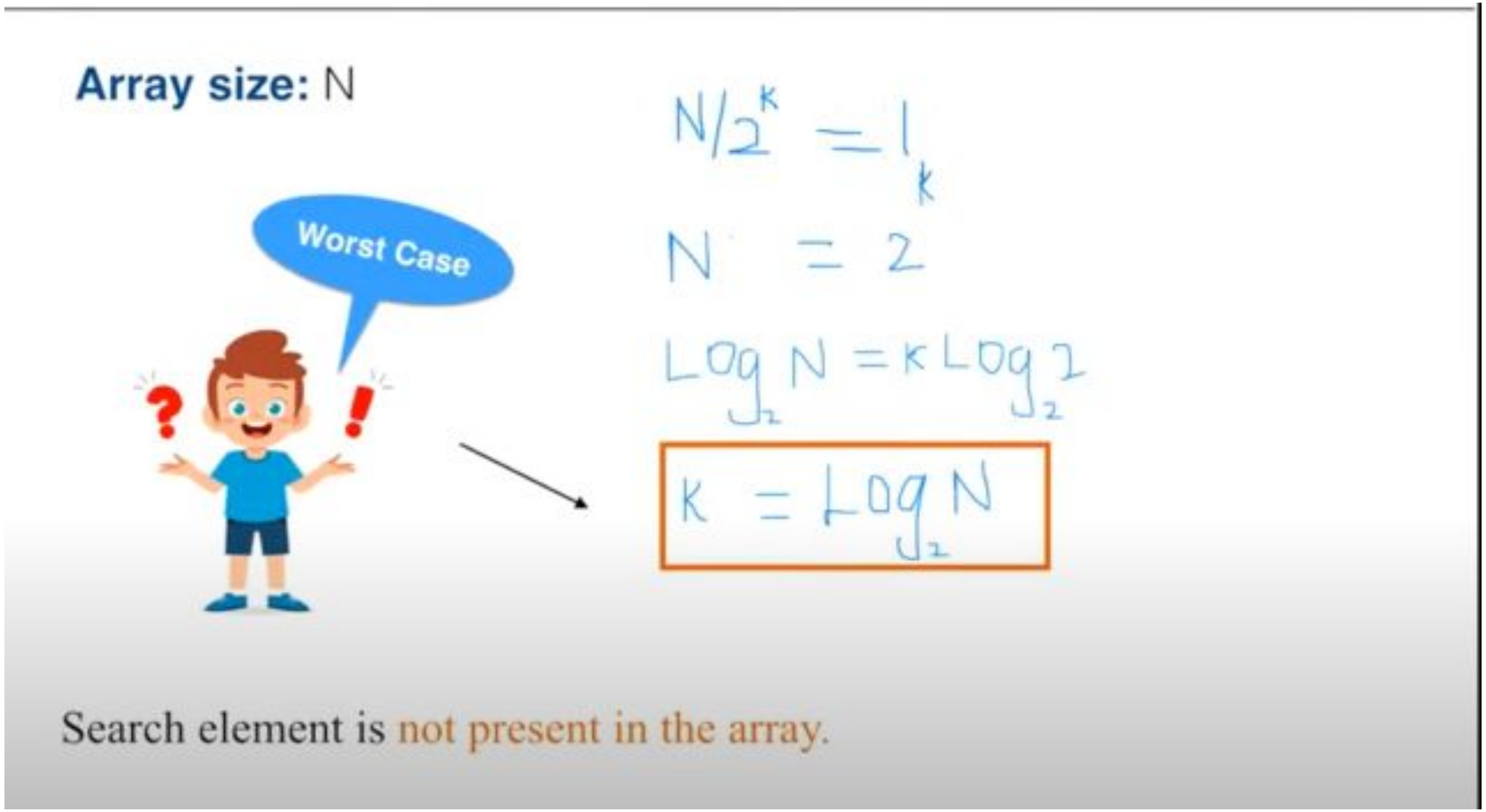
$8/2^1 \rightarrow 4 = 1$

$8/2^2 \rightarrow 2 = 2$

$8/2^3 \rightarrow 1 = 3$

Search element is present at either beginning of the array or end of the array.

$\boxed{N/2^K = 1}$



Array size: N

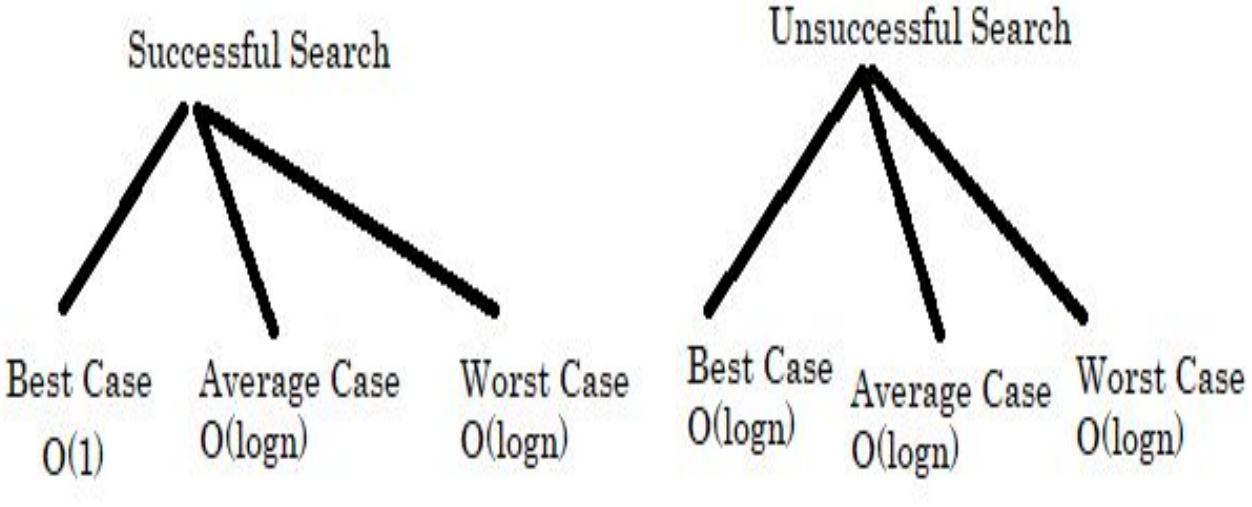
$$N/2^K = 1$$

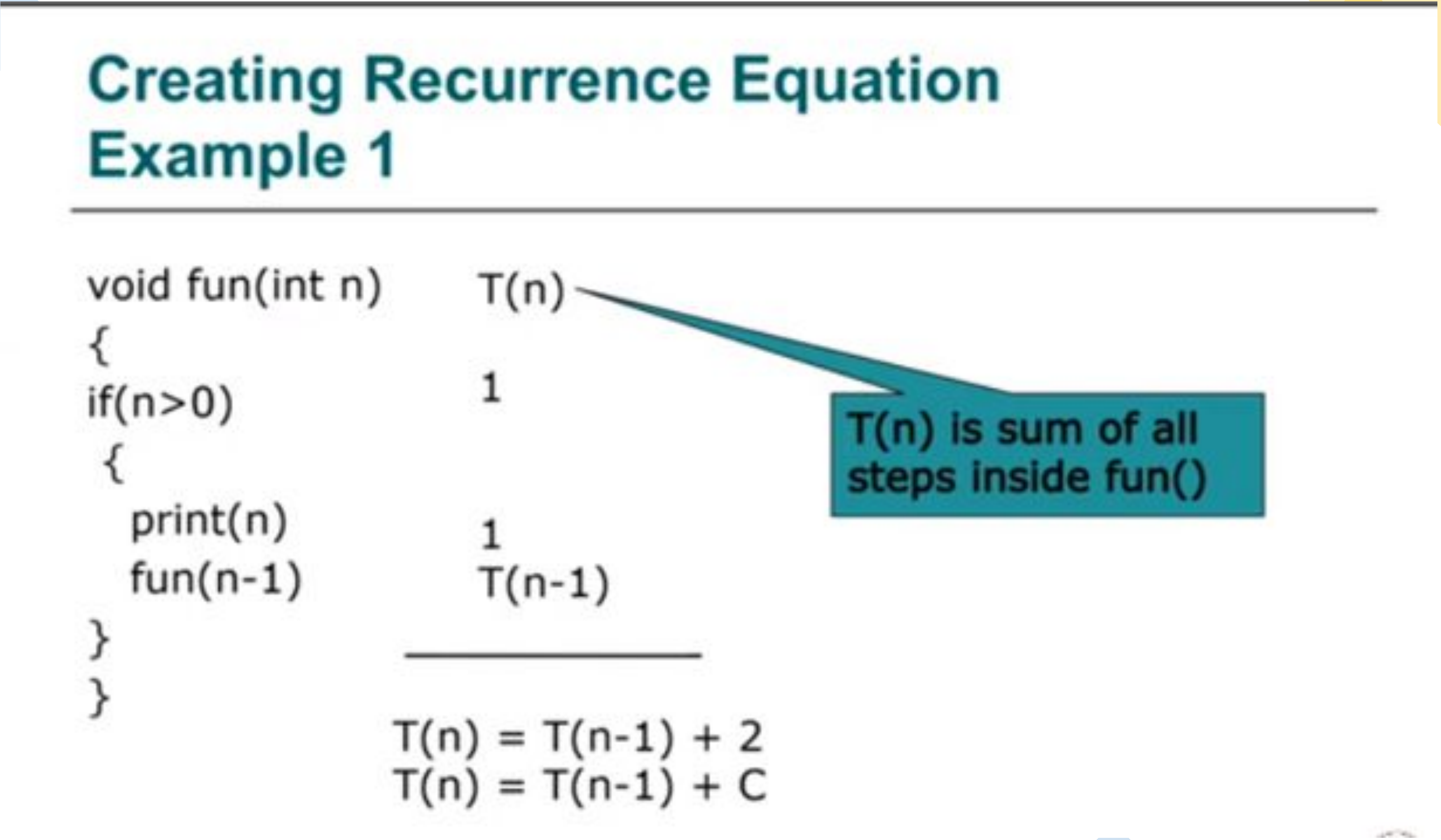
$$N = 2^K$$

$$\text{Log}_2 N = K\,\text{Log}_2 2$$

$$\boxed{K = \text{Log}_2 N}$$

Search element is not present in the array.


Successful Search
Best Case
O(1)
Average Case
O(logn)
Worst Case
O(logn)
Unsuccessful Search
Best Case
O(logn)
Average Case
O(logn)
Worst Case
O(logn)



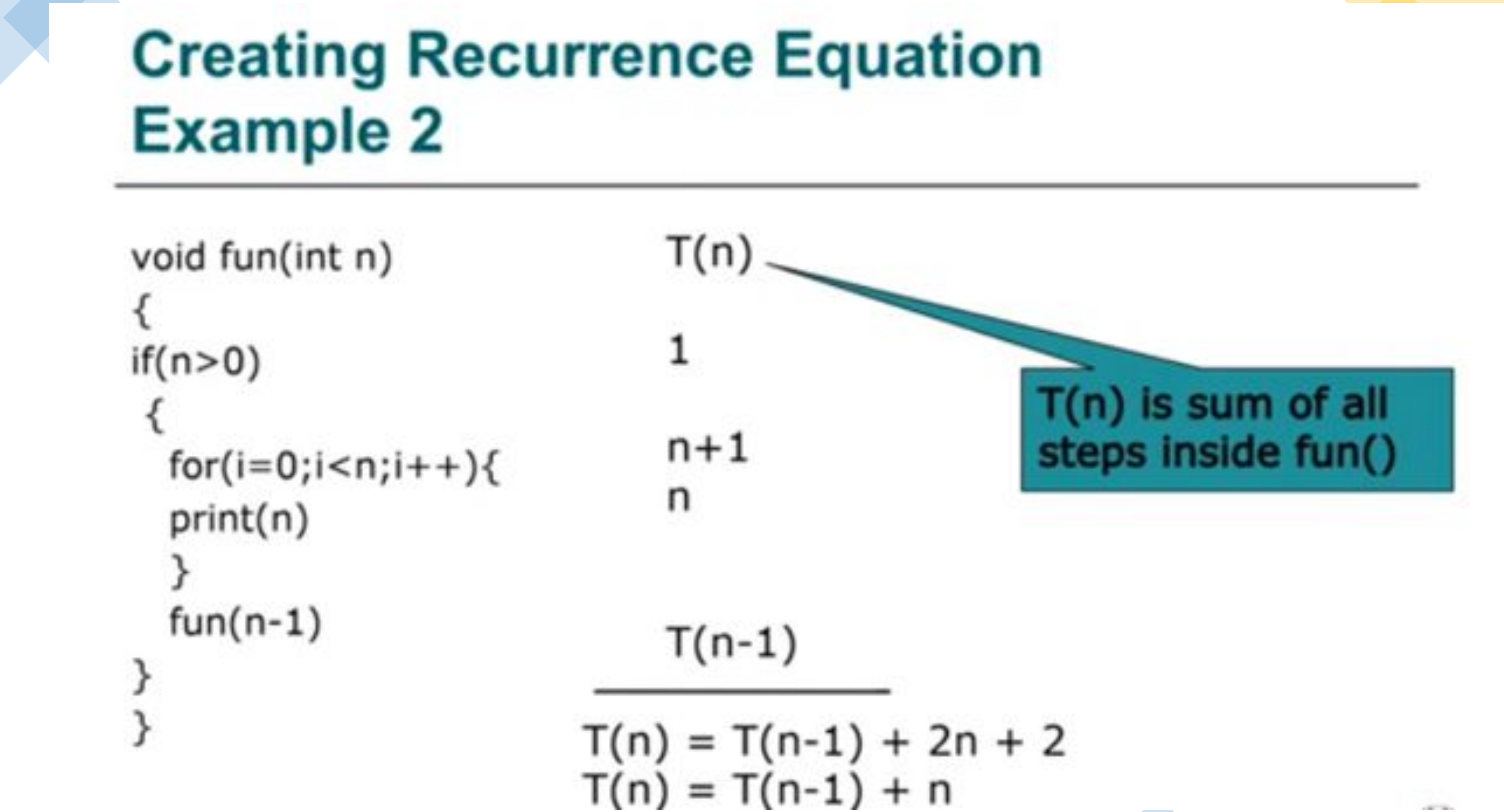
Creating Recurrence Equation Example 1

```
void fun(int n)        T(n)
{
if(n>0)                1
 {
   print(n)            1
   fun(n-1)            T(n-1)
}
}
```

T(n) is sum of all steps inside fun()

$$T(n) = T(n-1) + 2$$

$$T(n) = T(n-1) + C$$

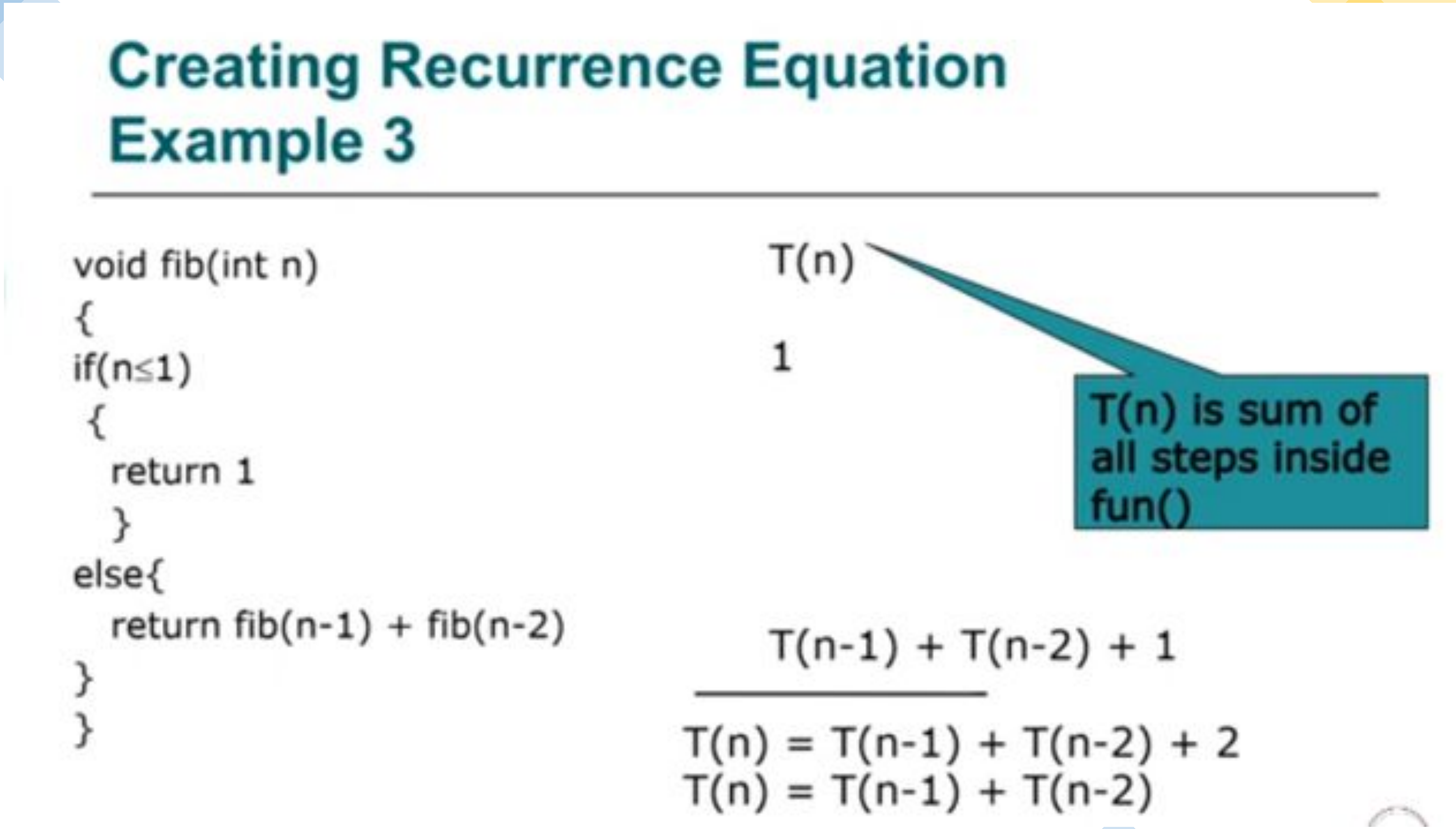


Creating Recurrence Equation Example 2

```
void fun(int n)              T(n)
{
if(n>0)                      1
 {
   for(i=0;i<n;i++){         n+1
   print(n)                  n
   }
   fun(n-1)                  T(n-1)
}
}
```

T(n) is sum of all steps inside fun()

$$T(n) = T(n-1) + 2n + 2$$
$$T(n) = T(n-1) + n$$



Creating Recurrence Equation Example 3

```
void fib(int n)
{
if(n≤1)
 {
   return 1
   }
else{
   return fib(n-1) + fib(n-2)
}
}
```

$T(n)$

1

T(n) is sum of all steps inside fun()

$T(n-1) + T(n-2) + 1$

$$T(n) = T(n-1) + T(n-2) + 2$$
$$T(n) = T(n-1) + T(n-2)$$

There are four methods for solving Recurrence:

1. Substitution Method
2. Iteration Method
3. Recursion Tree Method
4. Master Method

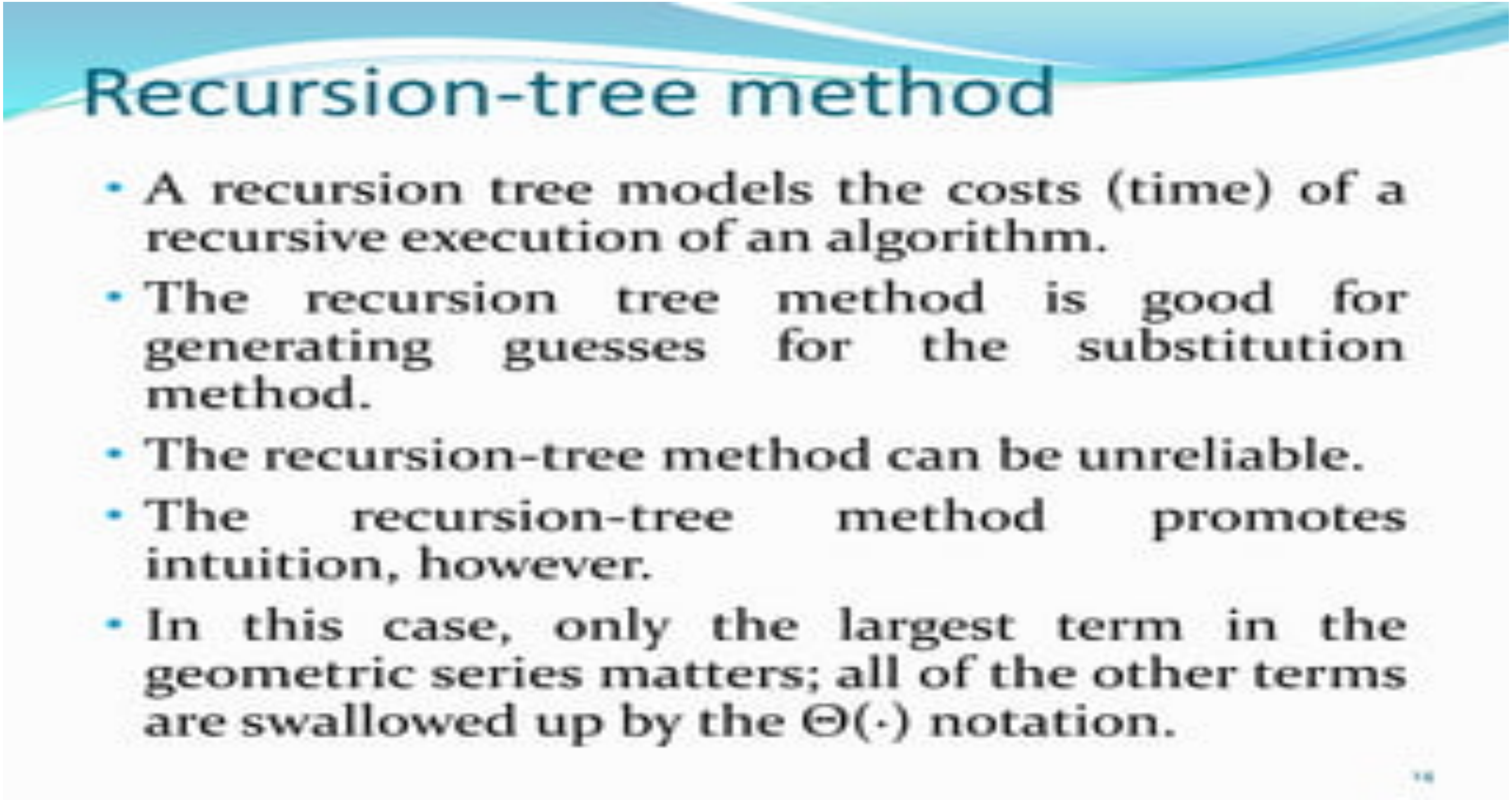
1. Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For Example1 Solve the equation by Substitution Method.

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



Recursion-tree method

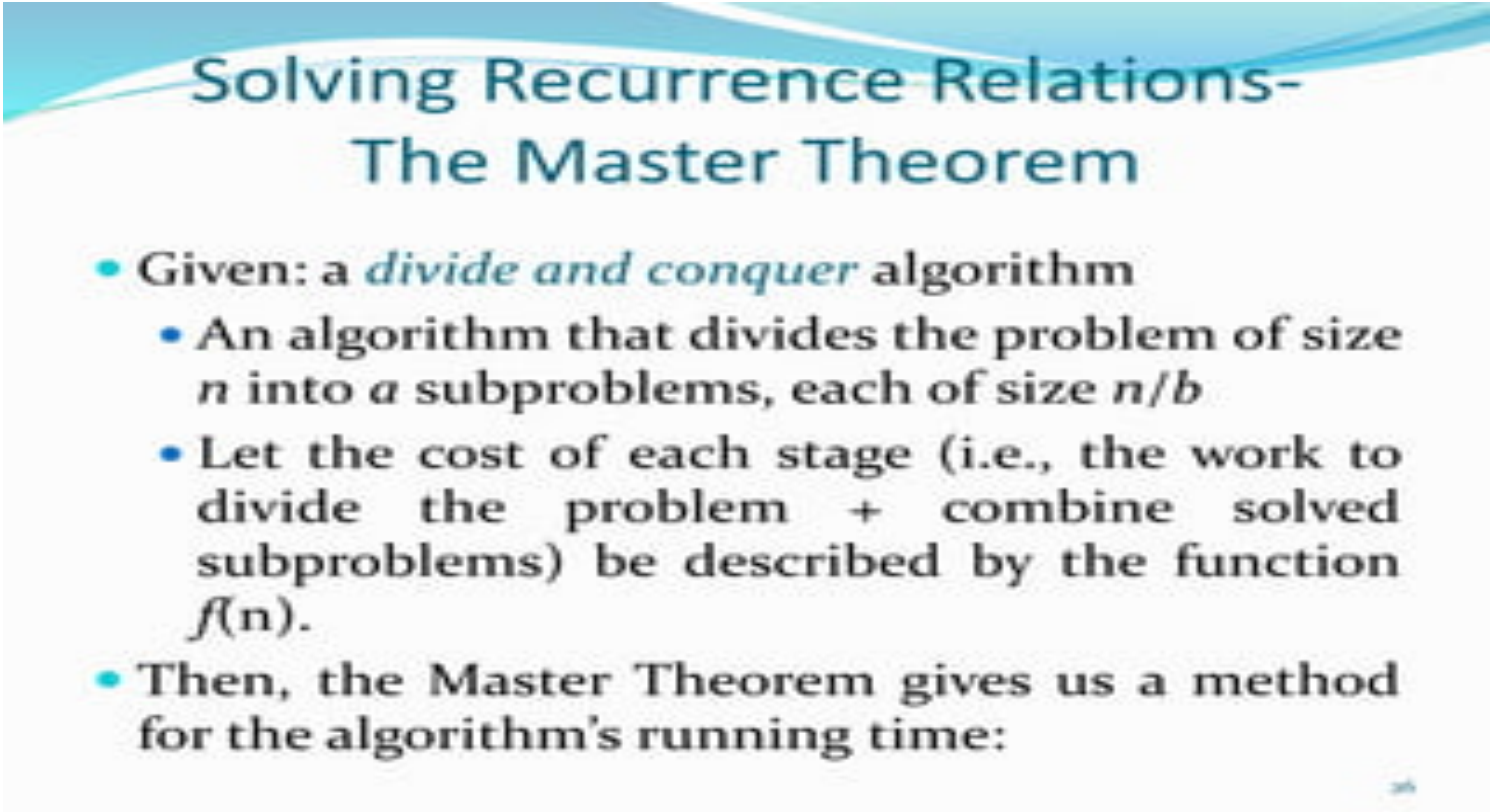
- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable.
- The recursion-tree method promotes intuition, however.
- In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the $\Theta(\cdot)$ notation.

Recursion Tree Method

Recursion is a fundamental concept in computer science and mathematics that allows functions to call themselves, enabling the solution of complex problems through iterative steps. One visual representation commonly used to understand and analyze the execution of recursive functions is a recursion tree. In this article, we will explore the theory behind recursion trees, their structure, and their significance in understanding recursive algorithms.

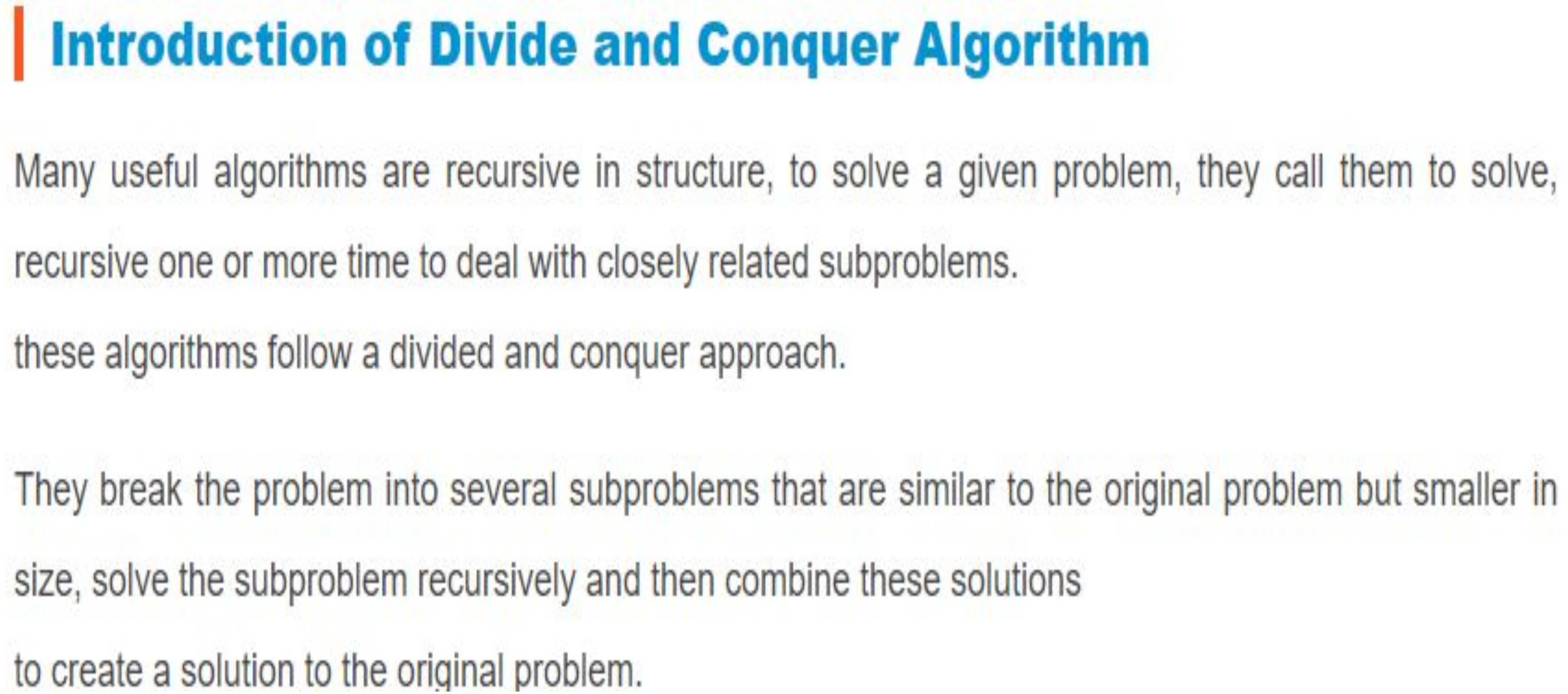
What is a Recursion Tree?

A recursion tree is a graphical representation that illustrates the execution flow of a recursive function. It provides a visual breakdown of recursive calls, showcasing the progression of the algorithm as it branches out and eventually reaches a base case. The tree structure helps in analyzing the time complexity and understanding the recursive process involved.



Solving Recurrence Relations- The Master Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size *n* into *a* subproblems, each of size *n*/*b*
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function *f*(n).
- Then, the Master Theorem gives us a method for the algorithm's running time:



Introduction of Divide and Conquer Algorithm

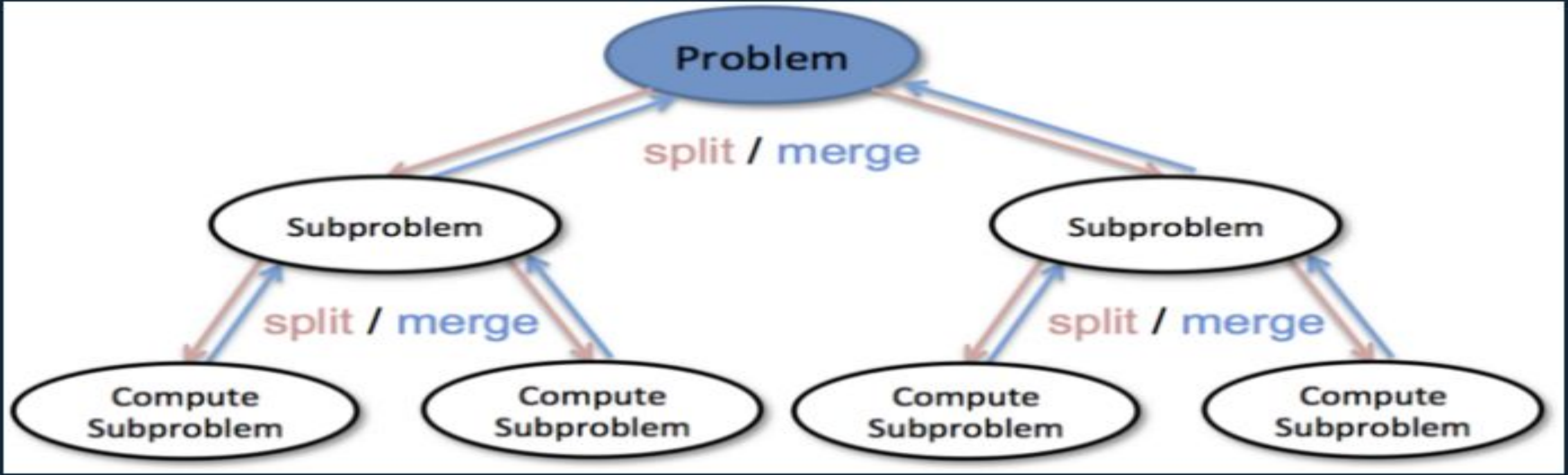
Many useful algorithms are recursive in structure, to solve a given problem, they call them to solve, recursive one or more time to deal with closely related subproblems.

these algorithms follow a divided and conquer approach.

They break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblem recursively and then combine these solutions

to create a solution to the original problem.

Divide and Conquer Algorithms



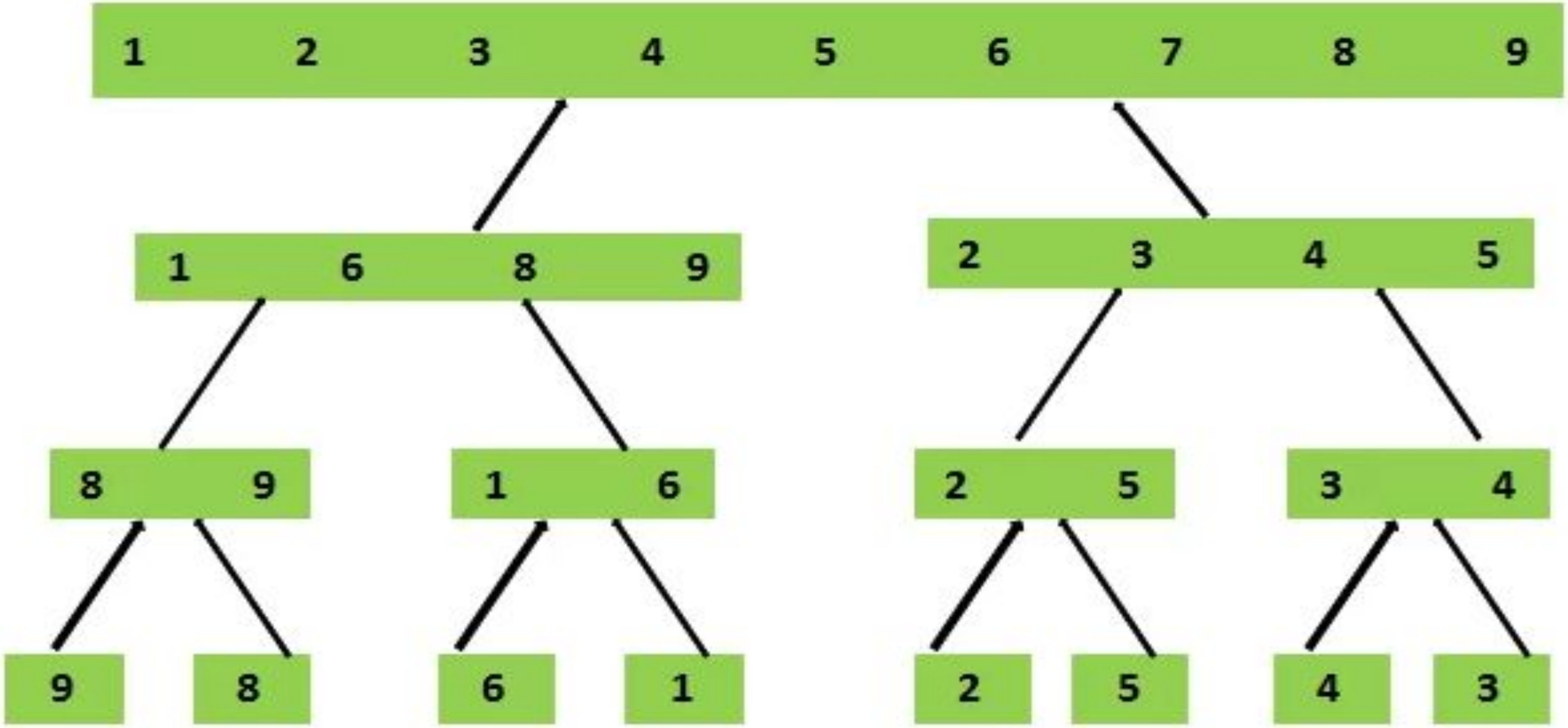
Divide:- The problem into a number of subproblems.

Conquer:- The subproblem by solving them recursively.

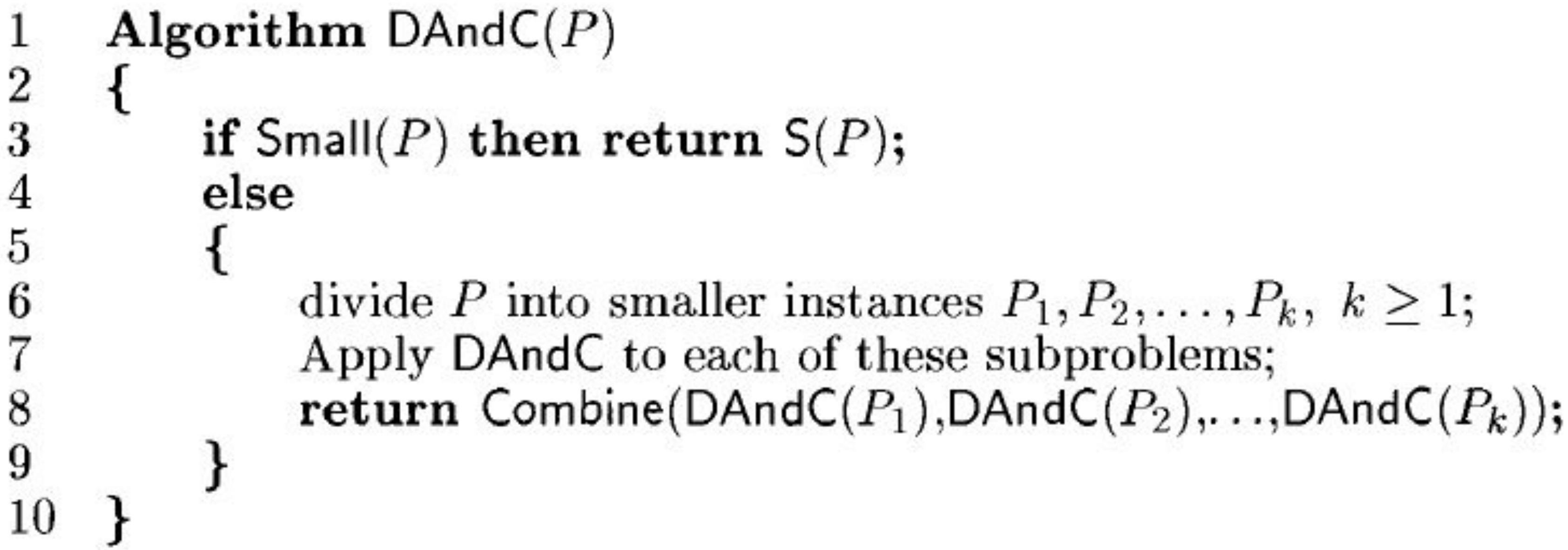
Combine:- it define as the solutions to the subproblem into the solution for the original problem

For Example

A={9,8,6,1,2,5,4,3}



Divide and Conquer



```
Algorithm DAndC($P$)
{
    if Small($P$) then return S($P$);
    else
    {
        divide $P$ into smaller instances $P_1, P_2, \ldots, P_k,\ k \geq 1$;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC($P_1$),DAndC($P_2$),. . .,DAndC($P_k$));
    }
}
```

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

Following, these are few standard algorithms that are Divide and Conquer algorithms.

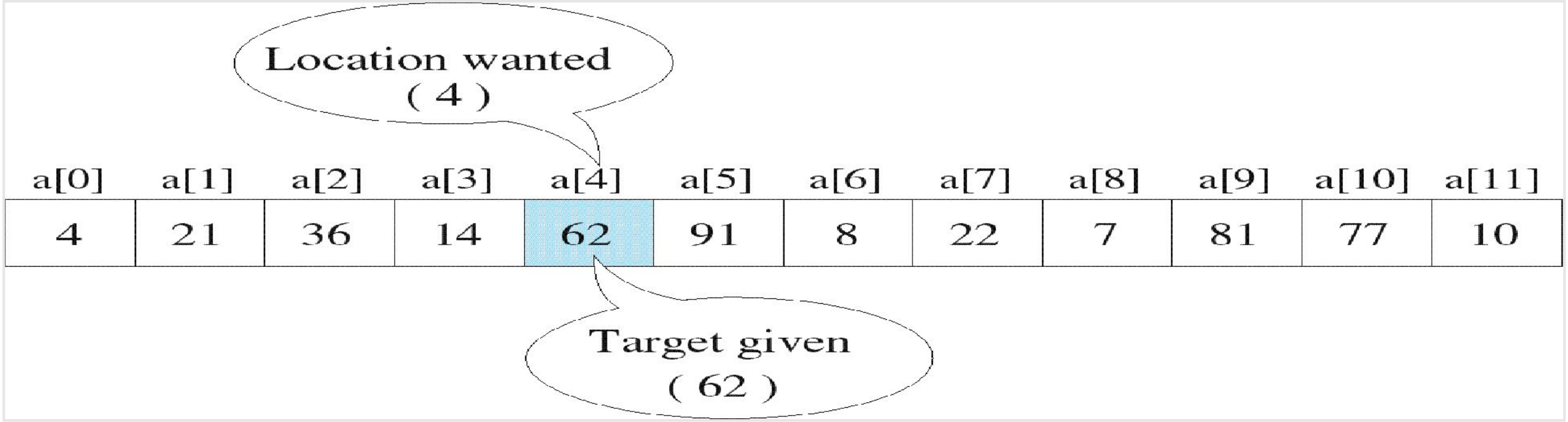
- BINARY SEARCH
- QUICK SORT
- MERGE SORT

SEARCHING:

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

Example: Searching an array finds the index of first element in an array containing that value

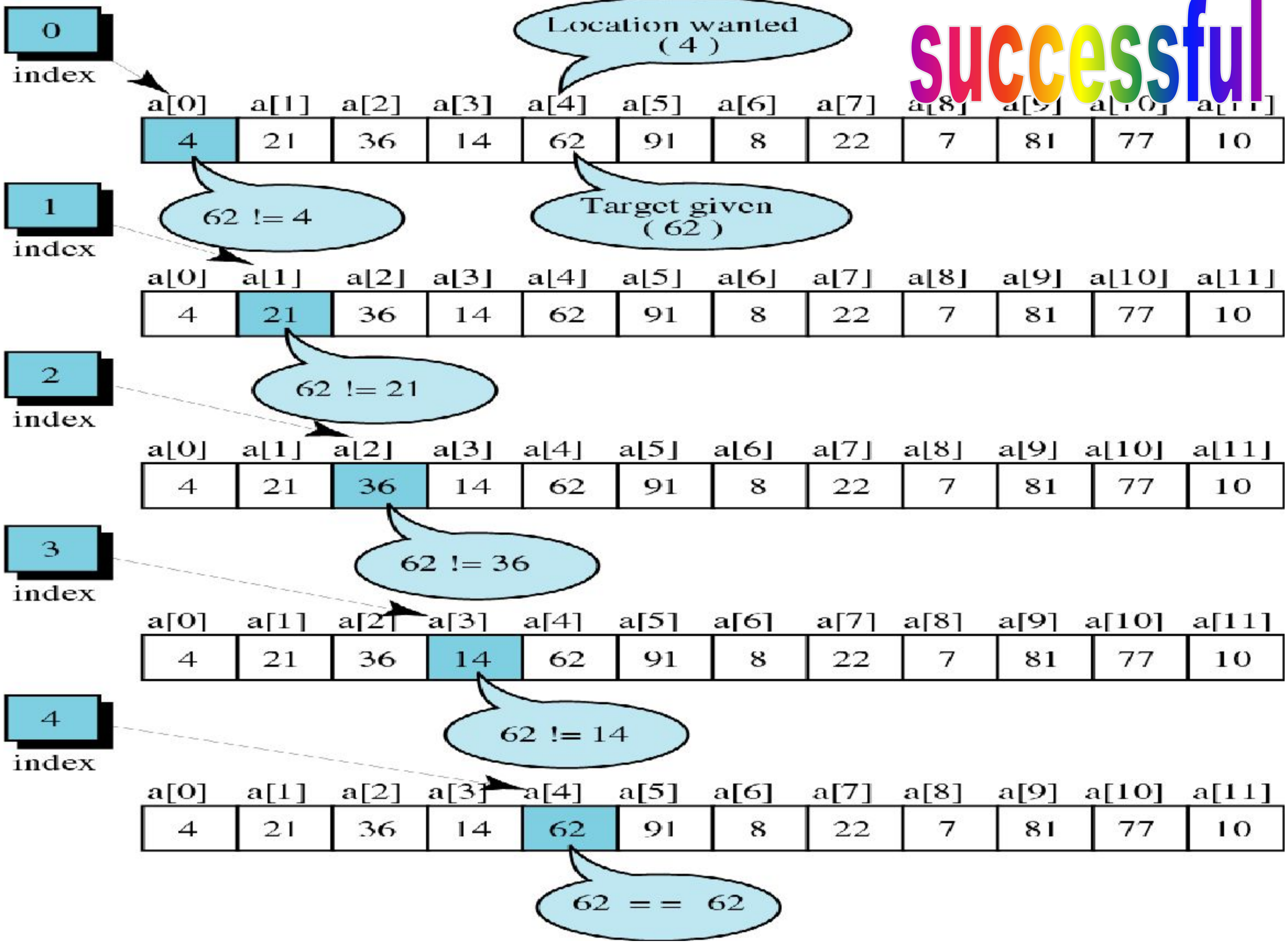


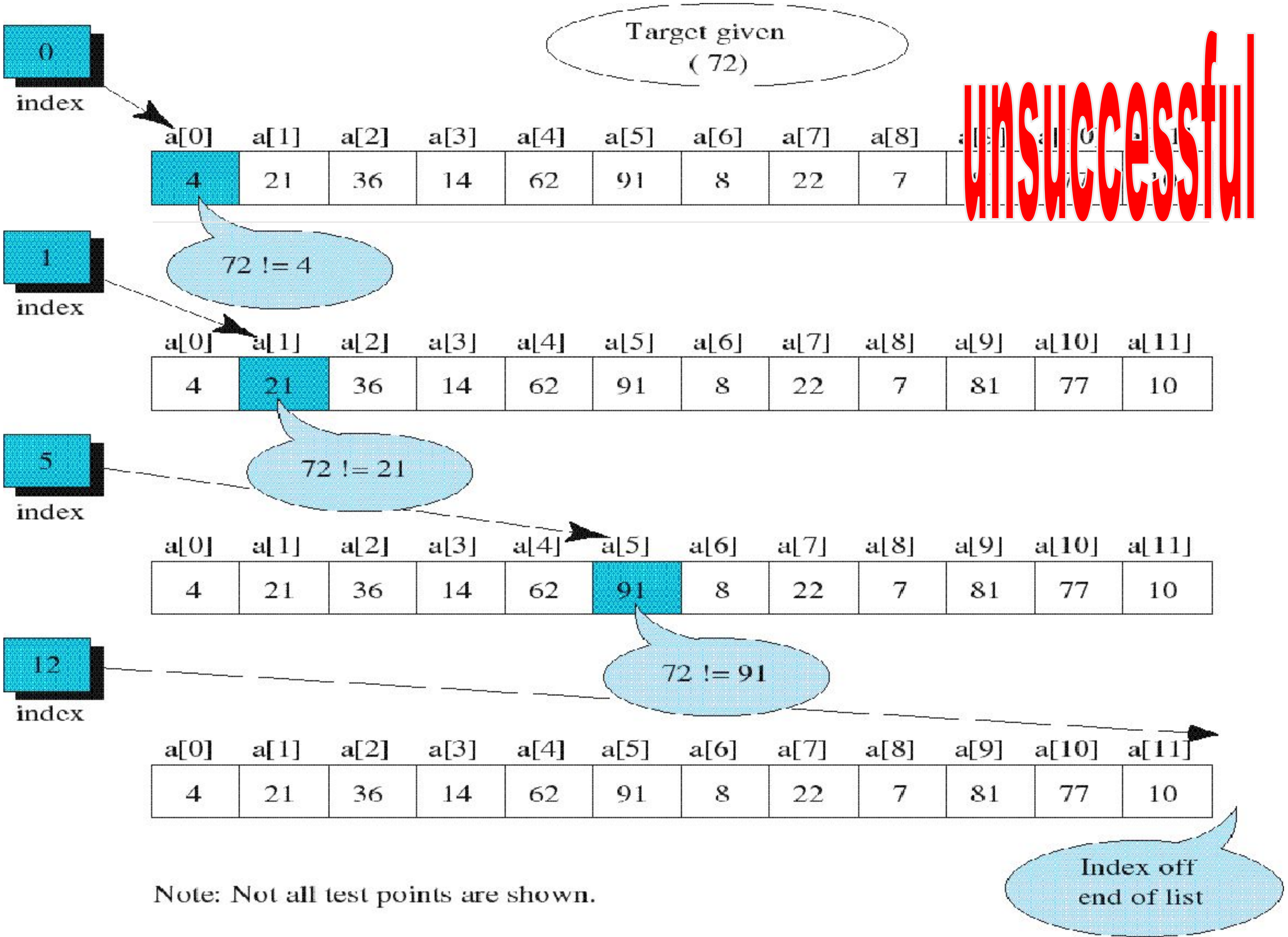
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	21	36	14	62	91	8	22	7	81	77	10

Linear Search:

Linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

successful


0
index
Location wanted
(4)
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 4
Target given
(62)
1
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 21
2
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 36
3
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 14
4
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 == 62



Target given
(72)

unsuccesful

0
index

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
4	21	36	14	62	91	8	22	7

72 != 4

1
index

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	21	36	14	62	91	8	22	7	81	77	10

72 != 21

5
index

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	21	36	14	62	91	8	22	7	81	77	10

72 != 91

12
index

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	21	36	14	62	91	8	22	7	81	77	10

Index off
end of list

Note: Not all test points are shown.

Non Recursive Algorithm:

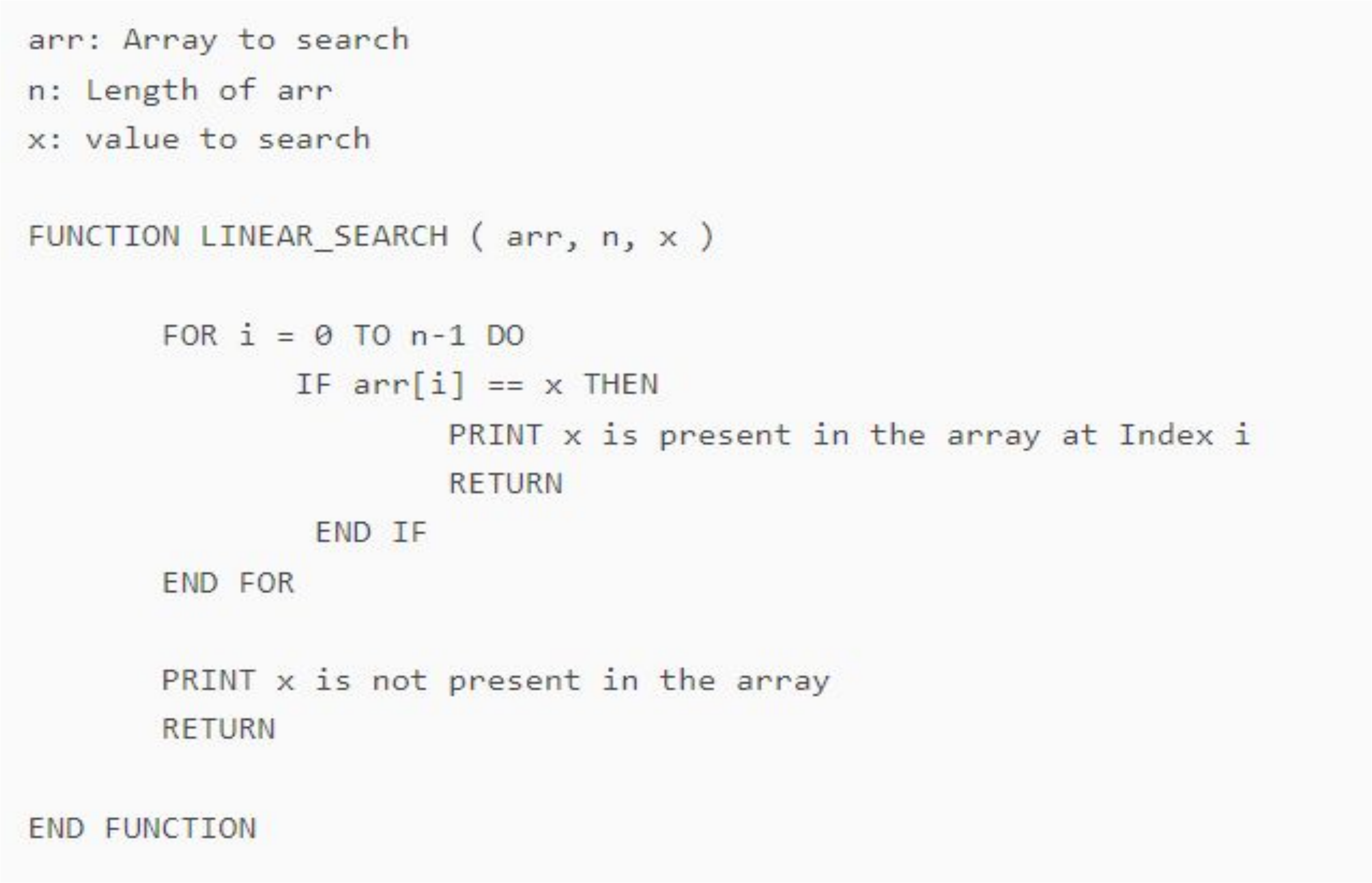
Algorithm Linearsearch(a<array>,n,ele)

Input: a is an array with n elements, ele is the element to be search.

Output: position of required element in array, if it is available.

```
1. i = 0
2. found = 0
3. while(i < n and found = = 0)
      a) if( a[i] = = ele)
            i) print(required element was found at position i)
            ii) found=1
b) else
i) i = i +1
c) end if
4. end if
5. if(found!=1)
      a) print(required element is not available)
6. end if
```

End Linearsearch



```
arr: Array to search
n: Length of arr
x: value to search

FUNCTION LINEAR_SEARCH ( arr, n, x )

       FOR i = 0 TO n-1 DO
              IF arr[i] == x THEN
                      PRINT x is present in the array at Index i
                      RETURN
               END IF
       END FOR

       PRINT x is not present in the array
       RETURN

END FUNCTION
```

Recursive Algorithm:

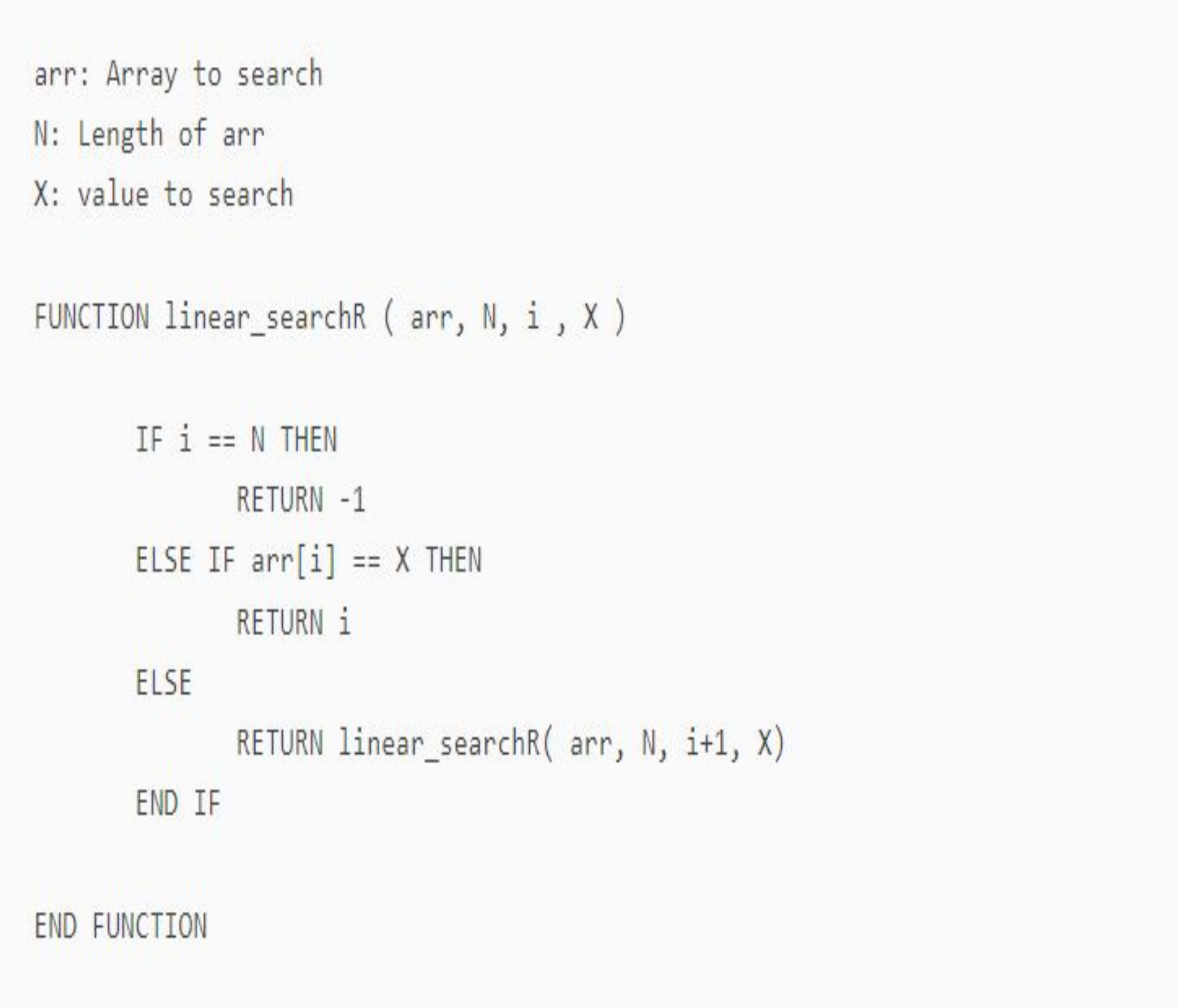
Algorithm Linearsearch_Recursion(a<array>,n,ele,i)

Input: a is an array with n elements,ele is the element to be search, i is starting index.

Output: position of required element in array, if it is available.

```
1. found = 0
2. if(i < n and found = = 0)
      a)  if( a[i] = = ele)
            i)  print(required element was found at position i)
            ii) found=1
b) else
i) i = i +1
ii)Linearsearch_Recursion (a,n,ele,i)
c) end if
3. end if
4. if(found!=1)
      a) print(required element is not available)
5. end if
```

End Linearsearch_Recursion



```
arr: Array to search
N: Length of arr
X: value to search

FUNCTION linear_searchR ( arr, N, i , X )

       IF i == N THEN
              RETURN -1
       ELSE IF arr[i] == X THEN
              RETURN i
       ELSE
              RETURN linear_searchR( arr, N, i+1, X)
       END IF

END FUNCTION
```

Performance of linear search

When comparing search algorithms, we only look at the number of comparisons, since we don't swap any values while searching. Often, when comparing performance, we look at three cases:

- Best case: What is the fewest number of comparisons necessary to find an item?
- Worst case: What is the most number of comparisons necessary to find an item?
- Average case: On average, how many comparisons does it take to find an item in the list?

- Best case: The best case occurs when the search term is in the first slot in the array. The number of comparisons in this case is 1.

- Worst case: The worst case occurs when the search term is in the last slot in the array, or is not in the array. The number of comparisons in this case is equal to the size of the array. If our array has N items, then it takes N comparisons in the worst case.

- Average case: On average, the search term will be somewhere in the middle of the array. The number of comparisons in this case is approximately N/2.

Binary Search:

- In computer science, a binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value.
- In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned.
- Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array, and a special "not found" indication is returned.
- A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic divide and conquer search algorithm.

successful

Example: binary search

- 14 ?

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	2	3	5	7	10	14	17

first mid last

A[4]	A[5]	A[6]	A[7]
7	10	14	17

first mid last

A[6]	A[7]
14	17

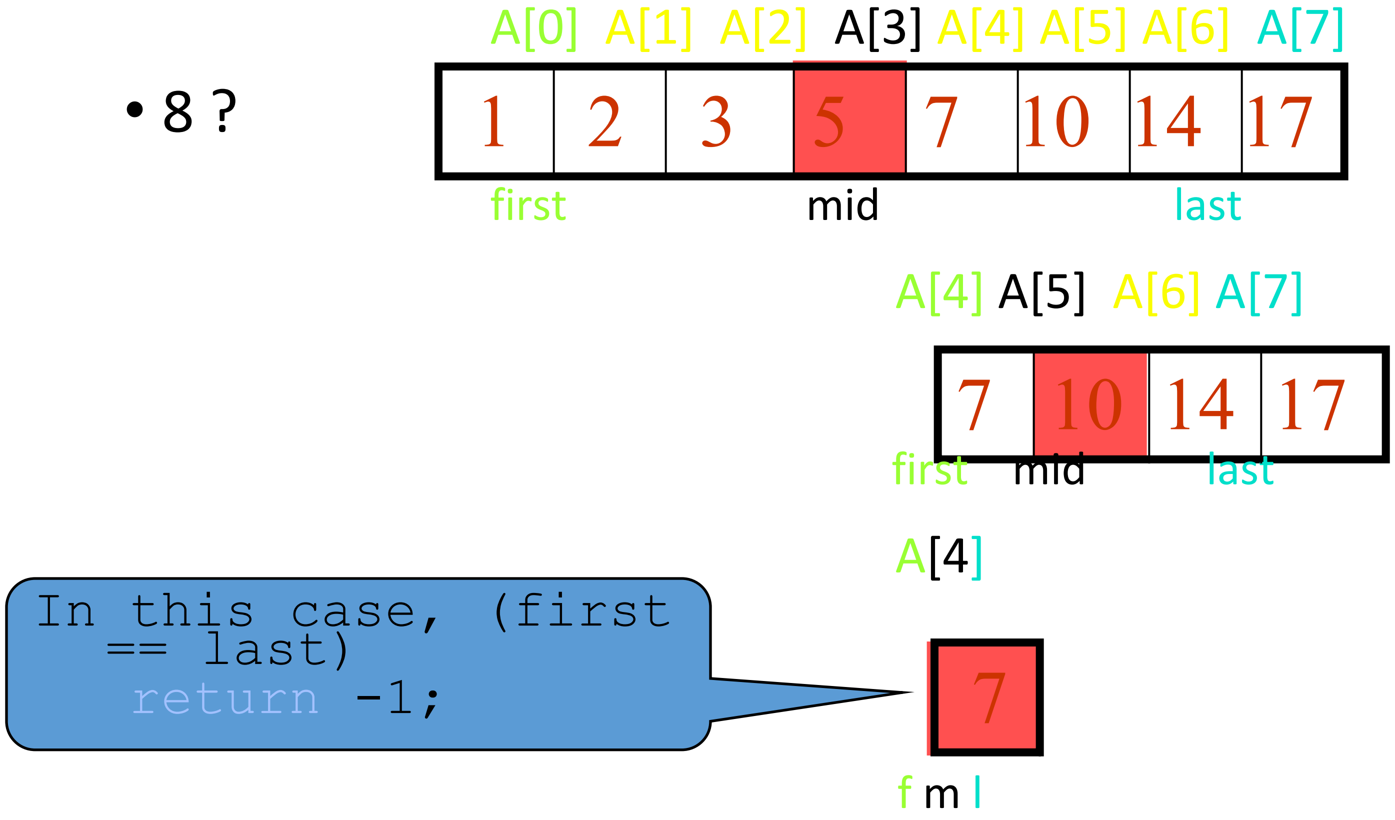
f mid last

```
In this case,
(data[middle] == value)
     return middle;
```

unsuccessful

Example: binary search

- 8 ?

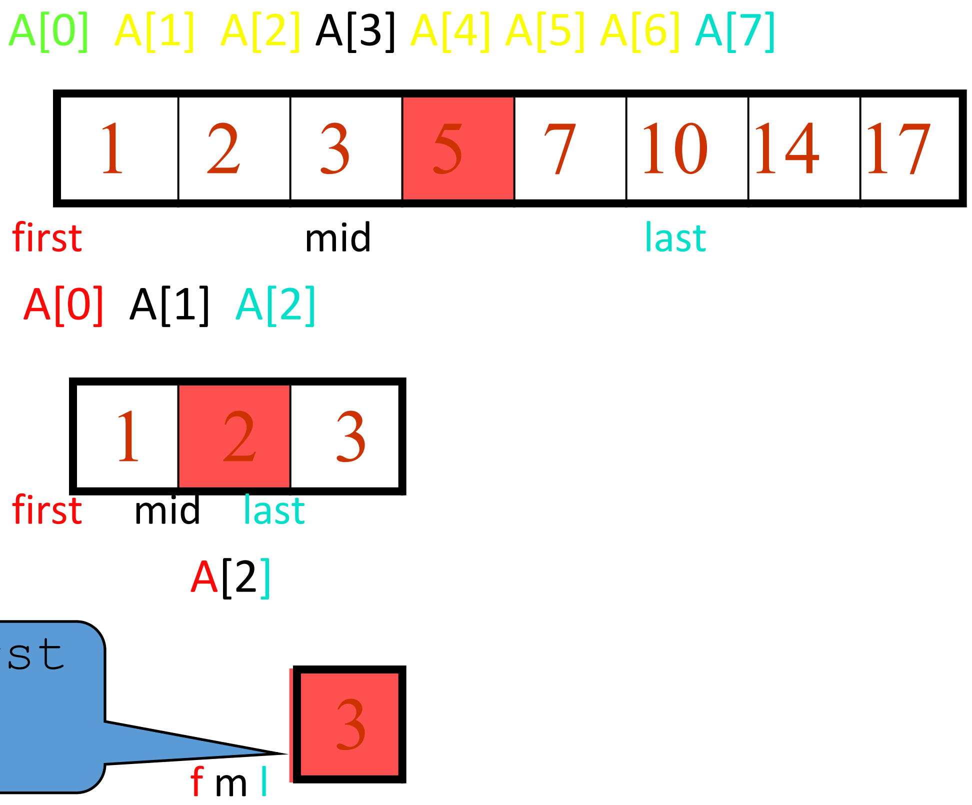


```
In this case, (first
   == last)
    return -1;
```

Example: binary search

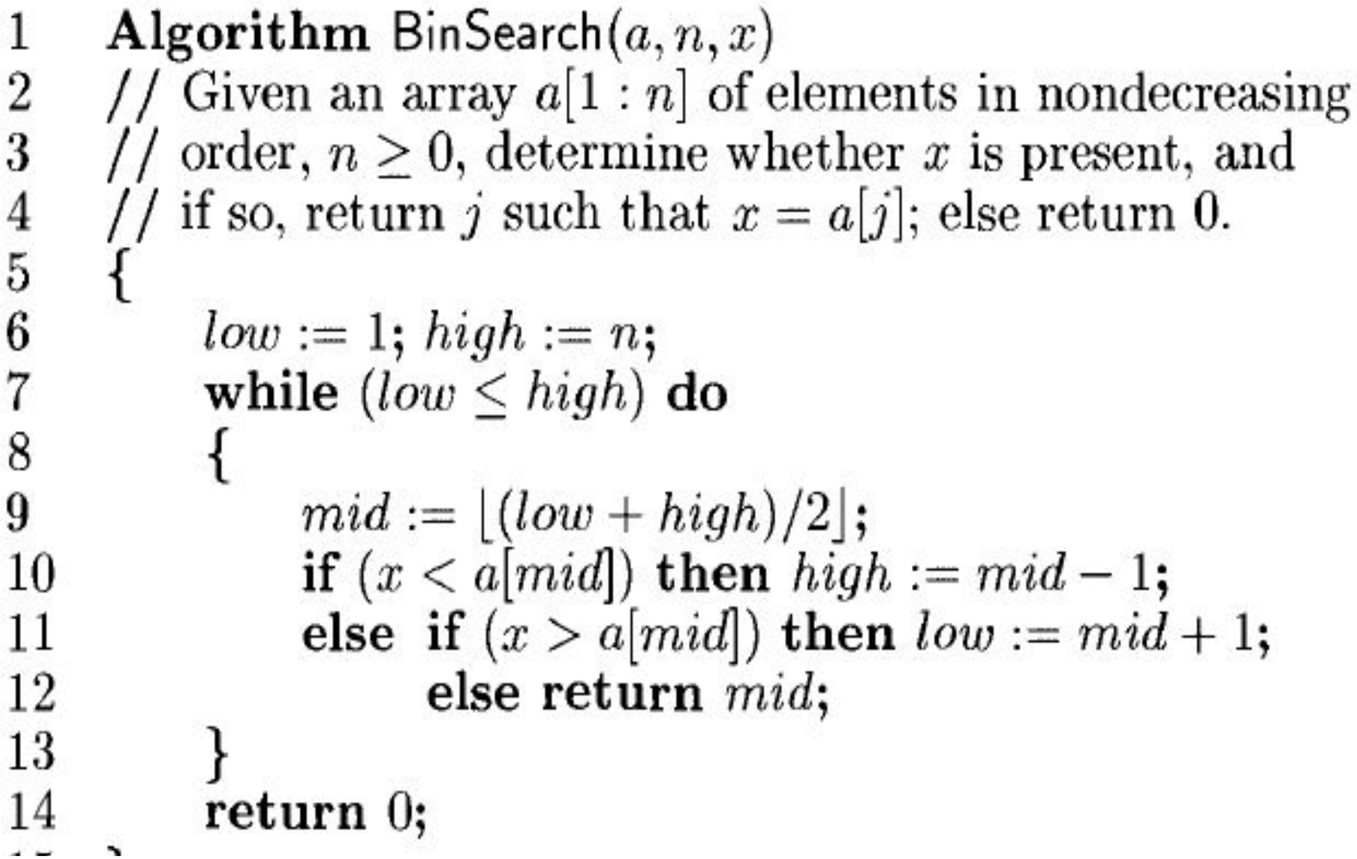
unsuccessful

- 4 ?



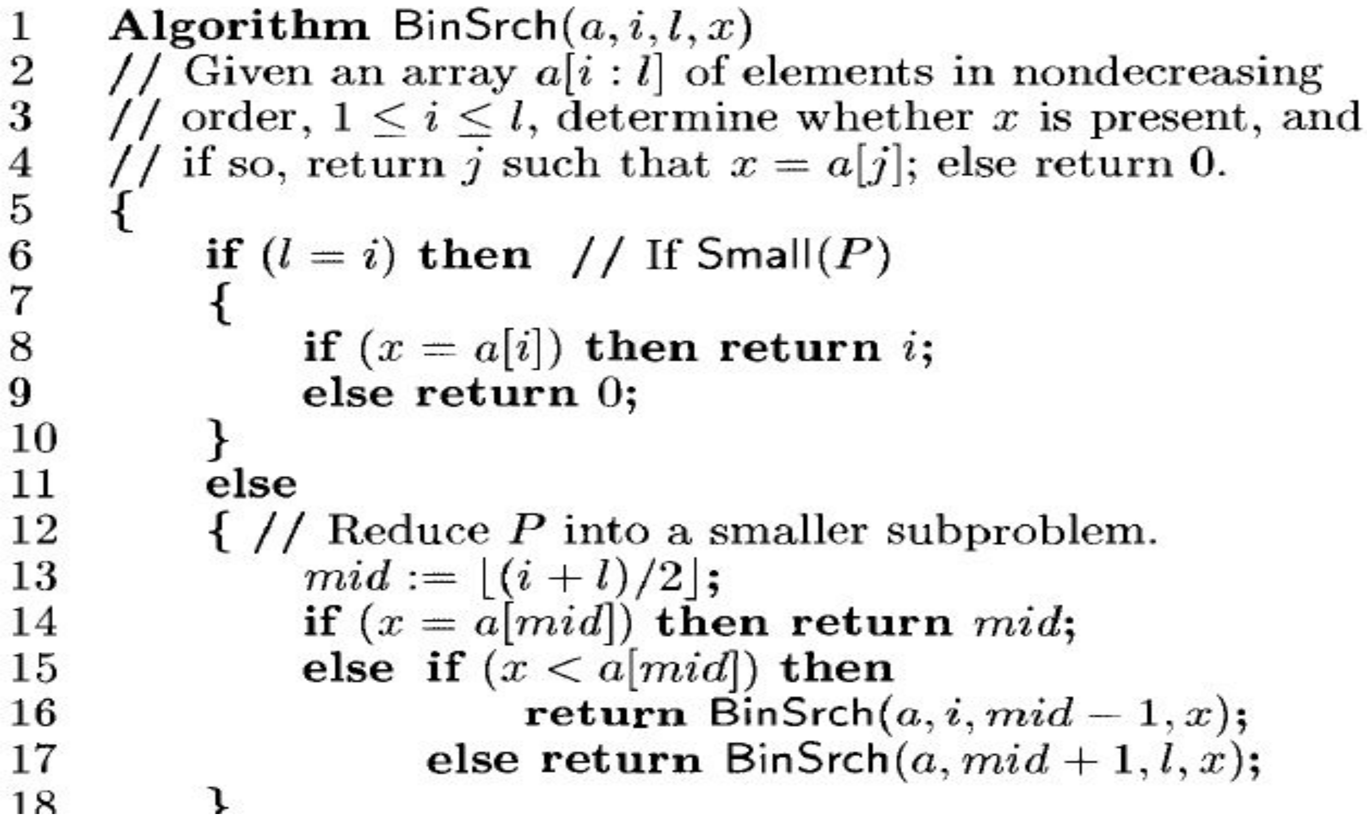
```
In this case, (first
  == last)
  return -1;
```

Non-Recursive Algorithm:



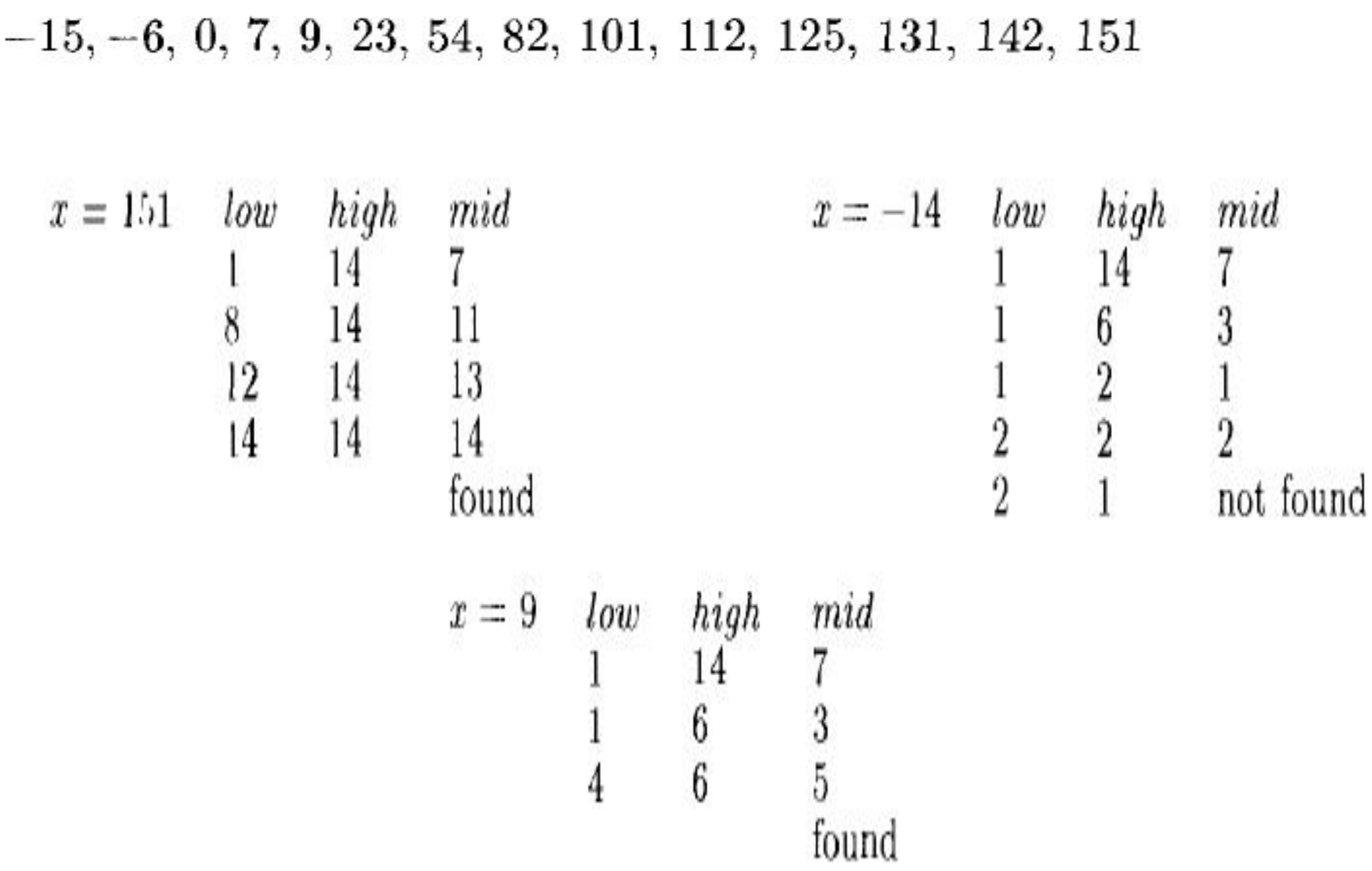
```
Algorithm BinSearch($a, n, x$)
// Given an array $a[1:n]$ of elements in nondecreasing
// order, $n \geq 0$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    $low := 1$; $high := n$;
    while ($low \leq high$) do
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        if ($x < a[mid]$) then $high := mid - 1$;
        else  if ($x > a[mid]$) then $low := mid + 1$;
             else return $mid$;
    }
    return 0;
}
```

Recursive Algorithm:



```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i:l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, 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.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else  if ($x < a[mid]$) then
                  return BinSrch($a, i, mid - 1, x$);
              else return BinSrch($a, mid + 1, l, x$);
    }
```

Example:



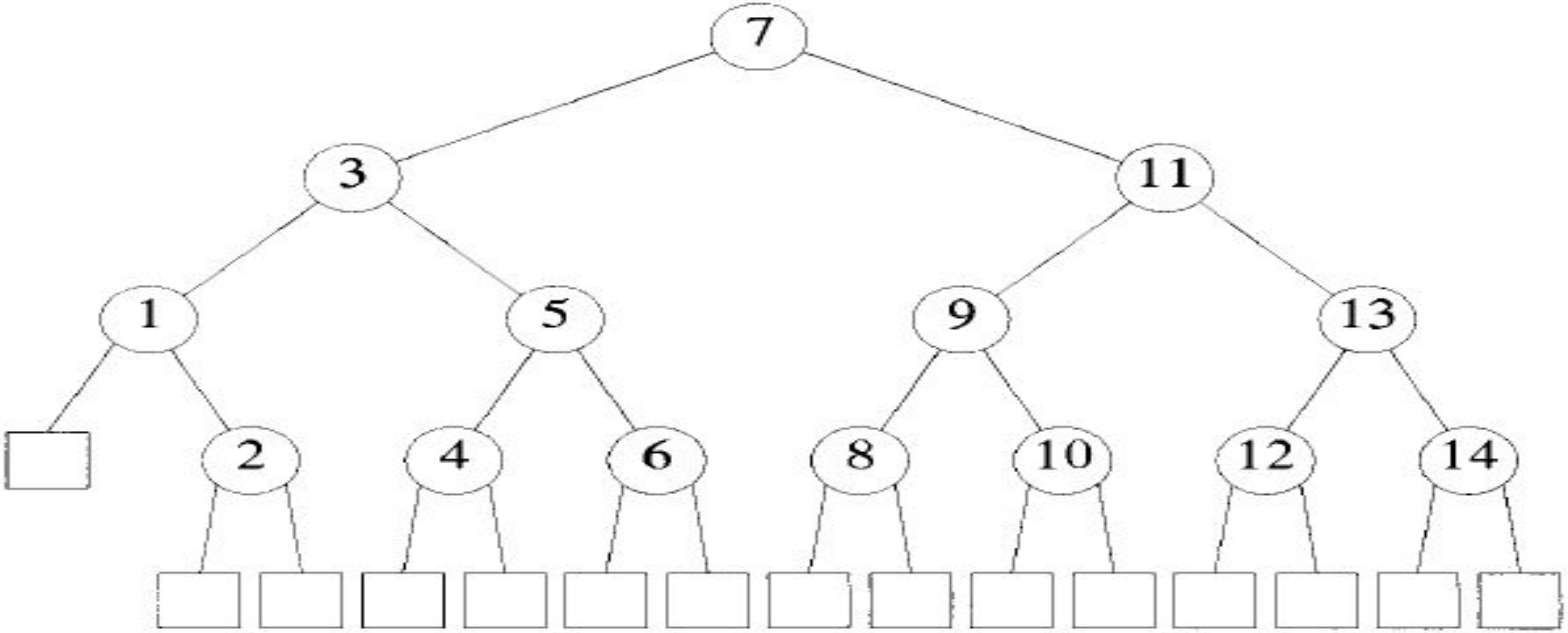
$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$

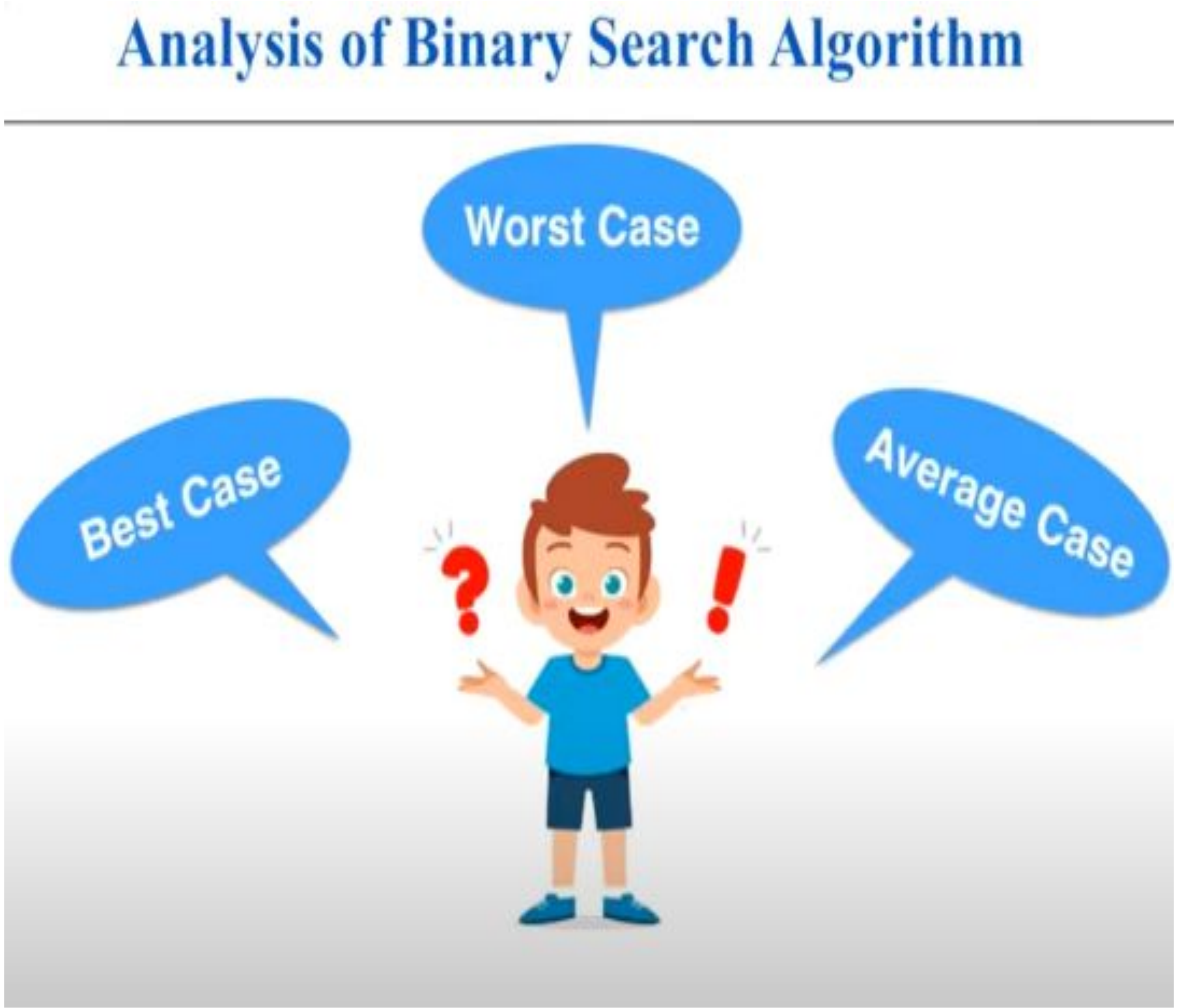
$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

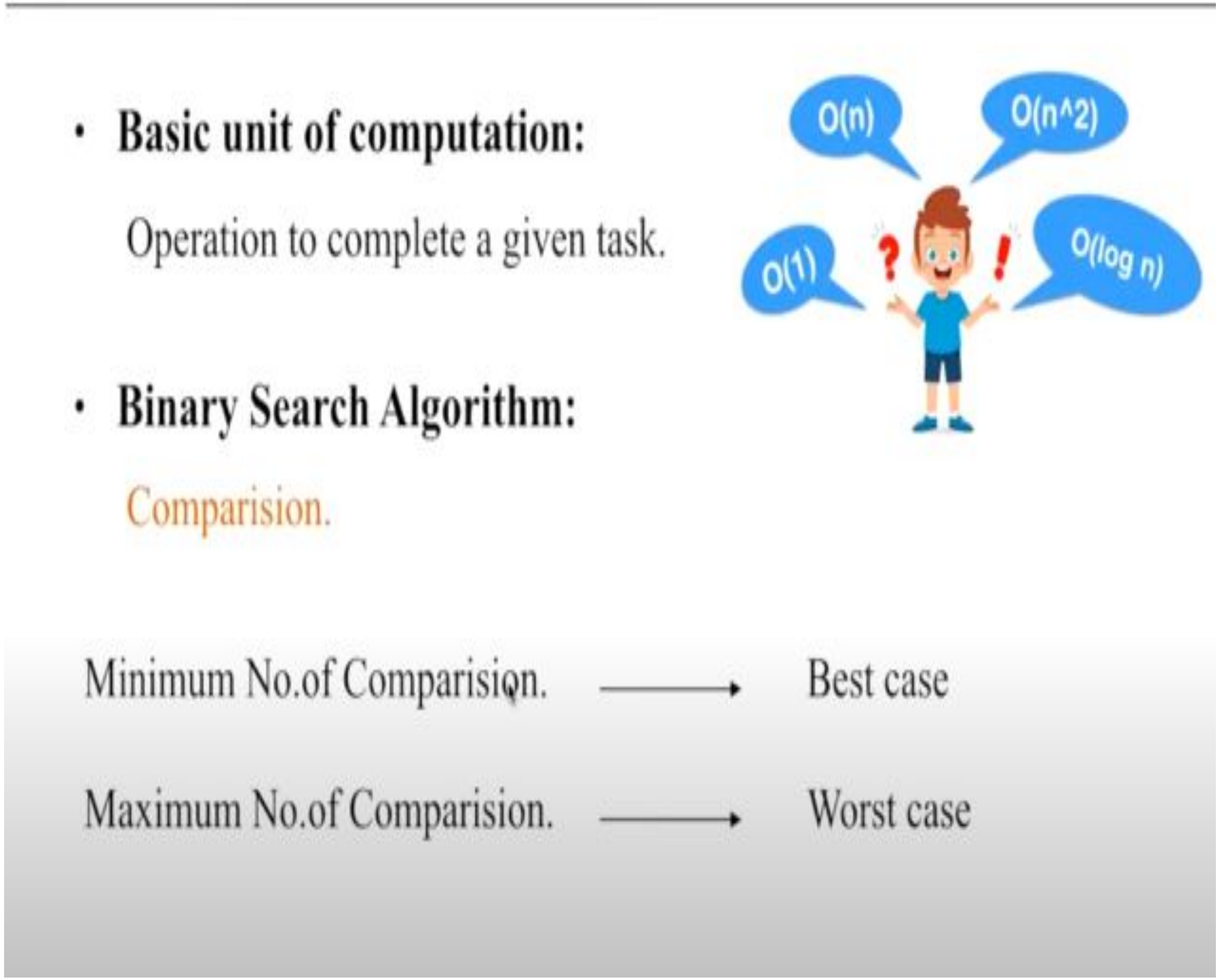
$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4





Analysis of Binary Search Algorithm



- **Basic unit of computation:**

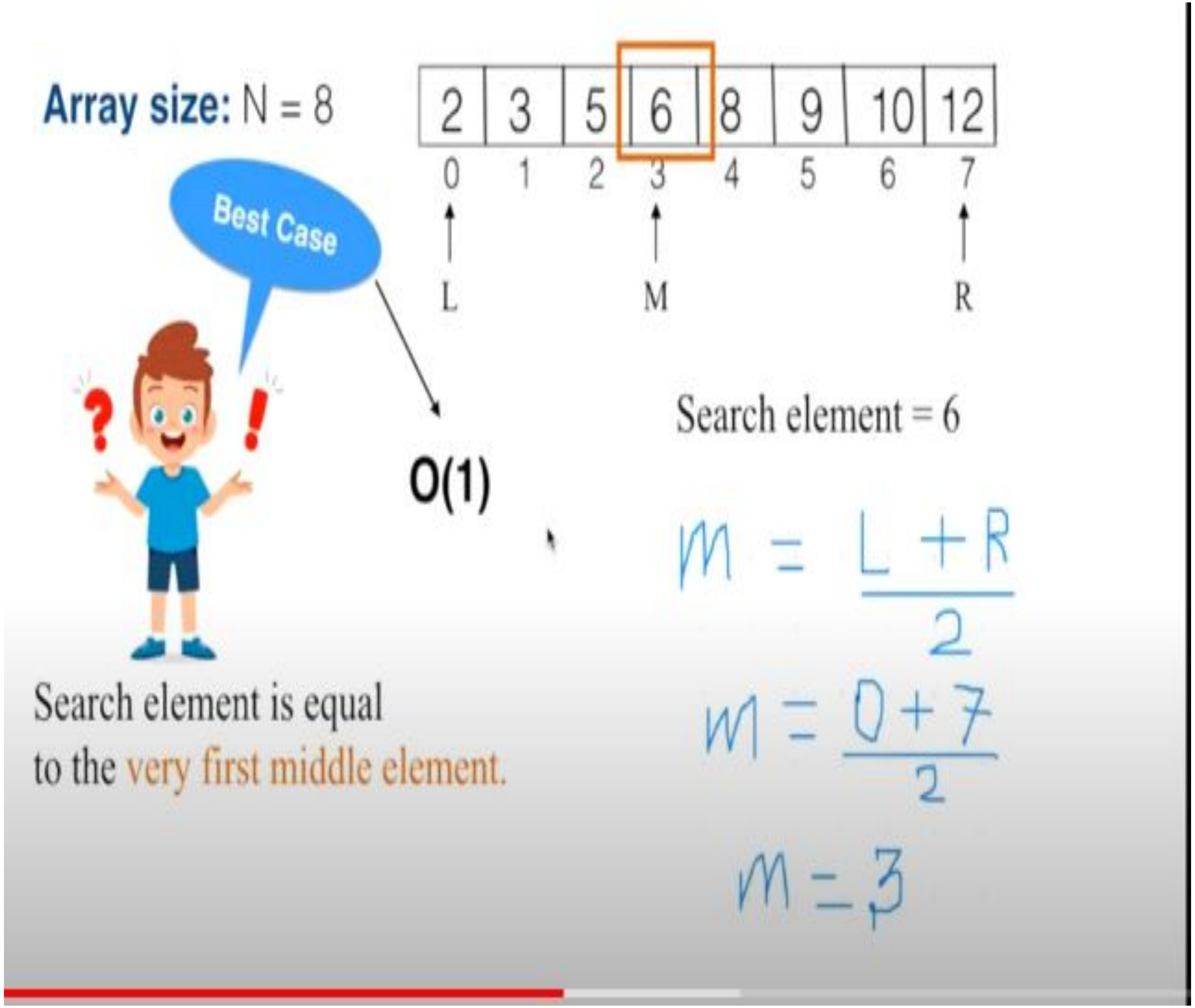
  Operation to complete a given task.

- **Binary Search Algorithm:**

  Comparision.

Minimum No.of Comparision. → Best case

Maximum No.of Comparision. → Worst case



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7
L			M				R

Best Case

O(1)

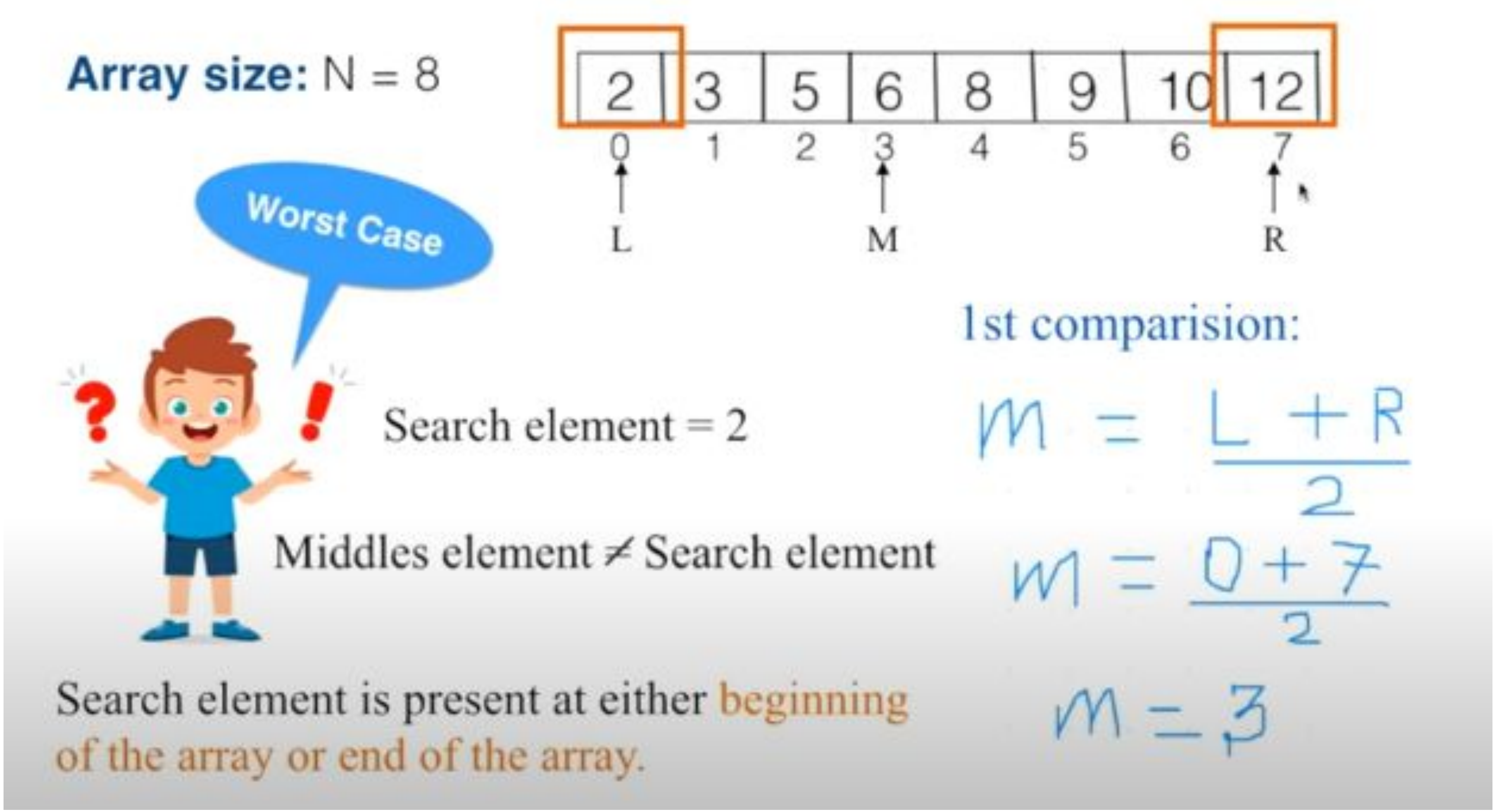
Search element = 6

$$m = \frac{L + R}{2}$$

$$m = \frac{0 + 7}{2}$$

$$m = 3$$

Search element is equal to the very first middle element.



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

Search element = 2

Middles element ≠ Search element

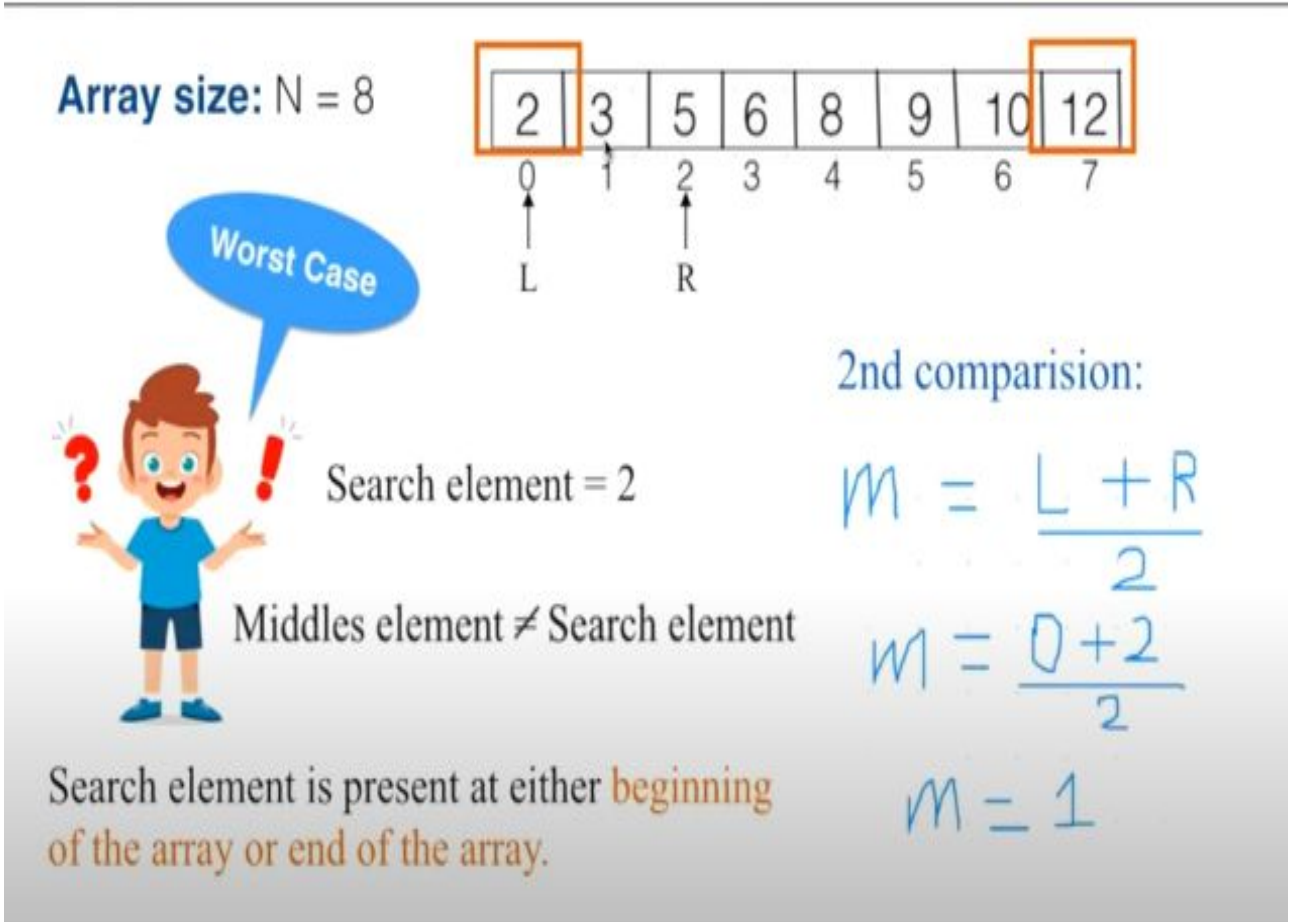
Search element is present at either beginning of the array or end of the array.

1st comparision:

$$m = \frac{L+R}{2}$$

$$m = \frac{0+7}{2}$$

$$m = 3$$



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7
L		R					

2nd comparision:

$$m = \frac{L+R}{2}$$

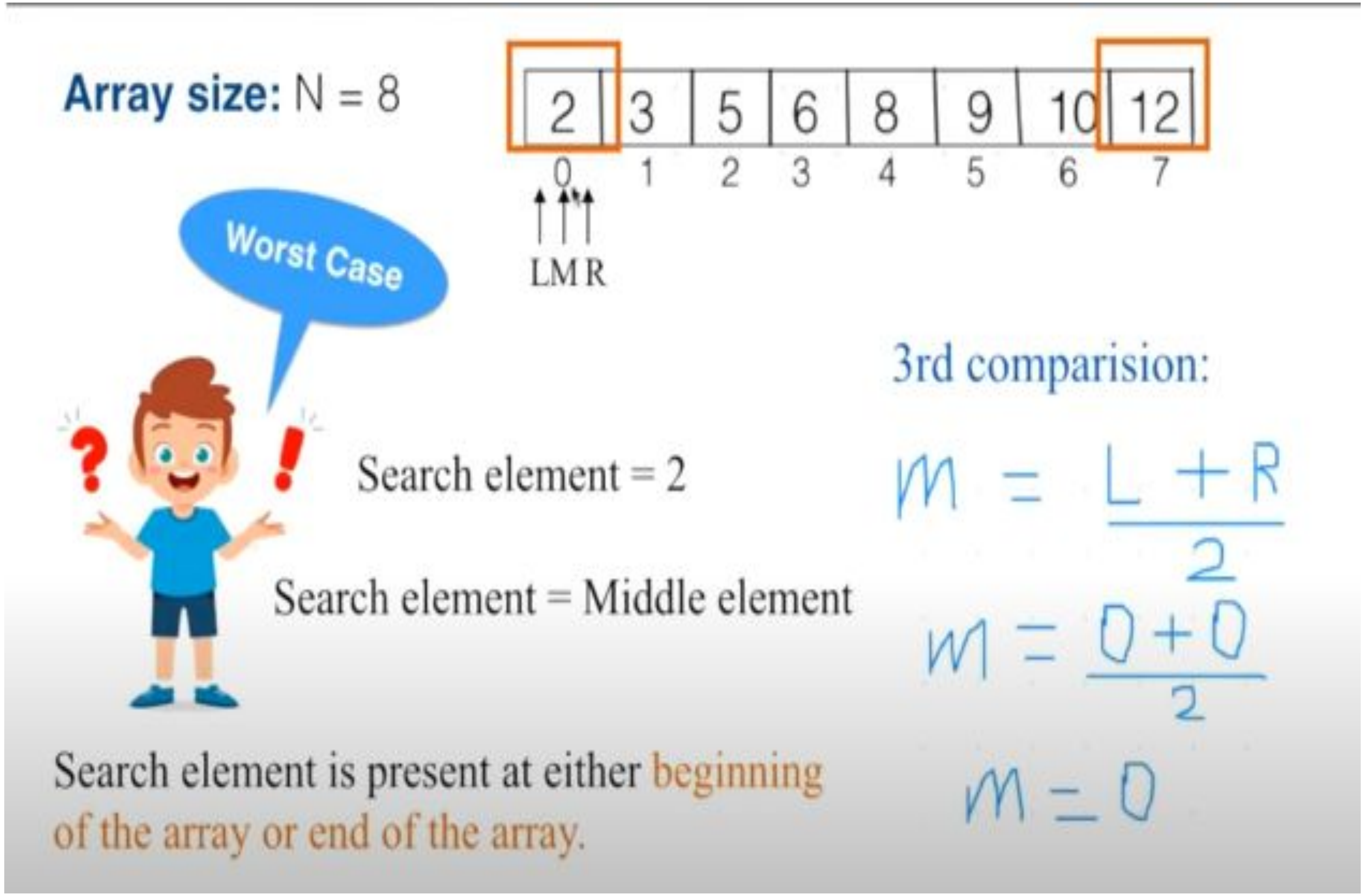
$$m = \frac{0+2}{2}$$

$$m = 1$$

Search element = 2

Middles element ≠ Search element

Search element is present at either beginning of the array or end of the array.



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

3rd comparision:

$$m = \frac{L + R}{2}$$

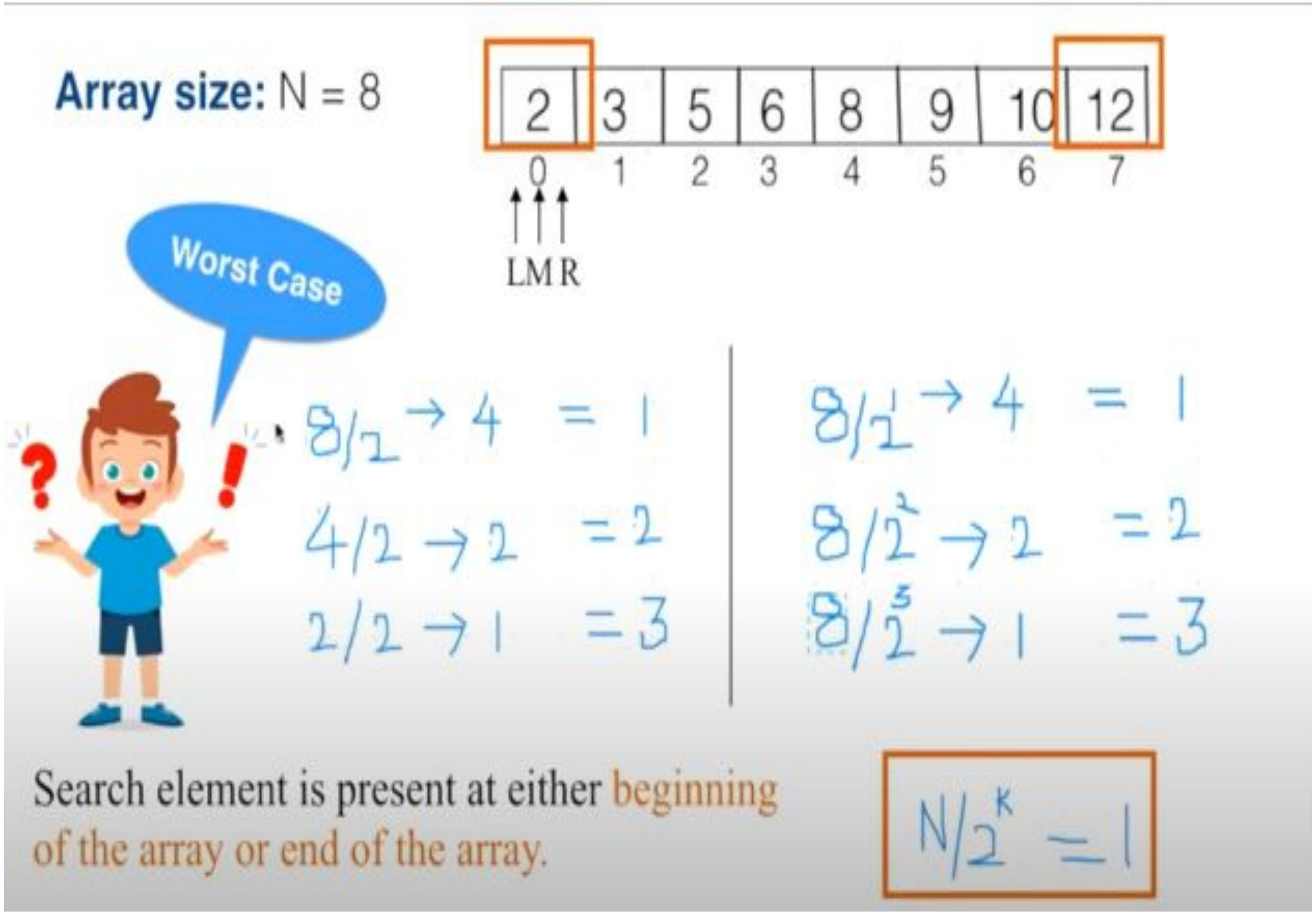
$$m = \frac{0 + 0}{2}$$

$$m = 0$$

Search element = 2

Search element = Middle element

Search element is present at either beginning of the array or end of the array.



Array size: N = 8

2	3	5	6	8	9	10	12
0	1	2	3	4	5	6	7

$8/2 \rightarrow 4 \quad = 1$

$4/2 \rightarrow 2 \quad = 2$

$2/2 \rightarrow 1 \quad = 3$

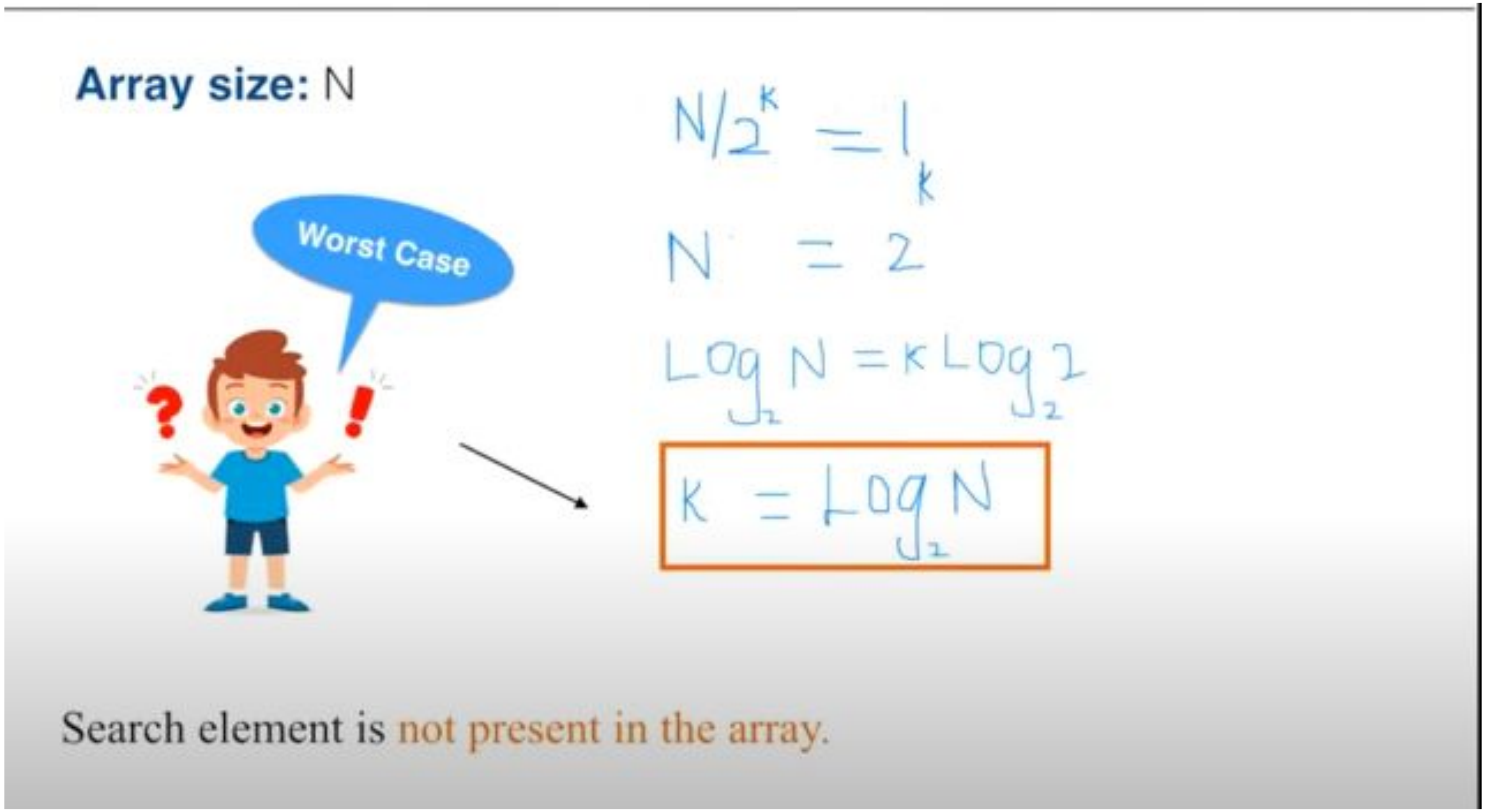
$8/2^1 \rightarrow 4 \quad = 1$

$8/2^2 \rightarrow 2 \quad = 2$

$8/2^3 \rightarrow 1 \quad = 3$

Search element is present at either beginning of the array or end of the array.

$\boxed{N/2^K = 1}$



Array size: N

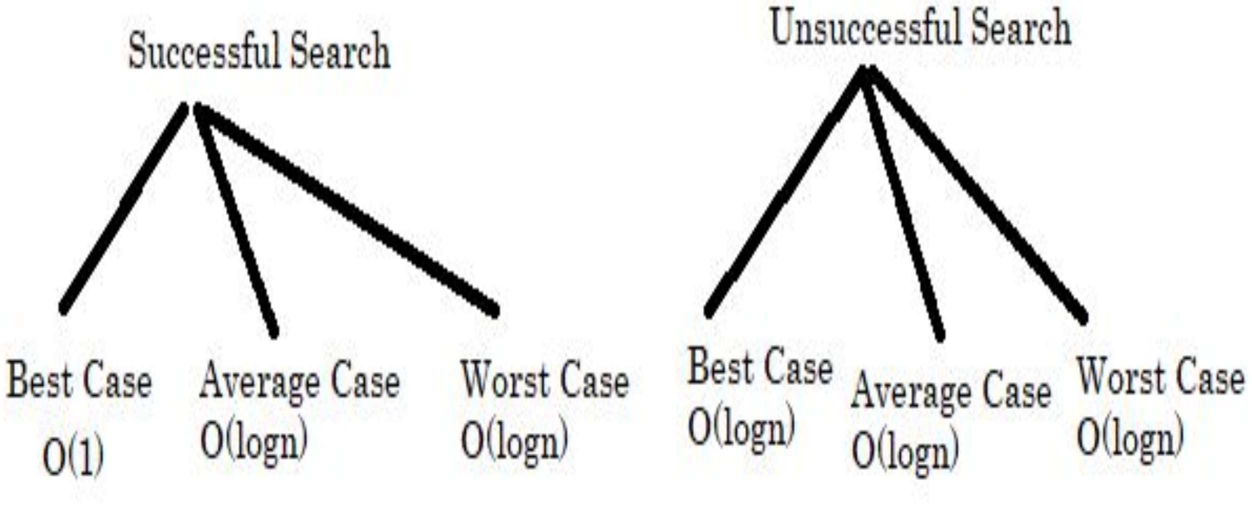
$$N/2^k = 1$$

$$N = 2^k$$

$$\log_2 N = k \log_2 2$$

$$\boxed{K = \log_2 N}$$

Search element is not present in the array.


Successful Search
Best Case
O(1)
Average Case
O(logn)
Worst Case
O(logn)
Unsuccessful Search
Best Case
O(logn)
Average Case
O(logn)
Worst Case
O(logn)

SORTING:

- Sorting is the operation of arranging the records of a table into some sequential order according to an ordering criterion.
- Sorting is nothing but arranging the data in ascending or descending order.
- Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

The two main criteria to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of a few data elements.
These algorithms do not require any extra space and sorting is said to happen in-place,
for example, within the array itself. This is called **in-place sorting**.

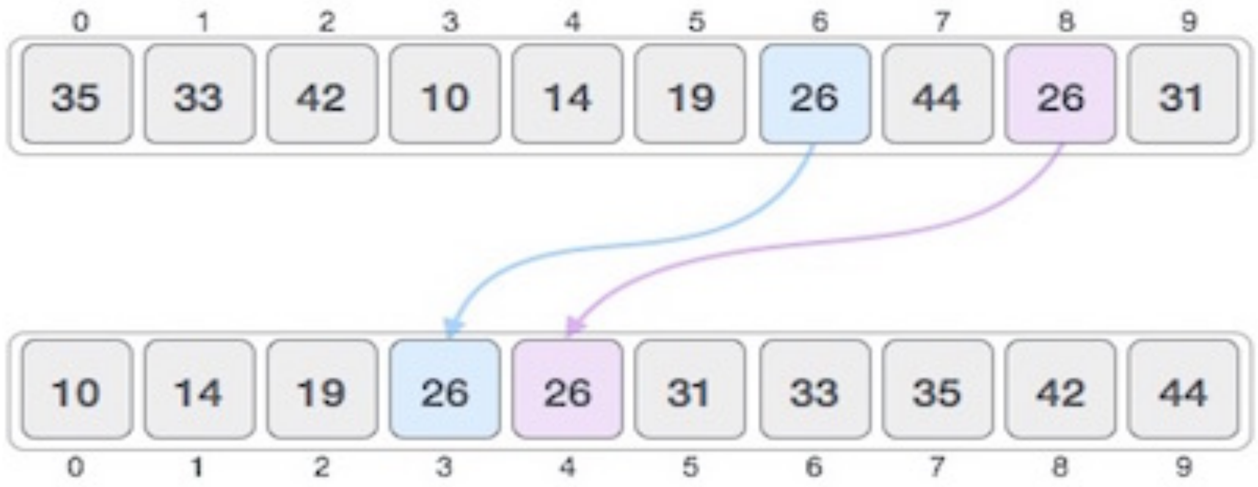
Bubble sort is an example of in-place sorting.

However, in some sorting algorithms, the program requires that which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**.

Merge-sort is an example of not-in-place sorting.

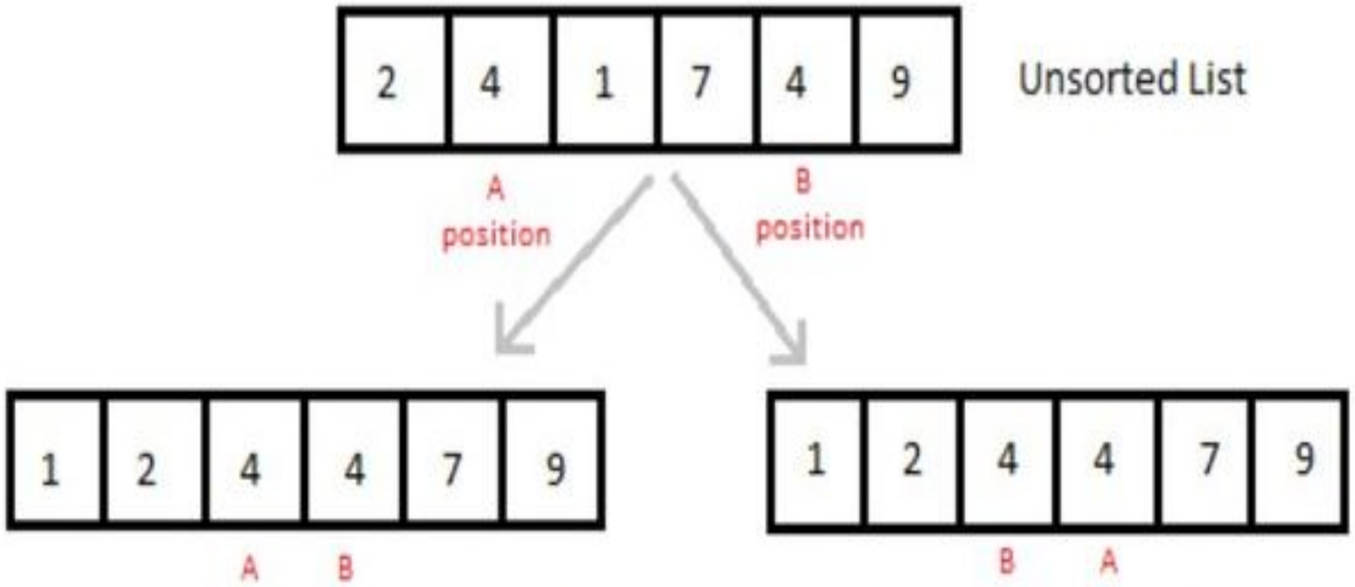
Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



0	1	2	3	4	5	6	7	8	9
35	33	42	10	14	19	26	44	26	31

10	14	19	26	26	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9



Stable Sort, because the order of equal elements is maintained in sorted list.

UnStable Sort, because the order of equal elements is not maintained in the sorted list.

MERGE SORT

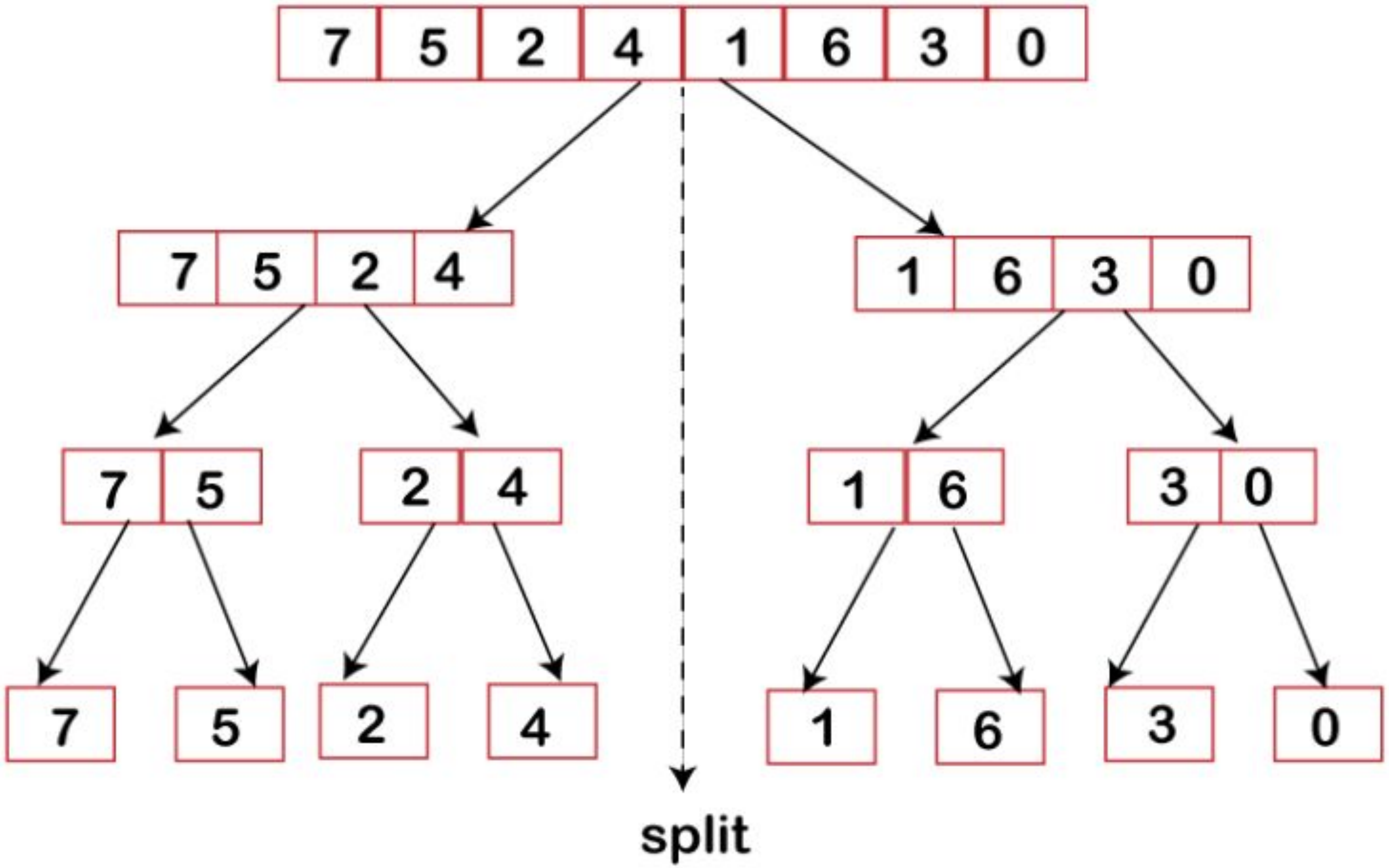
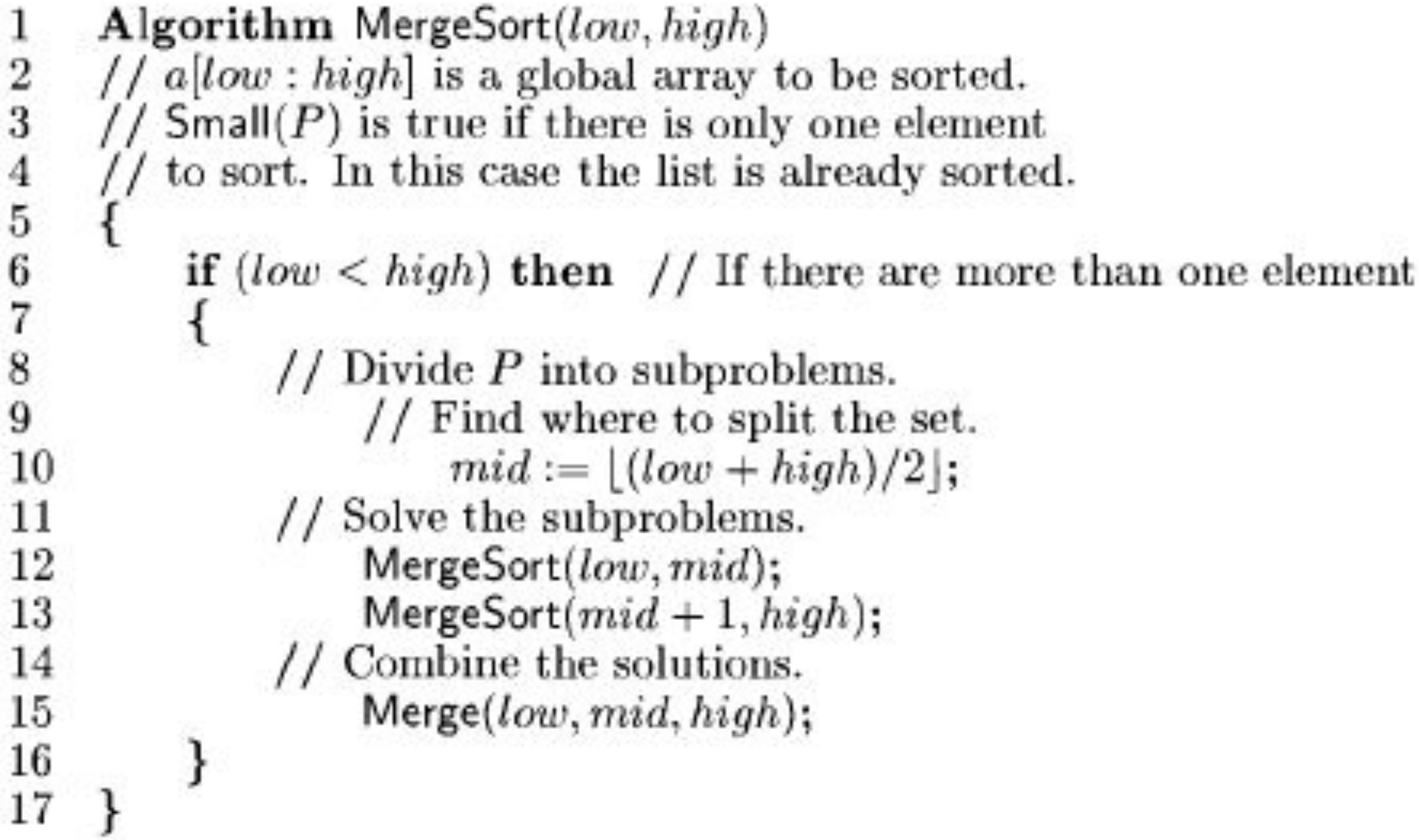


Figure 1: Merge Sort Divide Phase

MERGE SORT



```
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  // If there are more than one element
    {
        // Divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
            MergeSort($low, mid$);
            MergeSort($mid + 1, high$);
        // Combine the solutions.
            Merge($low, mid, high$);
    }
}
```

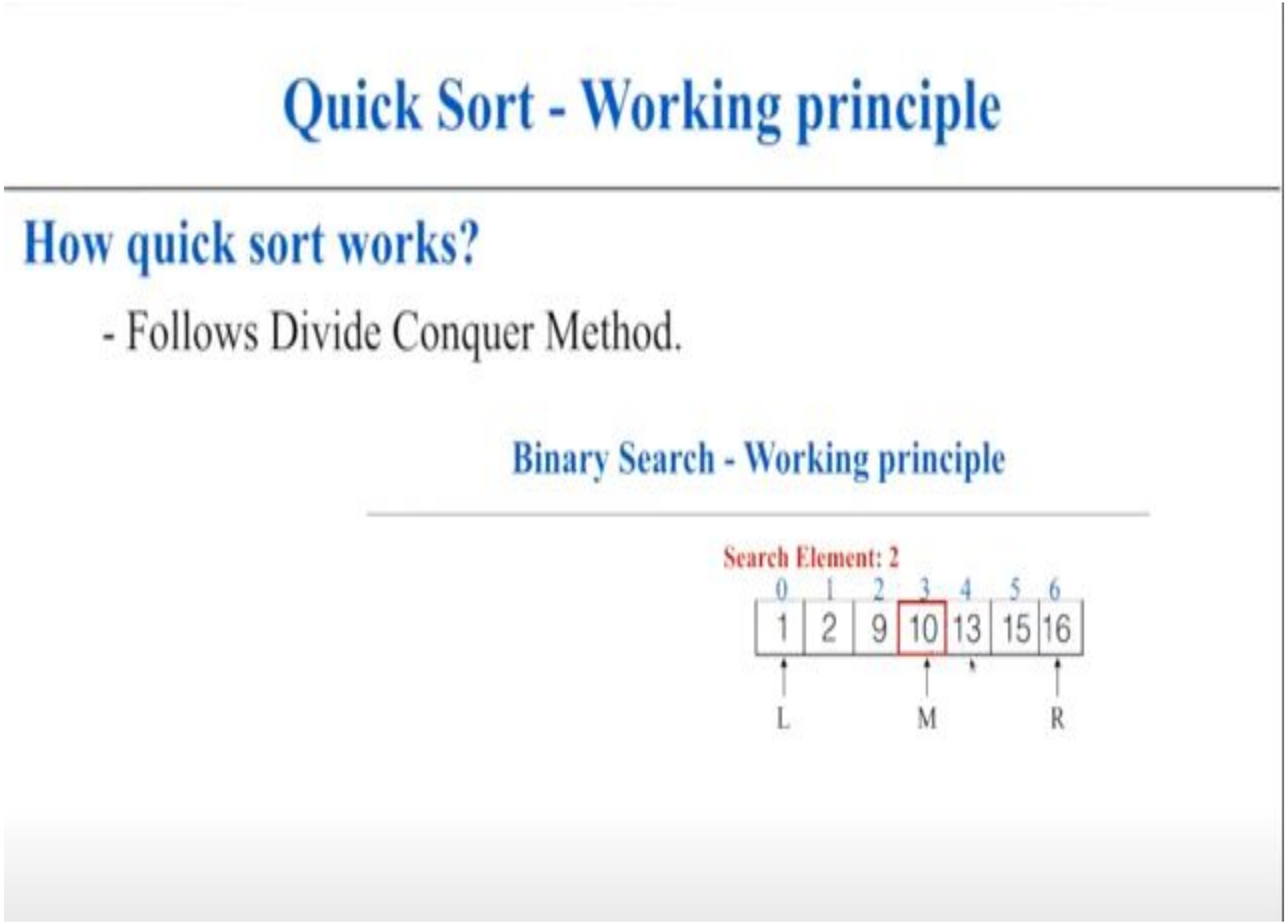
```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
    $h := low$; $i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

(310 | 285 | 179 | 652, 351 | 423, 861, 254, 450, 520)

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$



Quick Sort - Working principle

How quick sort works?

- Follows Divide Conquer Method.

Binary Search - Working principle

Search Element: 2

Quick Sort - Working principle

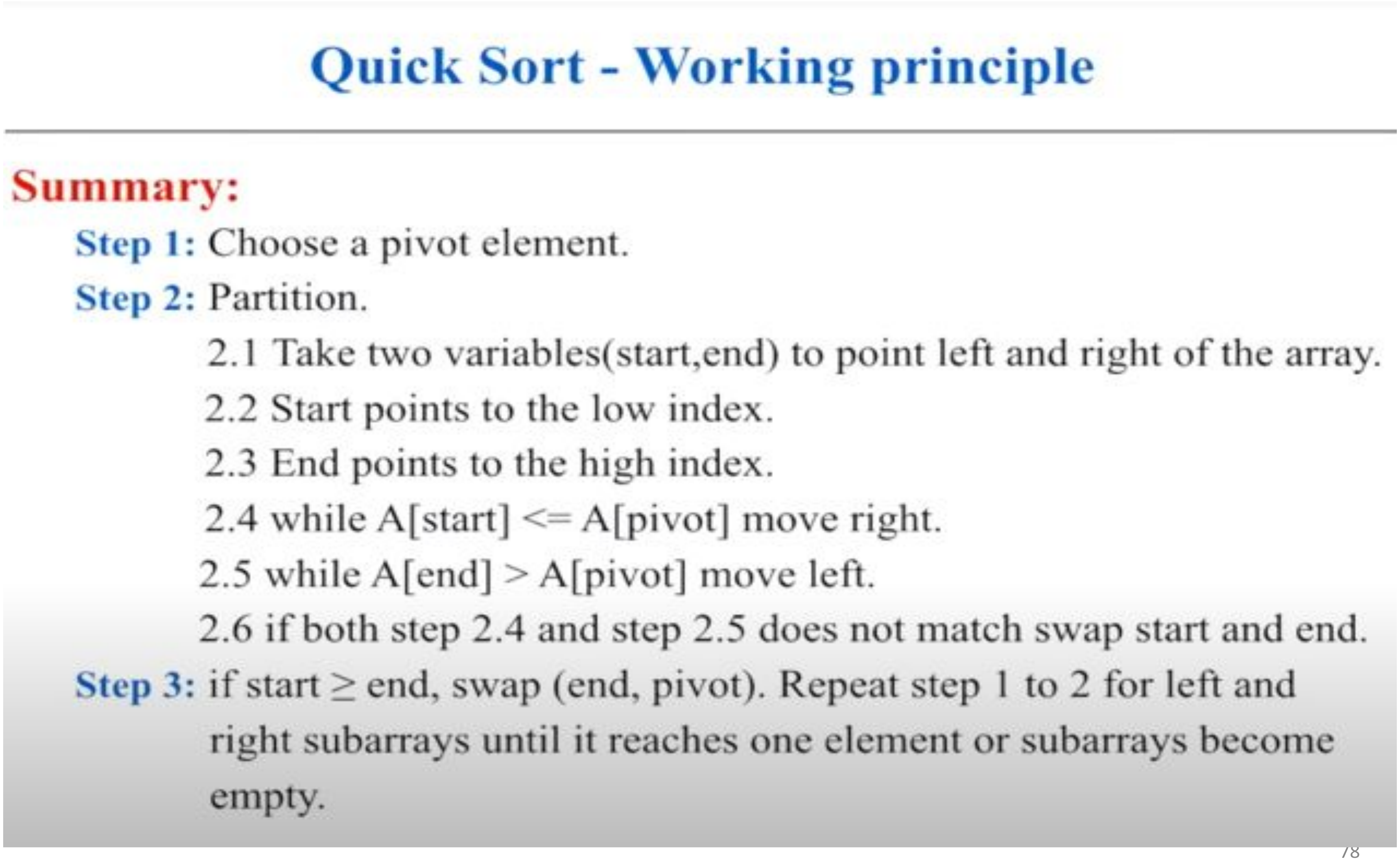
How quick sort works?

- Follows Divide Conquer Method.

	0	1	2	3	4	5	6
A	10	15	1	2	9	16	13

- Steps:

1. Choose pivot/key element.
    - Pick first element as pivot.
    - Pick last element as pivot.
    - Pick random element as pivot.
    - Pick median element as pivot.
2. Partitioning of array.
    - { left subarray < pivot < right subarray }
3. Repeat step 1 and 2 for left, right subarrays.



Quick Sort - Working principle

Summary:

Step 1: Choose a pivot element.

Step 2: Partition.

2.1 Take two variables(start,end) to point left and right of the array.

2.2 Start points to the low index.

2.3 End points to the high index.

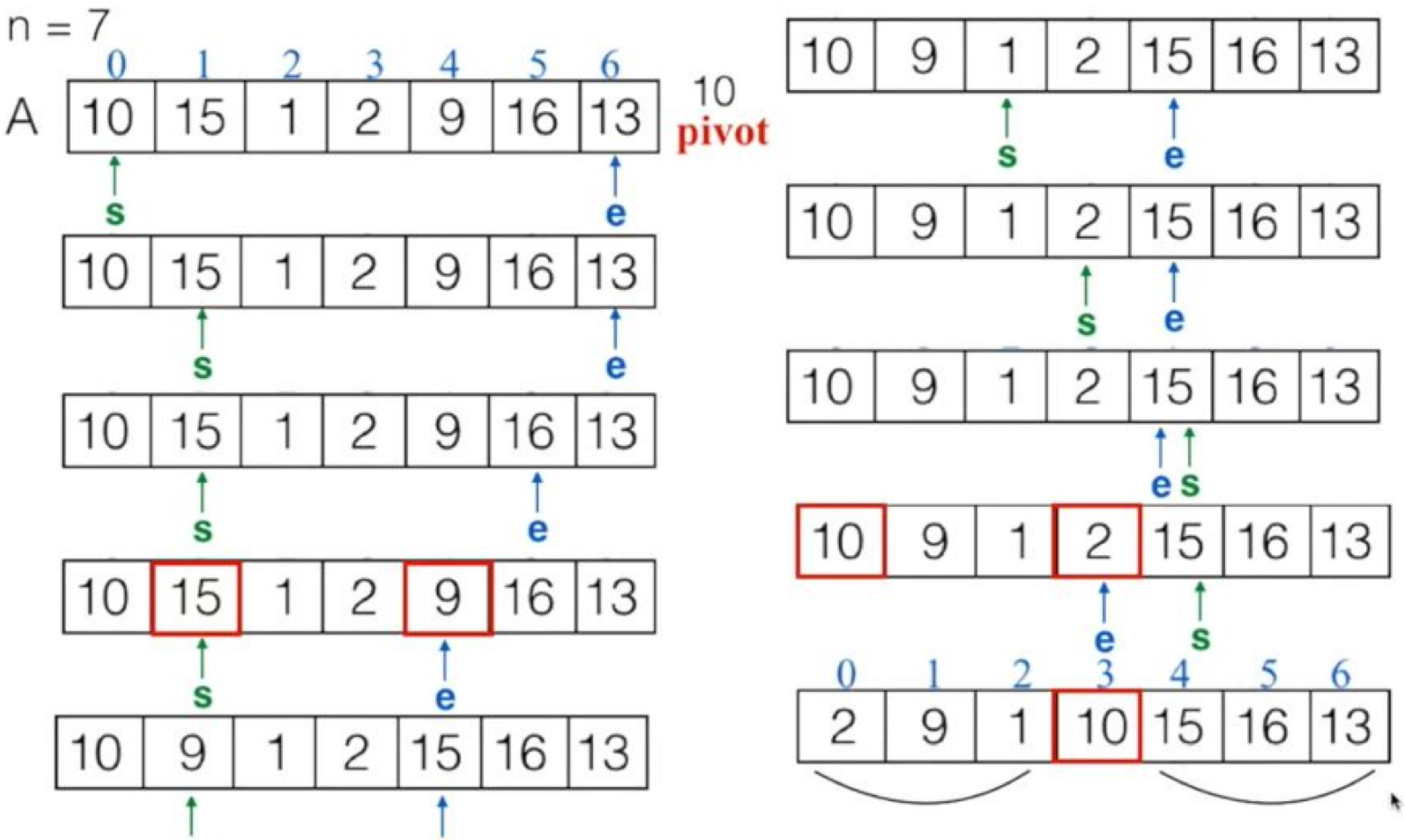
2.4 while A[start] <= A[pivot] move right.

2.5 while A[end] > A[pivot] move left.

2.6 if both step 2.4 and step 2.5 does not match swap start and end.

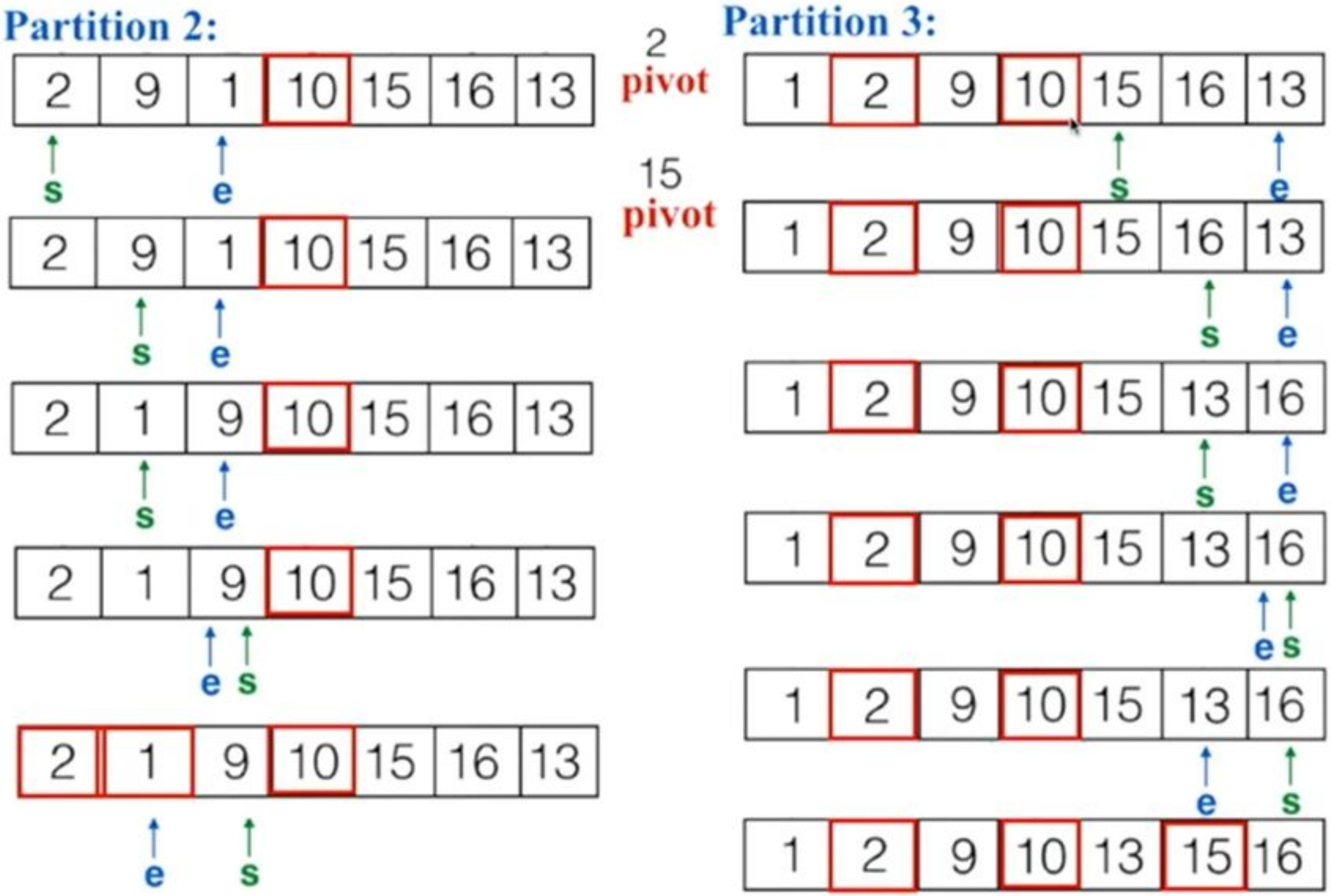
Step 3: if start ≥ end, swap (end, pivot). Repeat step 1 to 2 for left and right subarrays until it reaches one element or subarrays become empty.

Quick Sort - Working principle



Partition 1

Quick Sort - Working principle



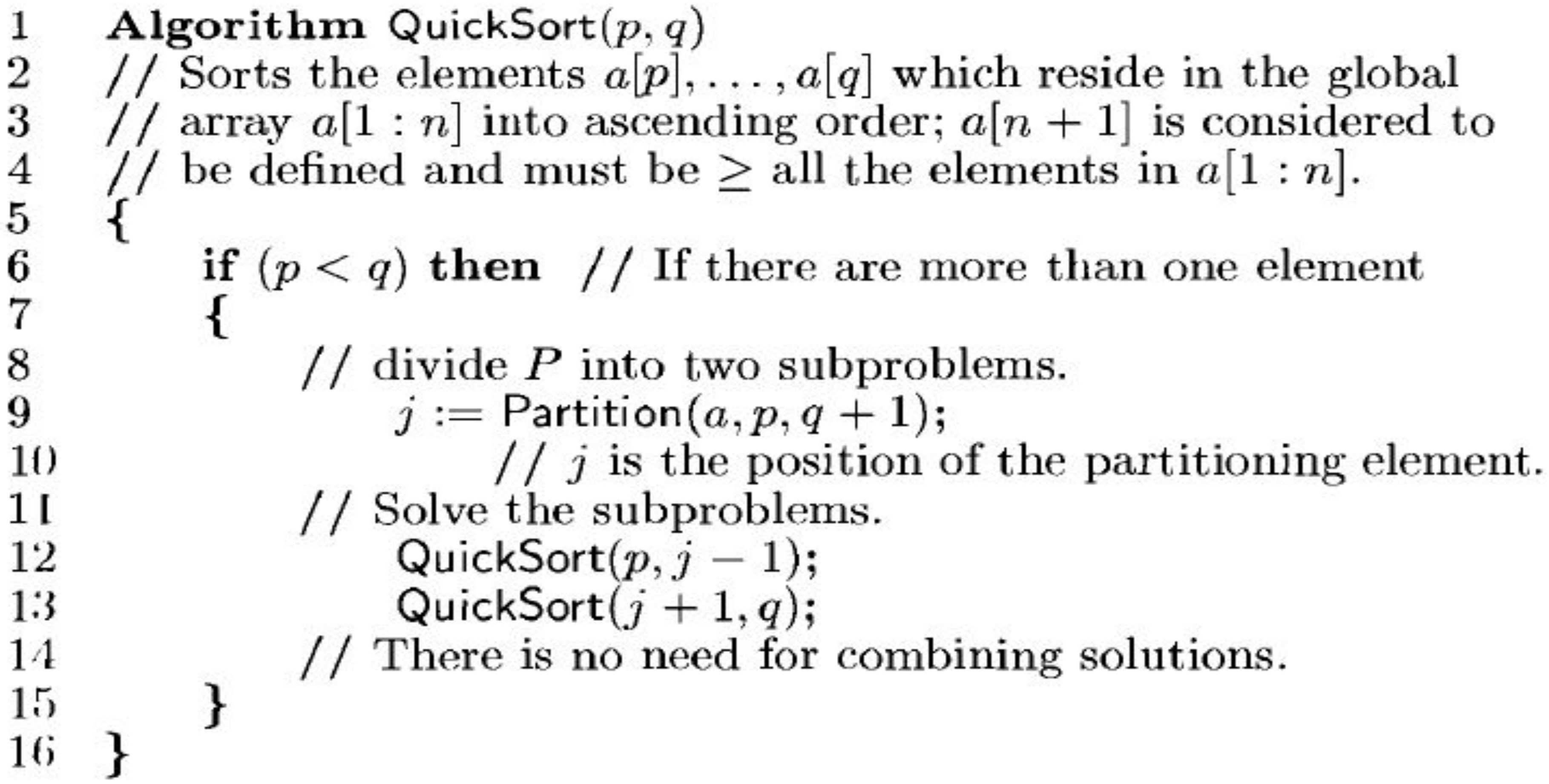
Partition 2:

2	9	1	10	15	16	13
2	9	1	10	15	16	13
2	1	9	10	15	16	13
2	1	9	10	15	16	13
2	1	9	10	15	16	13

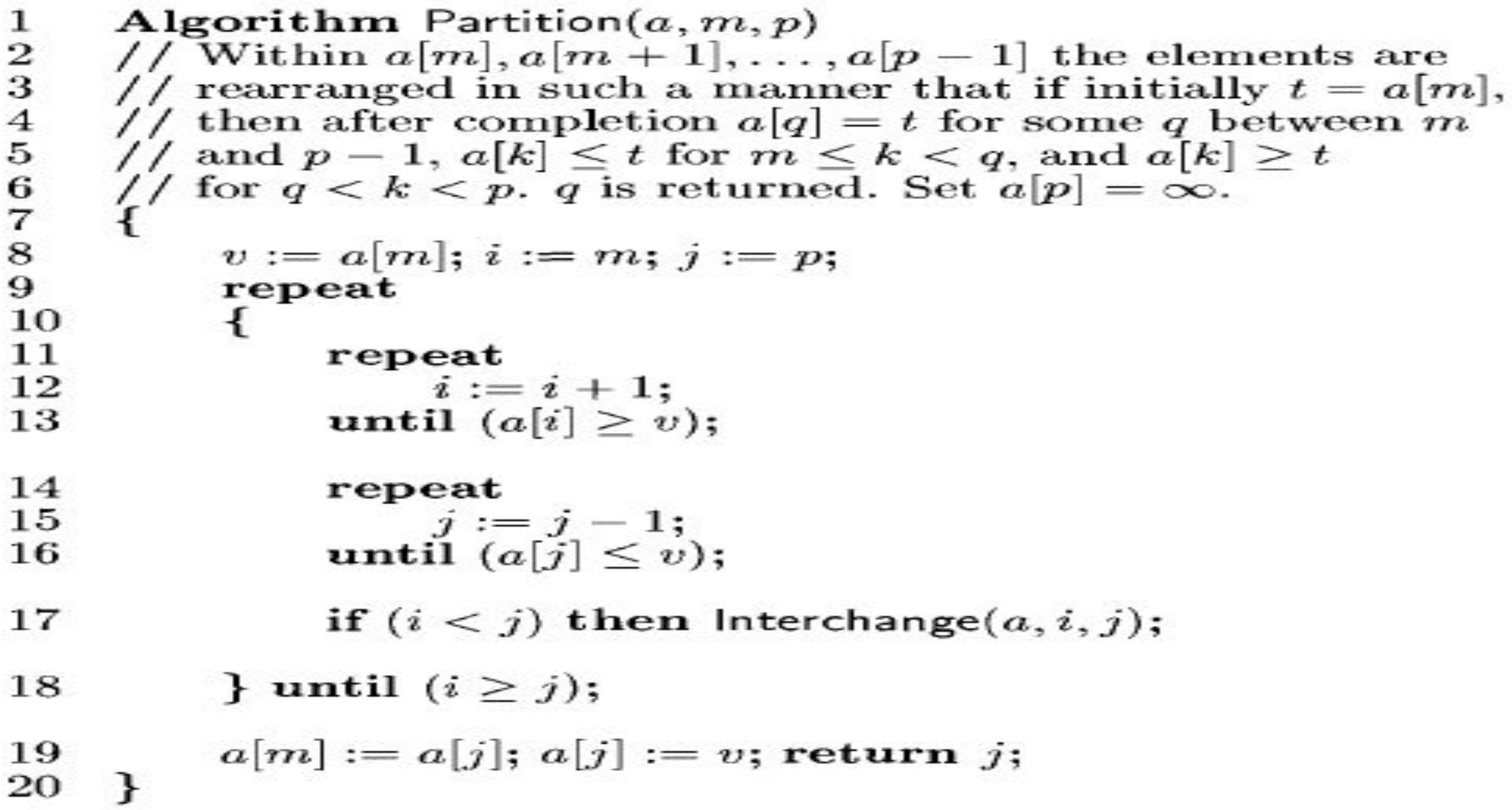
Partition 3:

1	2	9	10	15	16	13
1	2	9	10	15	16	13
1	2	9	10	15	13	16
1	2	9	10	15	13	16
1	2	9	10	15	13	16
1	2	9	10	13	15	16

QUICKSORT



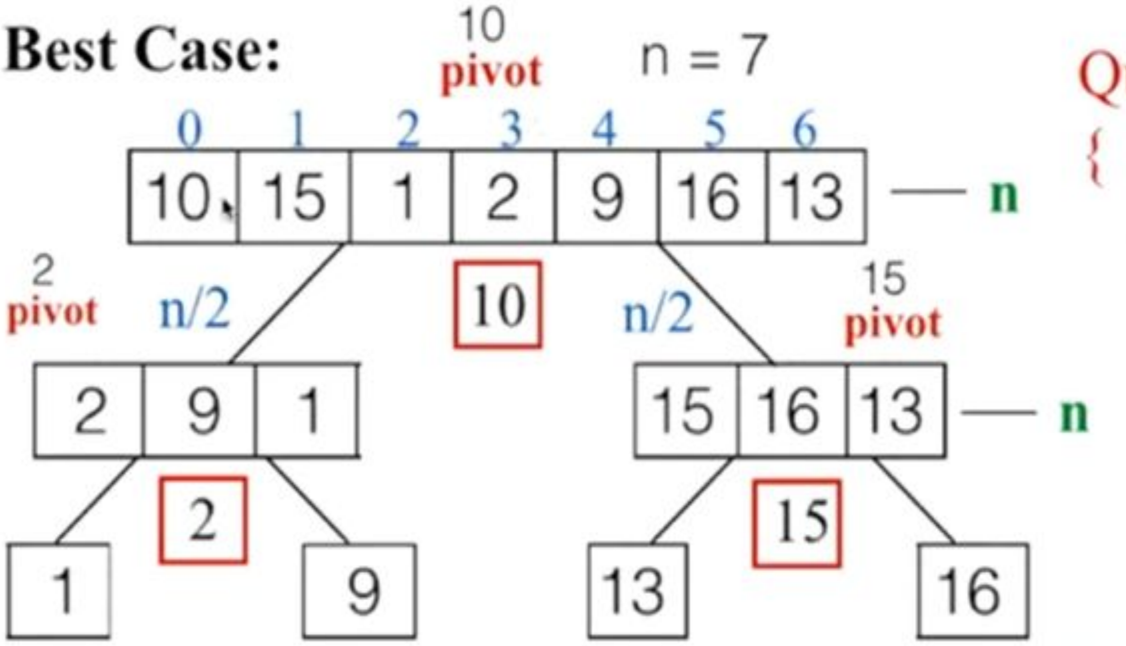
```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j :=$ Partition($a, p, q+1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j-1$);
            QuickSort($j+1, q$);
        // There is no need for combining solutions.
    }
}
```



```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}
```

Analysis of Quick Sort

Best Case:

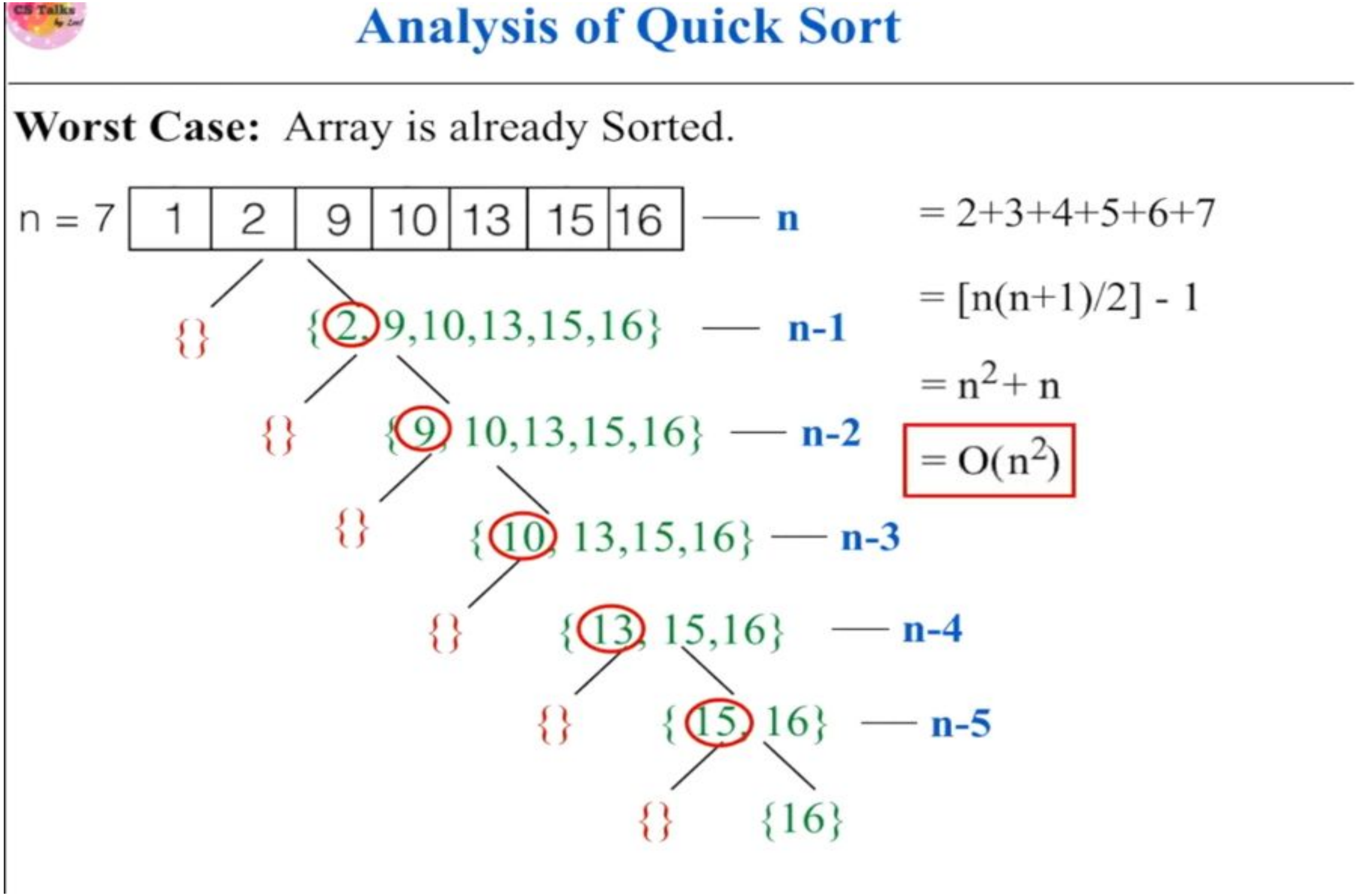


```
QuickSort(l,h)
{
  if(l < h)
  {
     loc = partition(l,h);
     QuickSort(l, loc-1);
     QuickSort(loc+1, h);
  }
}
```

Time complexity of entire recursion tree : No.of levels * Time taken for one level

$\frac{n}{2^k} = 1$ | $\log_2 n = k \log_2 2$ $\boxed{\log_2 n * n}$ **Best Case: O (n * log n)**

$n = 2^k$ | $k = \log_2 n$



Analysis of Quick Sort

Worst Case: Array is already Sorted.

n = 7 [1 | 2 | 9 | 10 | 13 | 15 | 16] — **n**

{} {2,9,10,13,15,16} — **n-1**

{} {9,10,13,15,16} — **n-2**

{} {10,13,15,16} — **n-3**

{} {13,15,16} — **n-4**

{} {15,16} — **n-5**

{} {16}

$= 2+3+4+5+6+7$

$= [n(n+1)/2] - 1$

$= n^2 + n$

$\boxed{= O(n^2)}$

Average Case

$$T_A(n) = \frac{\text{Summation of all possibilities}}{\text{Total no. of. possibilities}}$$

$$\begin{array}{l}
T_A(0) + T_A(n-1) + (n+1) \\
T_A(1) + T_A(n-2) + (n+1) \\
T_A(2) + T_A(n-3) + (n+1) \\
\vdots \\
T_A(n-2) + T_A(1) + (n+1) \\
T_A(n-1) + T_A(0) + (n+1) \\
\hline
2\left[T_A(0) + T_A(1) + \cdots + T_A(n-1)\right] + n.(n+1)
\end{array}$$

$$T_A(n) = \frac{2\left[T_A(0) + T_A(1) + \cdots + T_A(n-1)\right] + n(n+1)}{n}$$

$$n.T_A(n) = 2\left[T_A(0) + T_A(1) + \cdots + T_A(n-1)\right] + n(n+1)$$

$n = n-1$

$$n \cdot T_A(n) = 2\left[T_A(0) + T_A(1) + \cdots + T_A(n-1)\right] + n(n+1) \quad — \textcircled{1}$$

Same Equation for $n = n-1$

$$(n-1) \cdot T_A(n-1) = 2\left[T_A(0) + T_A(1) + \cdots + T_A(n-2)\right] + (n-1) \cdot n \quad — \textcircled{2}$$

Eq $\textcircled{1} - \textcircled{2}$

$$n T_A(n) - (n-1) T_A(n-1) = 2 T_A(n-1) + 2n$$

$$n \cdot T_A(n) = 2 T_A(n-1) + (n-1) T_A(n-1) + 2n$$

$$n \cdot T_A(n) = (n+1) \cdot T_A(n-1) + 2n$$

divide both Sides by n.(n+1)

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

$$\frac{1}{n+1} \cdot T_A(n) = \frac{2}{n+1} + \frac{T_A(n-1)}{\checkmark\; n}$$

$$= \boxed{\frac{T_A(n-2)}{n-1} + \frac{2}{n}} + \frac{2}{n+1}$$

$$= \frac{T_A(n-3)}{n-2} + \frac{2}{(n-1)} + \frac{2}{n} + \frac{2}{n+1}$$

Do Um

4

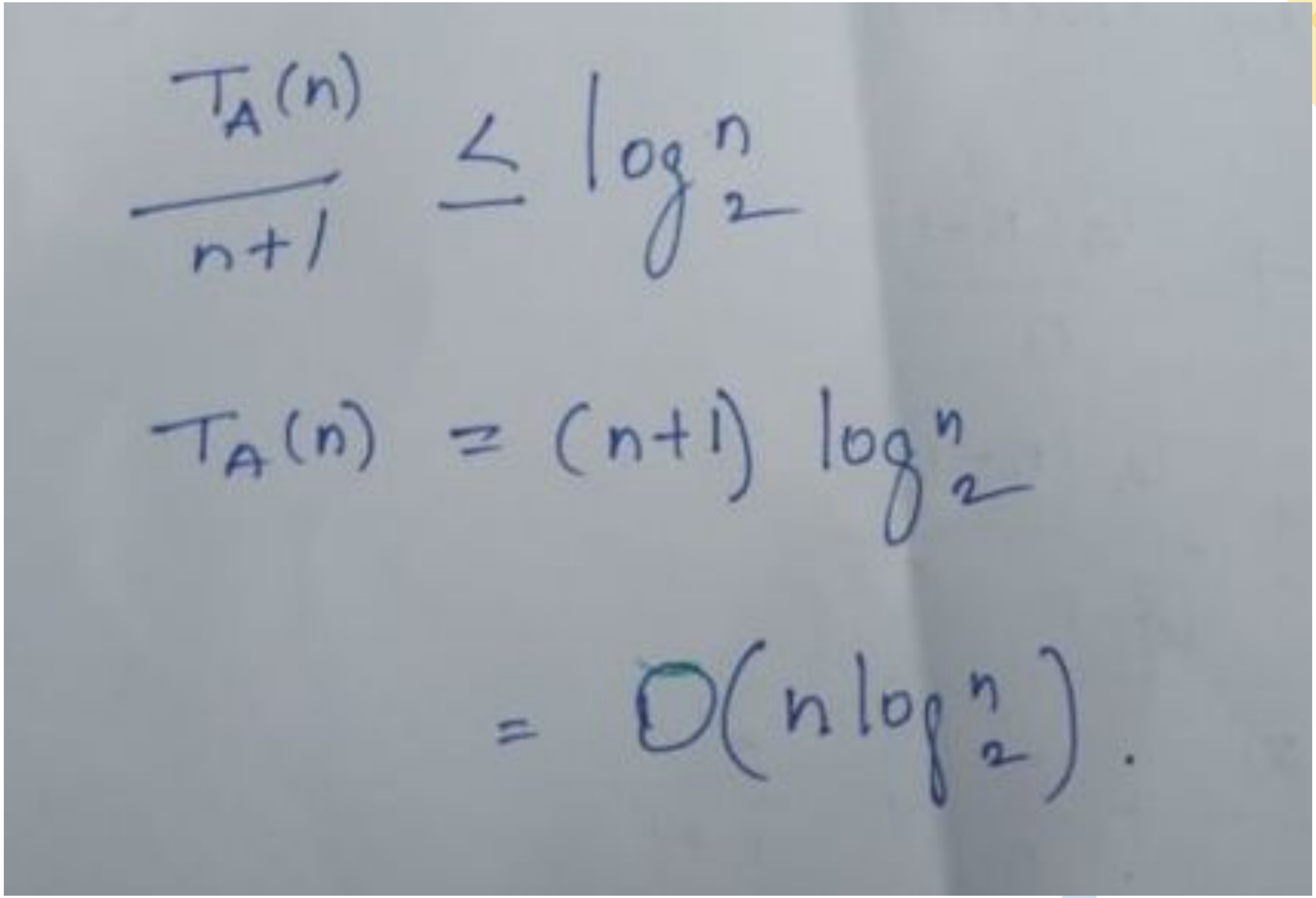
$$= \boxed{\frac{T_A(1)}{2}} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \cdots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

Value 0

$$= 2\left[\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}\right]$$

$$= 2 \sum_{3 \le K \le n+1} \frac{1}{K} = 2 \cdot \int_{3}^{n+1} \frac{1}{K} \cdot dK$$

$$= 2 \cdot \Big[\log K\Big]_{3}^{n+1} =$$



$$\frac{T_A(n)}{n+1} \le \log_2^n$$

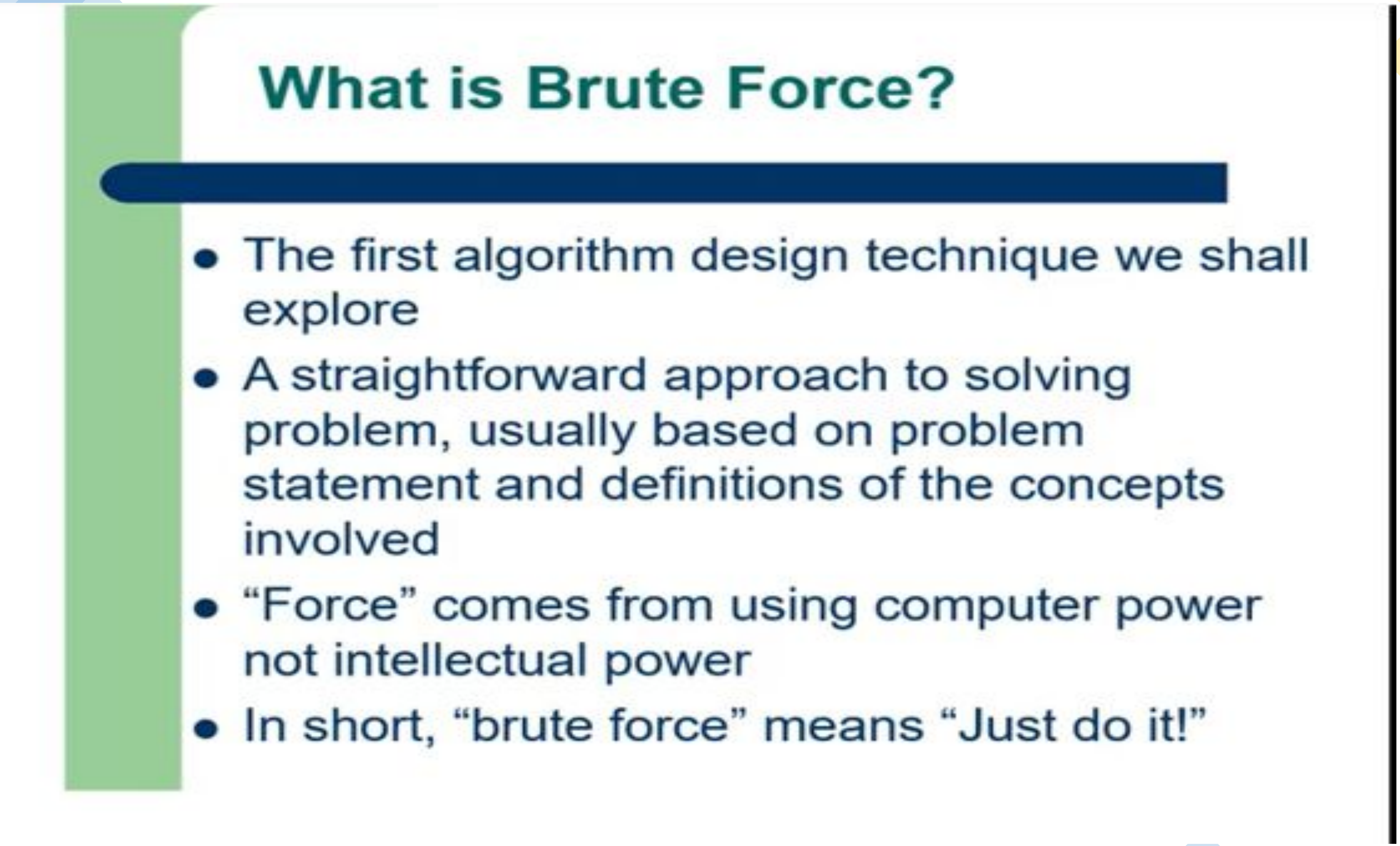
$$T_A(n) = (n+1)\log_2^n$$

$$= O(n\log_2^n).$$

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

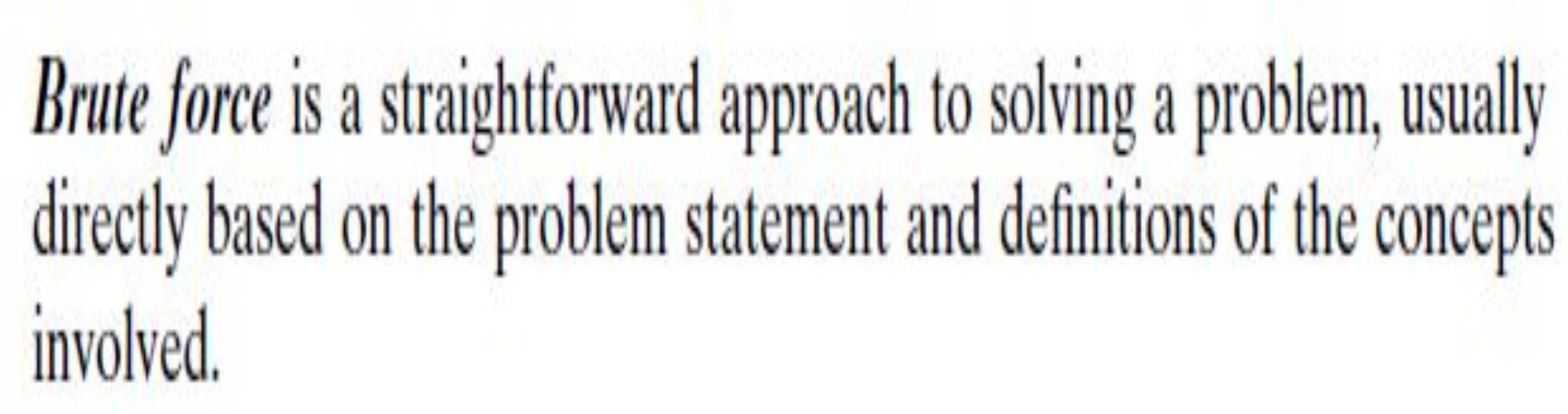
$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} [C_A(k-1)) + C_A(n-k)]$$



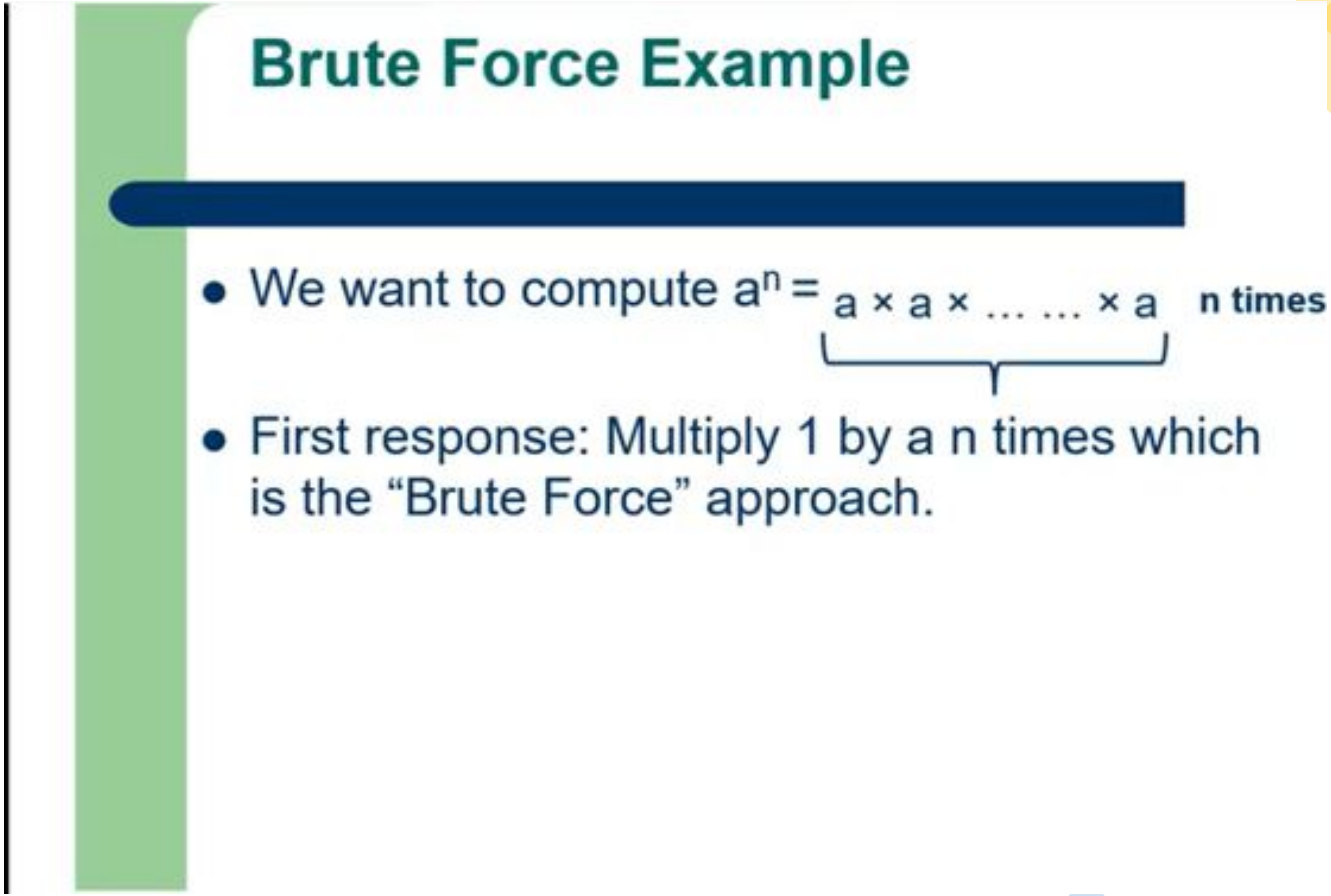
What is Brute Force?

- The first algorithm design technique we shall explore
- A straightforward approach to solving problem, usually based on problem statement and definitions of the concepts involved
- “Force” comes from using computer power not intellectual power
- In short, “brute force” means “Just do it!”

Brute Force



Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.



Brute Force Example

- We want to compute $a^n = \underbrace{a \times a \times \ldots \ldots \times a}$ **n times**
- First response: Multiply 1 by a n times which is the “Brute Force” approach.

Brute Force: Straight Forward Approach

(i) Sorting: Bubble, Selection
(ii) Searching: Linear, String Matching
(iii) Closest Pair: convex hull problem
(iv) Exhaustive Search

An exhaustive search is simply **a brute-force approach to combinatorial problems such as Permutations, Combinations, Subsets of a set, etc**. It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding the desired element (e.g., the one that optimizes some objective function)

Traveling Sales Person(TSP)
0/1 Knapsack Problem
Assignment Problem

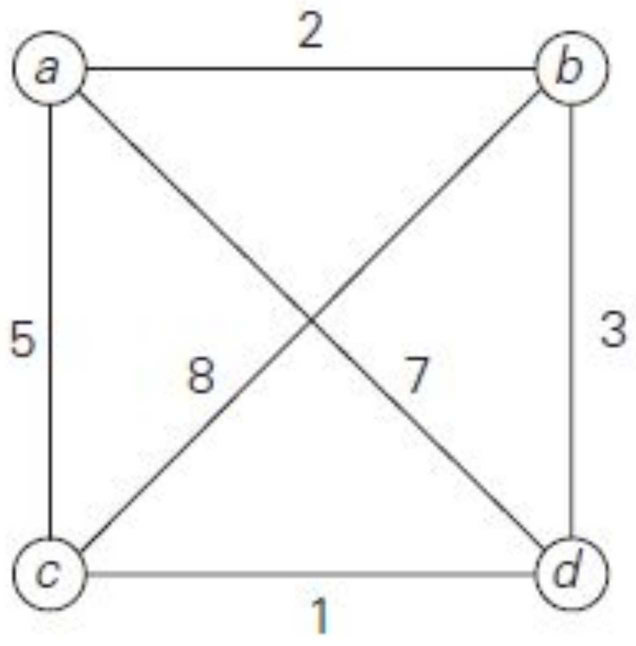
Travelling Sales Person Problem

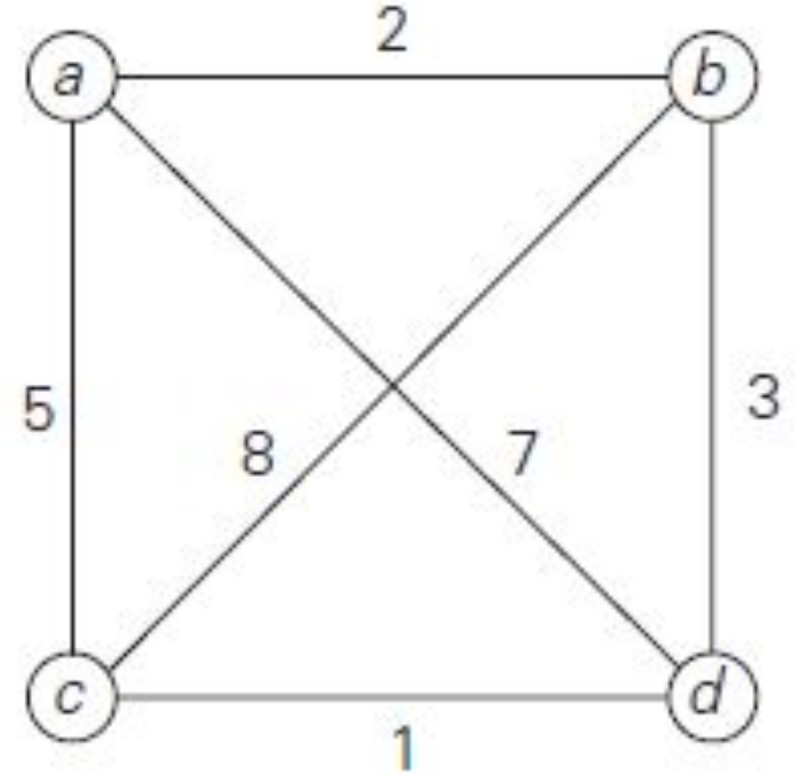
- The problem asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started.
- The problem can be conveniently modeled by a weighted graph, with the graph’s vertices representing the cities and the edge weights specifying the distances.
- Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

- It is easy to see that a Hamiltonian circuit can also be defined as a sequence of *n* + 1 adjacent vertices *vi*0*, vi*1*, . . . , vin*−1*, vi*0*,* where the first vertex of the sequence is the same as the last one and all the other *n* − 1 vertices are distinct.

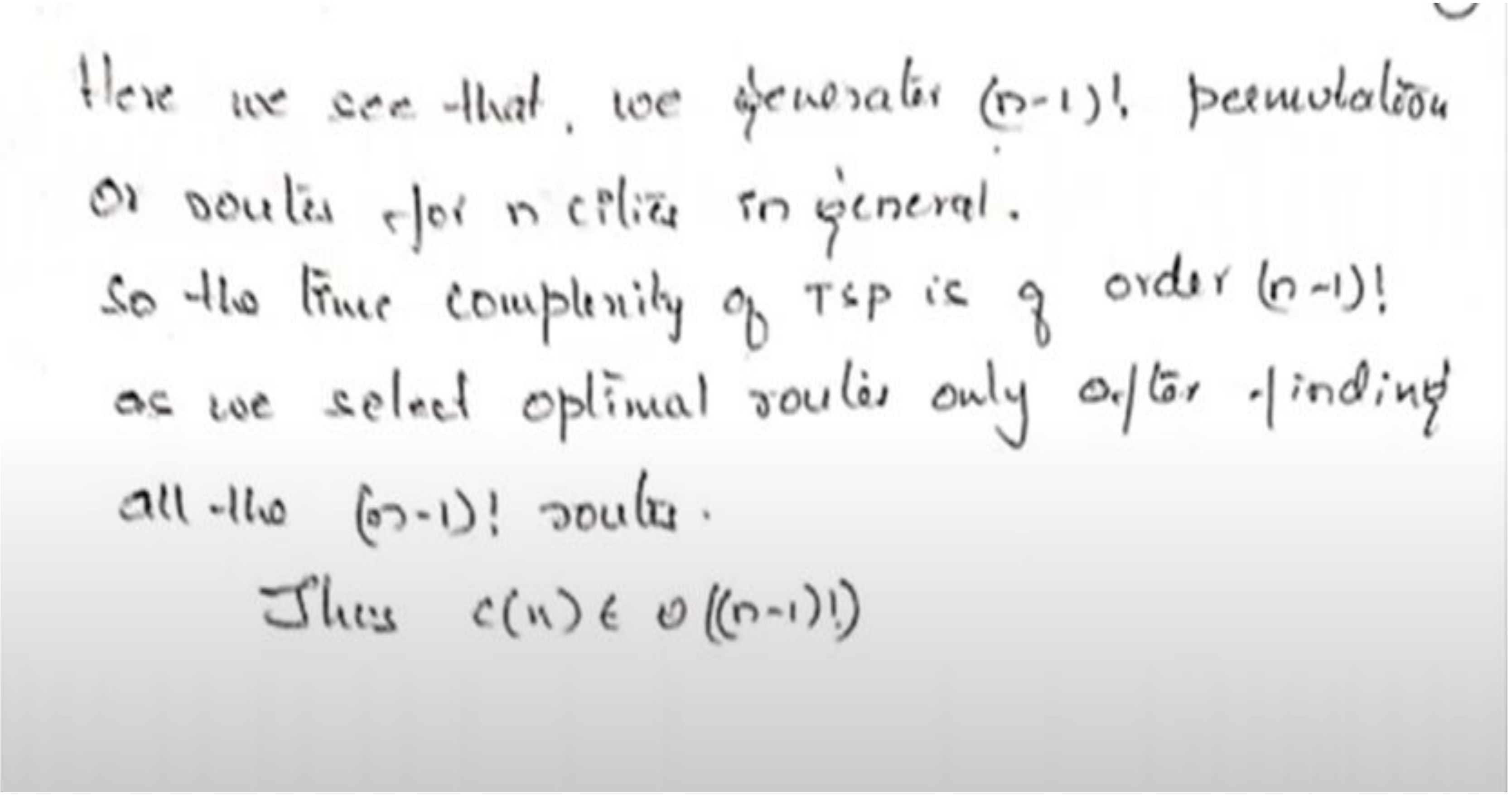
- Further, we can assume, with no loss of generality, that all circuits start and end at one particular vertex.

- Thus, we can get all the tours by generating all the permutations of *n* − 1 intermediate cities, compute the tour lengths, and find the shortest among them.


a
b
c
d
2
5
8
7
3
1



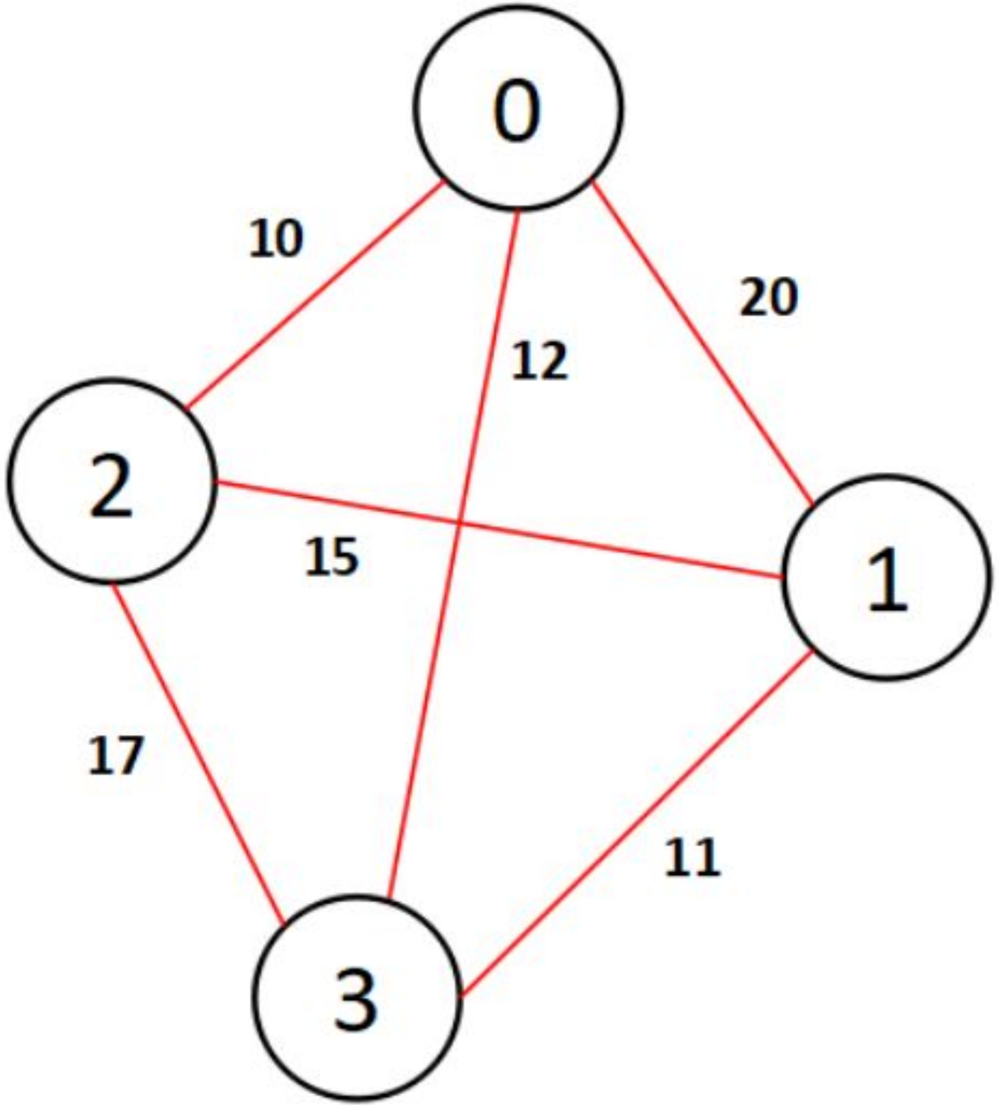
Tour	Length	
a --> *b* --> *c* --> *d* --> *a*	*l* = 2 + 8 + 1 + 7 = 18	
a --> *b* --> *d* --> *c* --> *a*	*l* = 2 + 3 + 1 + 5 = 11	optimal
a --> *c* --> *b* --> *d* --> *a*	*l* = 5 + 8 + 3 + 7 = 23	
a --> *c* --> *d* --> *b* --> *a*	*l* = 5 + 1 + 3 + 2 = 11	optimal
a --> *d* --> *b* --> *c* --> *a*	*l* = 7 + 3 + 8 + 5 = 23	
a --> *d* --> *c* --> *b* --> *a*	*l* = 7 + 1 + 8 + 2 = 18	



Here we see that, we generates $(n-1)!$ permutation or routes for n cities in general.

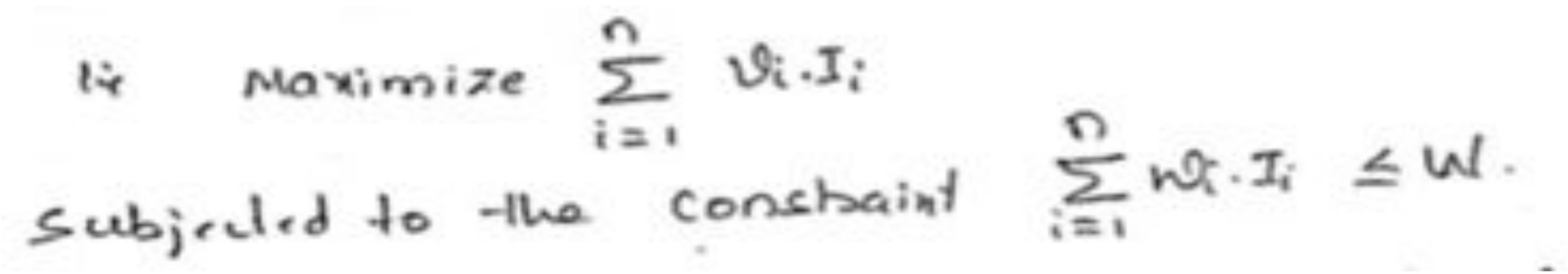
So the time complexity of TSP is of order $(n-1)!$ as we select optimal routes only after finding all the $(n-1)!$ routes.

Thus $c(n) \in \Theta((n-1)!)$


0
1
2
3
10
20
12
15
17
11

Knapsack problem

- Given *n* items of known weights *w*1*, w*2*, . . . , wn* and Profit values *v*1*, v*2*, . . . , vn* and a knapsack of capacity *W*, for each item I1, I2,……In

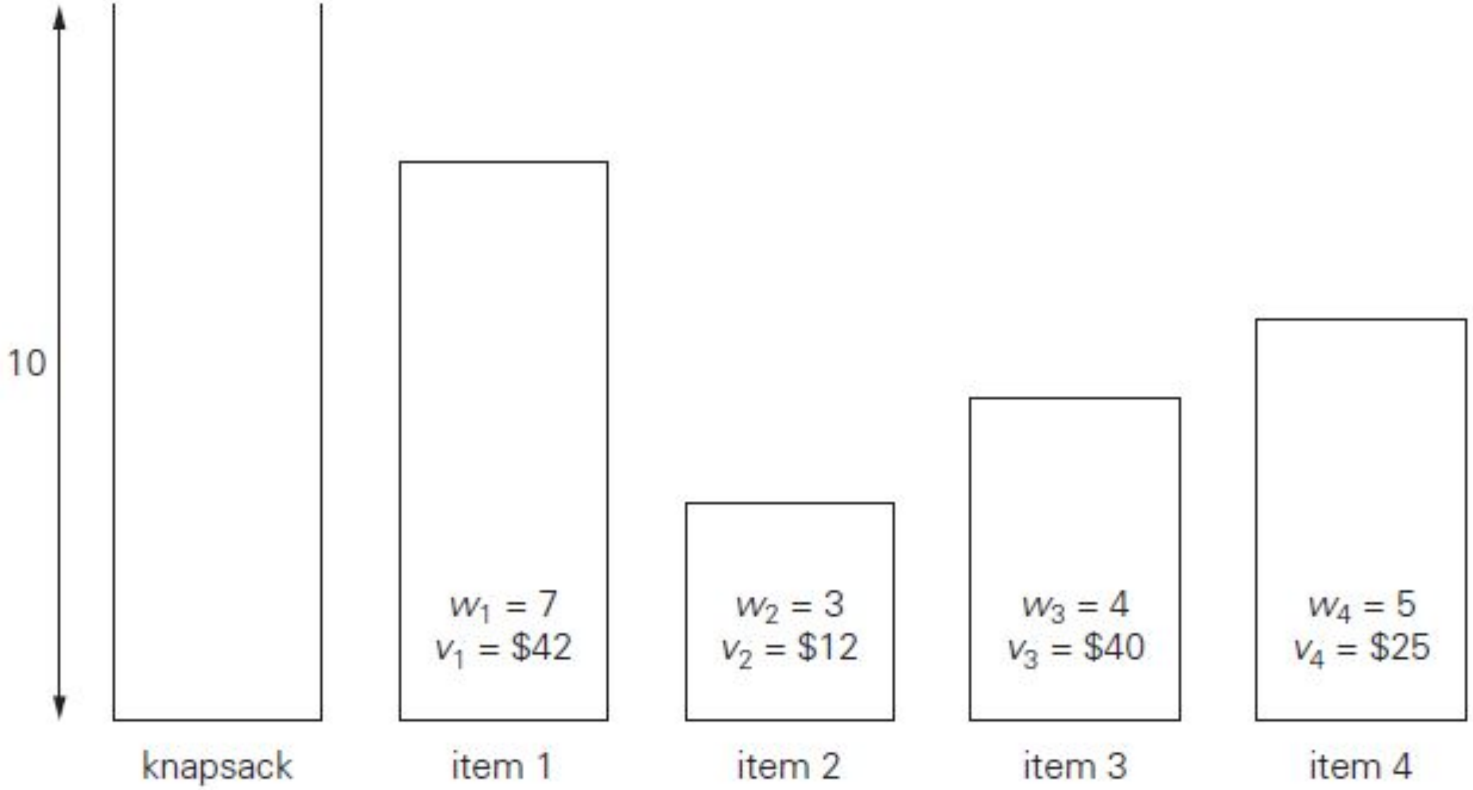


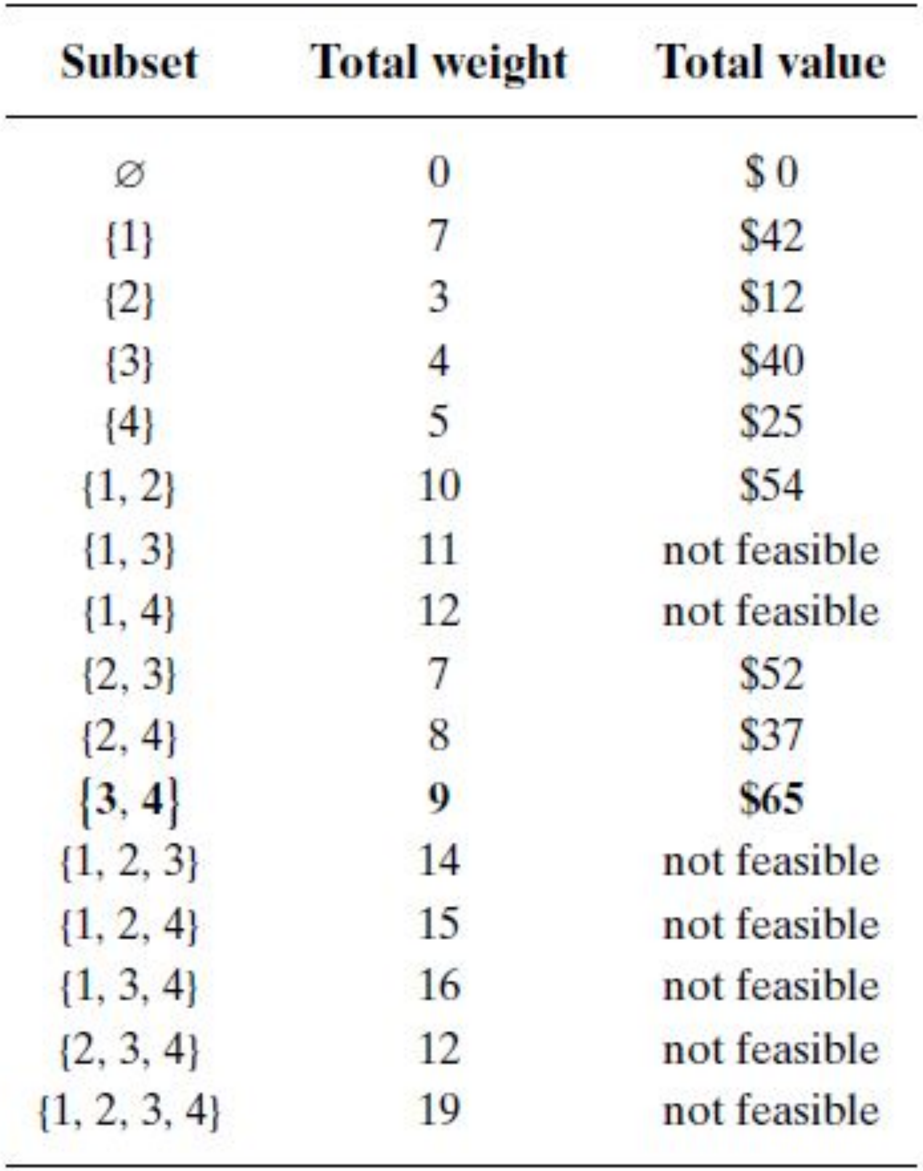
ie Maximize $\sum_{i=1}^{n} v_i \cdot I_i$

subjected to the constraint $\sum_{i=1}^{n} w_i \cdot I_i \leq W.$

- we have to find the most valuable subset of the items that fit into the knapsack.

- The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity),
- and finding a subset of the largest value among them.

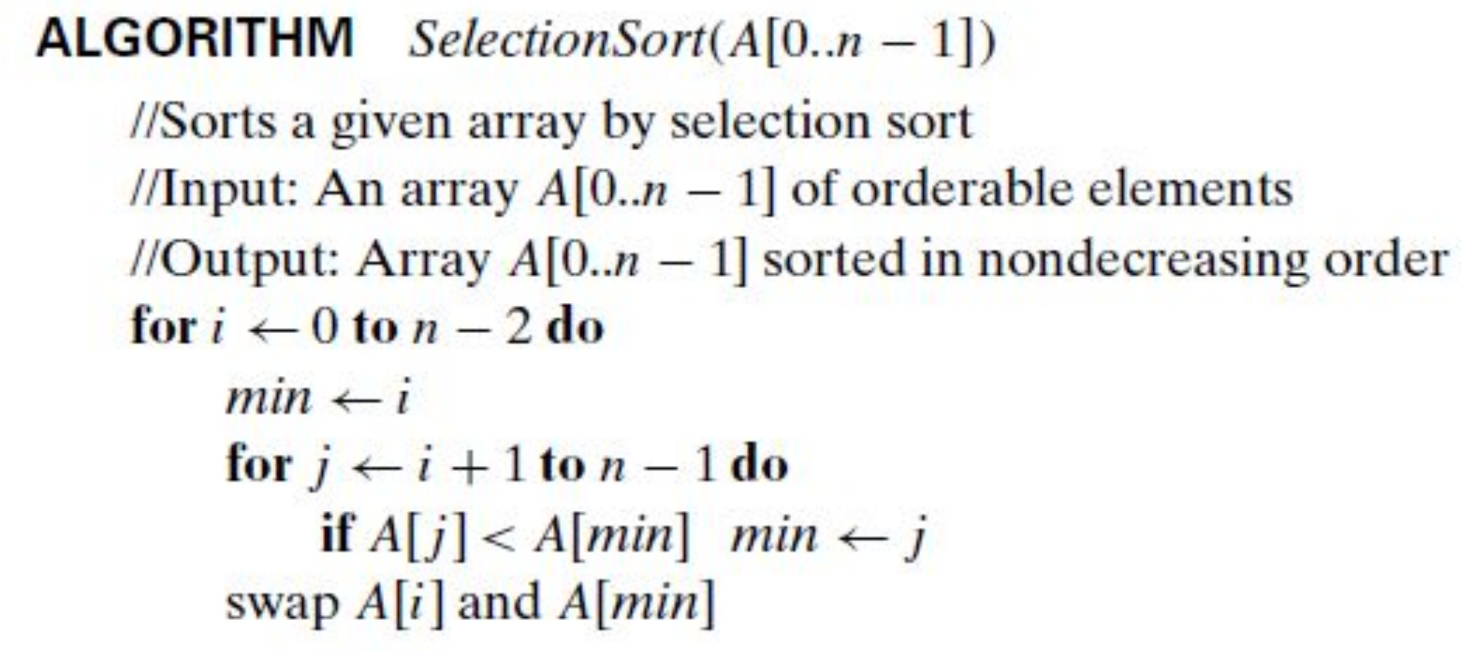

10
$w_1 = 7$
$v_1 = \$42$
$w_2 = 3$
$v_2 = \$12$
$w_3 = 4$
$v_3 = \$40$
$w_4 = 5$
$v_4 = \$25$
knapsack
item 1
item 2
item 3
item 4



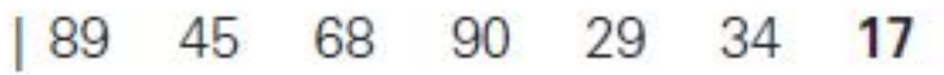
Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

For any knapsack problem with n items, we have to find out the possible subsets.
As we know that there will be 2^n subsets for a set containing “n” elements.
Hence time complexity of this problem is **$\Omega(2^n)$.**

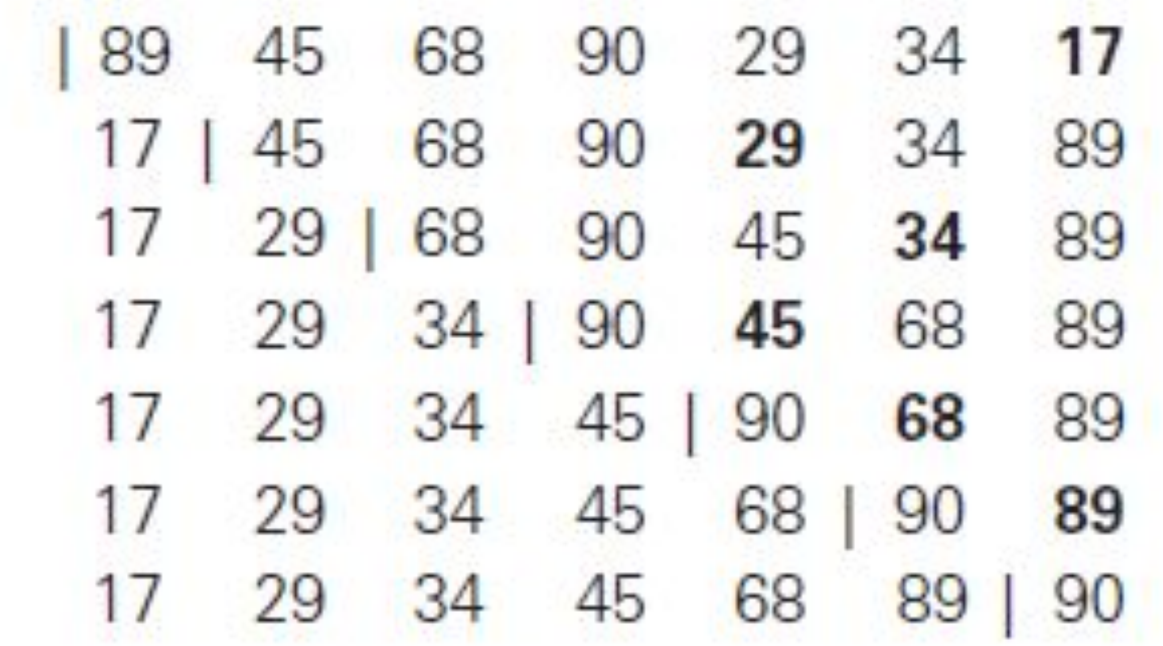
SELECTION SORT



```
ALGORITHM SelectionSort(A[0..n − 1])
    //Sorts a given array by selection sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min]   min ← j
        swap A[i] and A[min]
```



| 89 45 68 90 29 34 **17**



| 89 45 68 90 29 34 **17**

17 | 45 68 90 **29** 34 89

17 29 | 68 90 45 **34** 89

17 29 34 | 90 **45** 68 89

17 29 34 45 | 90 **68** 89

17 29 34 45 68 | 90 **89**

17 29 34 45 68 89 | 90

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 =$$

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

$$\sum_{i=l}^{u} 1 = u - l + 1$$

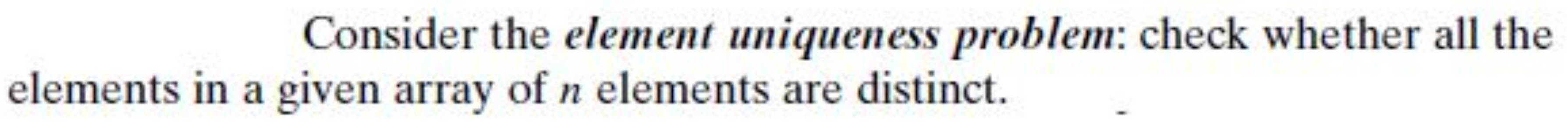
$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i,$$

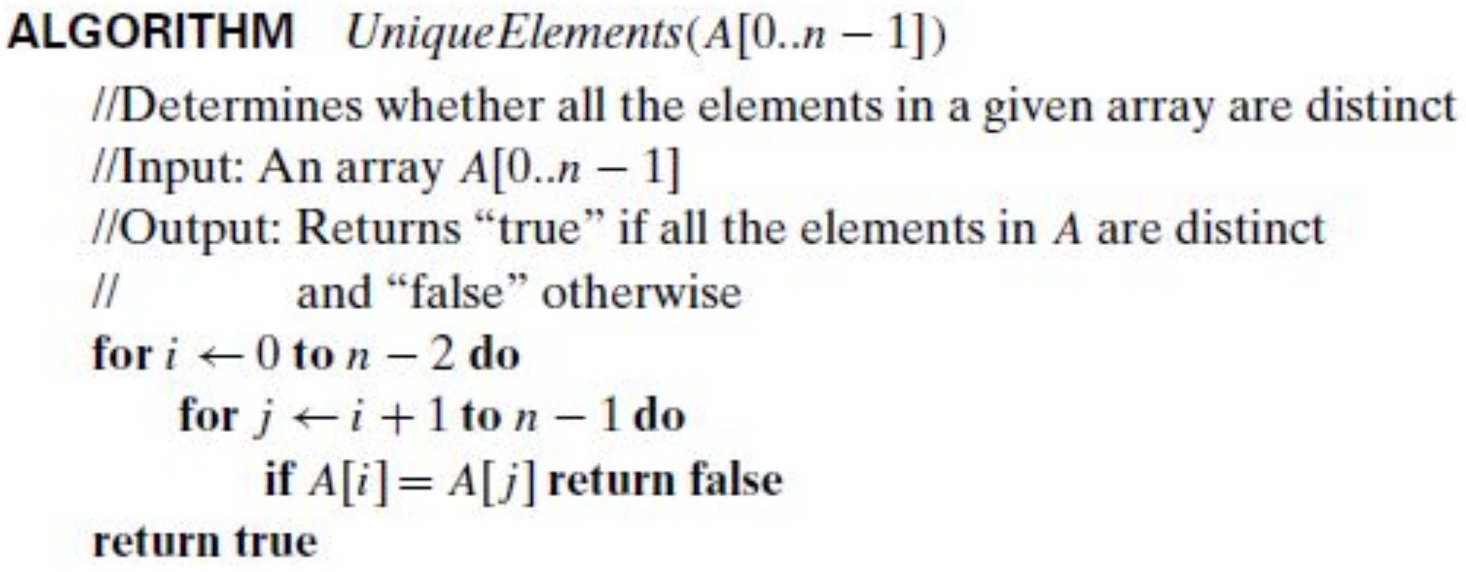
$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i,$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

ELEMENT UNIQUENESS PROBLEM



Consider the ***element uniqueness problem***: check whether all the elements in a given array of n elements are distinct.

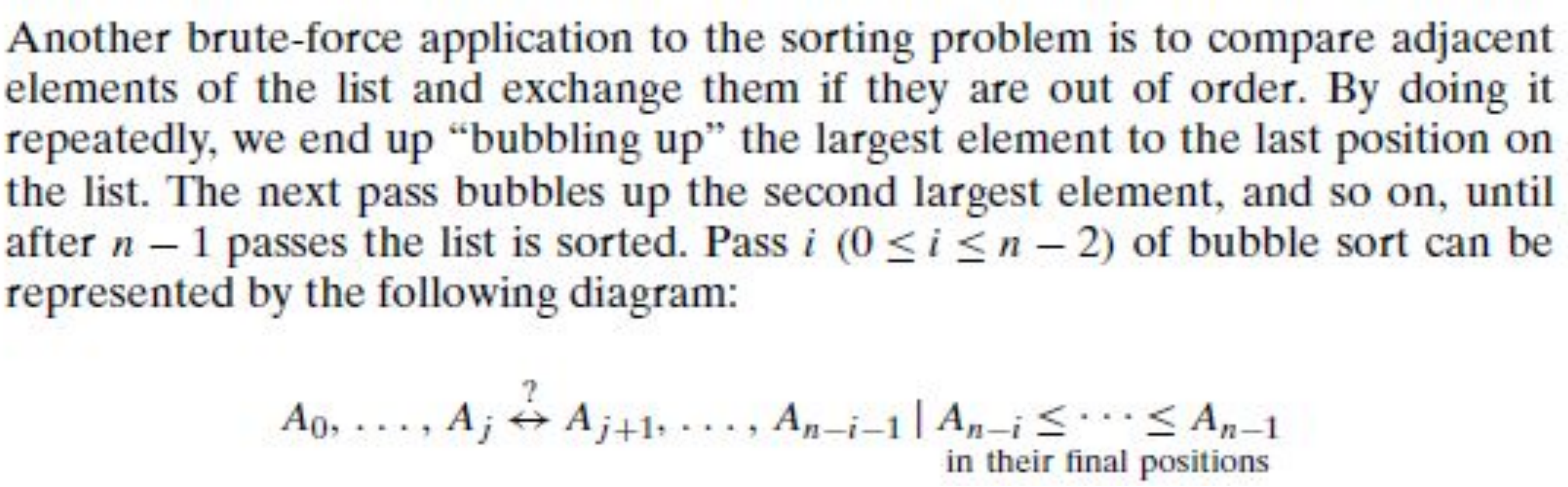


```
ALGORITHM  UniqueElements($A[0..n-1]$)
    //Determines whether all the elements in a given array are distinct
    //Input: An array $A[0..n-1]$
    //Output: Returns "true" if all the elements in $A$ are distinct
    //          and "false" otherwise
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[i] = A[j]$ return false
    return true
```

$$
\begin{aligned}
C\qquad(n) &= \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i) \\
&= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).
\end{aligned}
$$

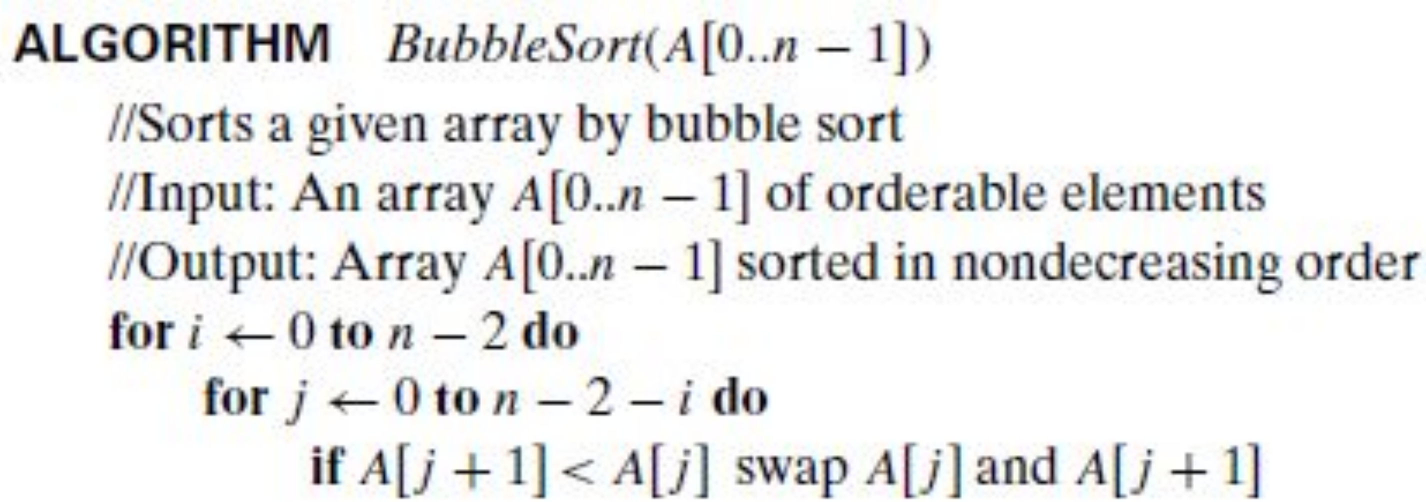
$$
\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},
$$

Bubble sort



Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted. Pass i $(0 \le i \le n - 2)$ of bubble sort can be represented by the following diagram:

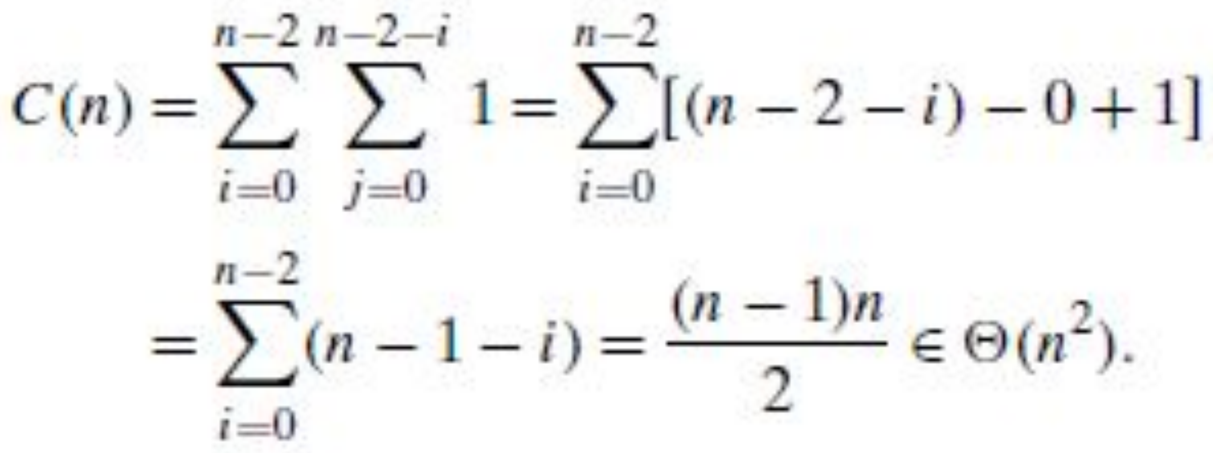
$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underset{\text{in their final positions}}{A_{n-i} \le \cdots \le A_{n-1}}$$



```
ALGORITHM  $BubbleSort(A[0..n-1])$
    //Sorts a given array by bubble sort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow 0$ to $n-2-i$ do
            if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$
```

$$\begin{array}{ccccccccccccc}
89 & \overset{?}{\leftrightarrow} & 45 & & 68 & & 90 & & 29 & & 34 & & 17 \\
45 & & 89 & \overset{?}{\leftrightarrow} & 68 & & 90 & & 29 & & 34 & & 17 \\
45 & & 68 & & 89 & \overset{?}{\leftrightarrow} & 90 & \overset{?}{\leftrightarrow} & 29 & & 34 & & 17 \\
45 & & 68 & & 89 & & 29 & & 90 & \overset{?}{\leftrightarrow} & 34 & & 17 \\
45 & & 68 & & 89 & & 29 & & 34 & & 90 & \overset{?}{\leftrightarrow} & 17 \\
45 & & 68 & & 89 & & 29 & & 34 & & 17 & | & 90
\end{array}$$

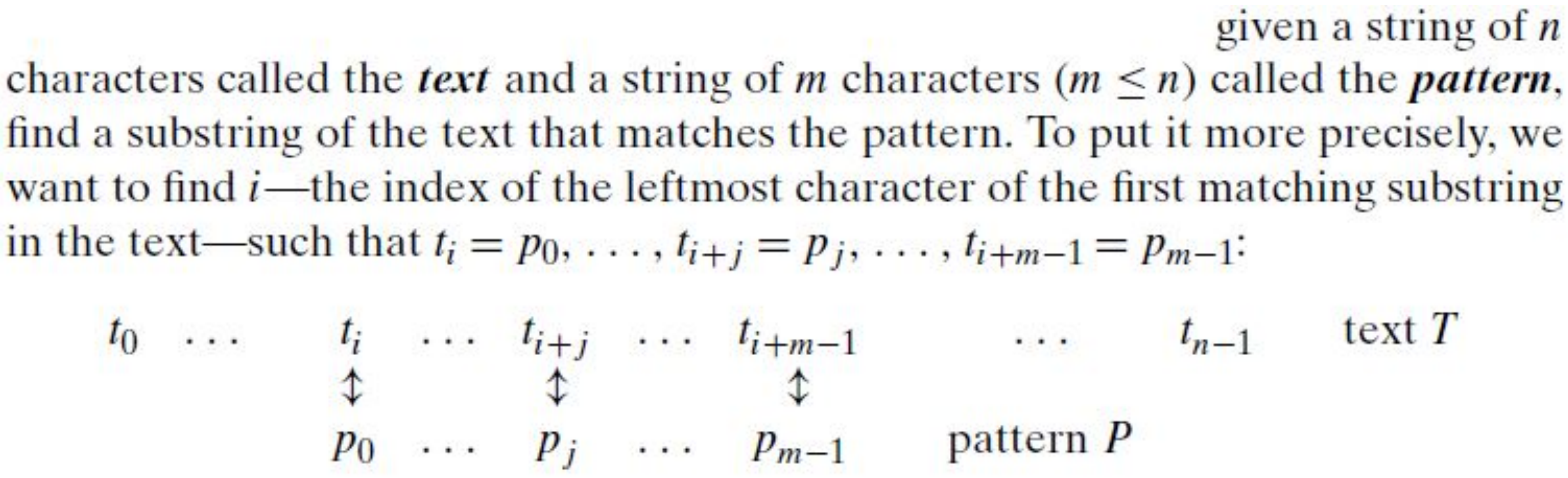
$$\begin{array}{ccccccccccccc}
45 & \overset{?}{\leftrightarrow} & 68 & \overset{?}{\leftrightarrow} & 89 & \overset{?}{\leftrightarrow} & 29 & & 34 & & 17 & | & 90 \\
45 & & 68 & & 29 & & 89 & \overset{?}{\leftrightarrow} & 34 & & 17 & | & 90 \\
45 & & 68 & & 29 & & 34 & & 89 & \overset{?}{\leftrightarrow} & 17 & | & 90 \\
45 & & 68 & & 29 & & 34 & & 17 & | & 89 & & 90
\end{array}$$



$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

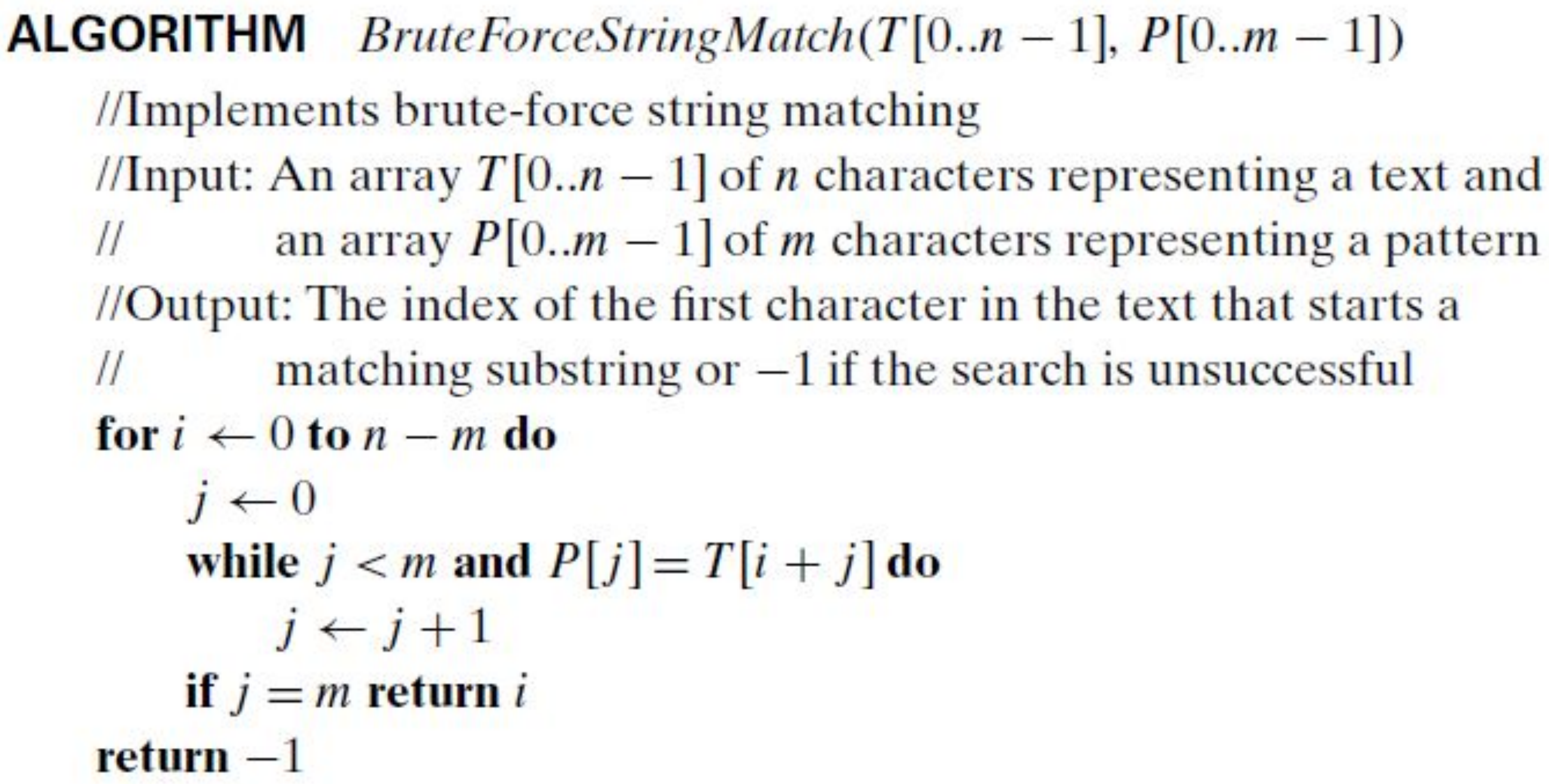
String Matching



given a string of n characters called the ***text*** and a string of m characters ($m \le n$) called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{cccccccccc}
t_0 & \dots & t_i & \dots & t_{i+j} & \dots & t_{i+m-1} & \dots & t_{n-1} & \text{text } T \\
 & & \updownarrow & & \updownarrow & & \updownarrow & & & \\
 & & p_0 & \dots & p_j & \dots & p_{m-1} & \text{pattern } P & &
\end{array}$$

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```



ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of $n$ characters representing a text and
//       an array $P[0..m-1]$ of $m$ characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or $-1$ if the search is unsuccessful
for $i \leftarrow 0$ to $n-m$ do
    $j \leftarrow 0$
    while $j < m$ and $P[j] = T[i+j]$ do
        $j \leftarrow j+1$
    if $j = m$ return $i$
return $-1$
```

The worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries.

Assignment Problem

- There are *n* people who need to be assigned to execute *n* jobs, one person per job.

- (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the *i*th person is assigned to the *j*th job is a known quantity $C[i, j\ ]$ for each pair $i, j = 1, 2, \ldots, n$.

- The problem is to find an assignment with the minimum total cost.

Hence problem is stated as

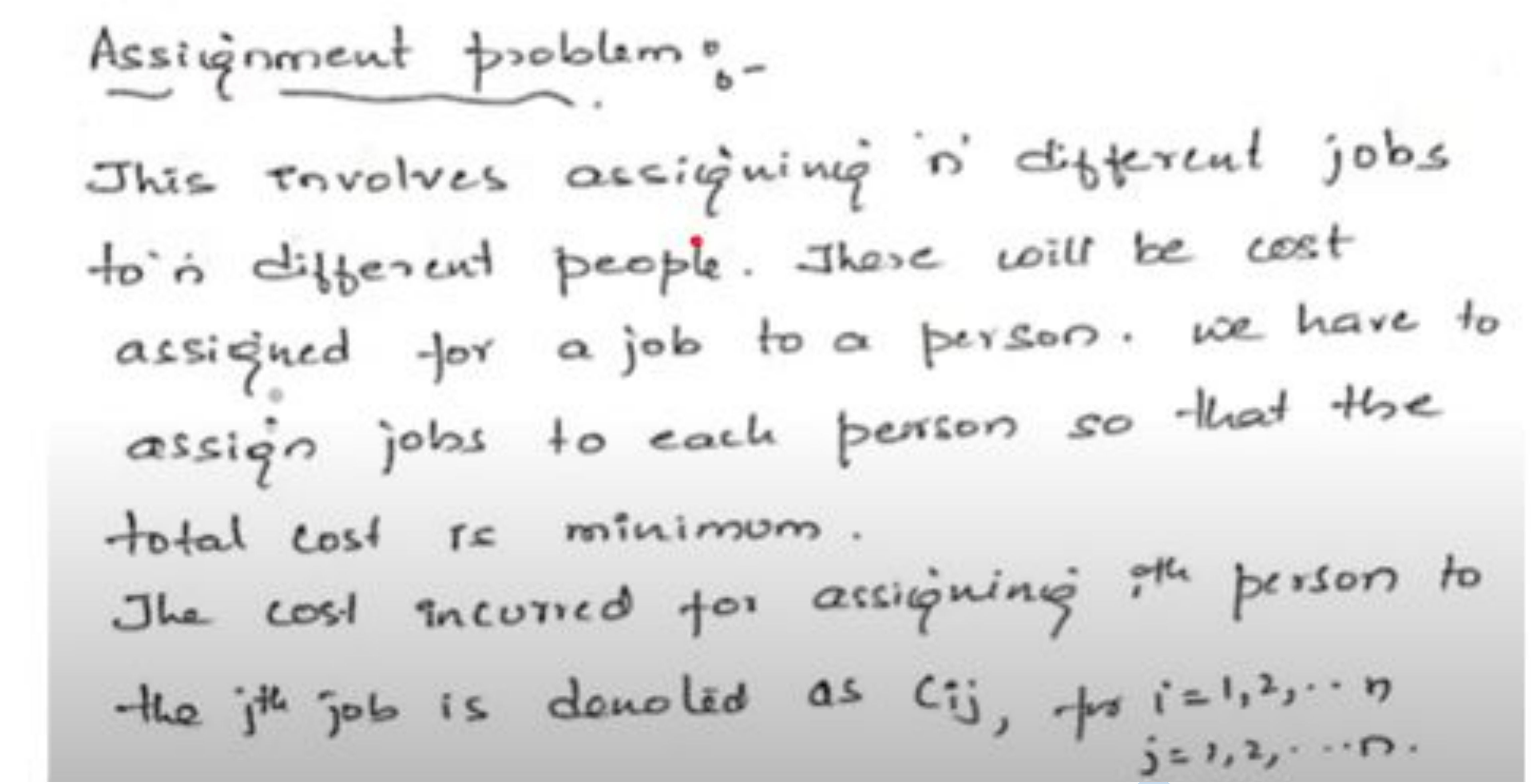
$$\text{minimum } \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \cdot x_{ij}$$

Subjected to constraints

$$x_{ij} = \begin{cases} 1, & \text{if } i^{th} \text{ person is assigned } j^{th} \text{ job} \\ 0, & \text{otherwise.} \end{cases}$$

and $\sum_{j=1}^{n} x_{ij} = 1$, ie exactly one job is assigned to i^{th} person.

- The exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2, . . . , *n*, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum

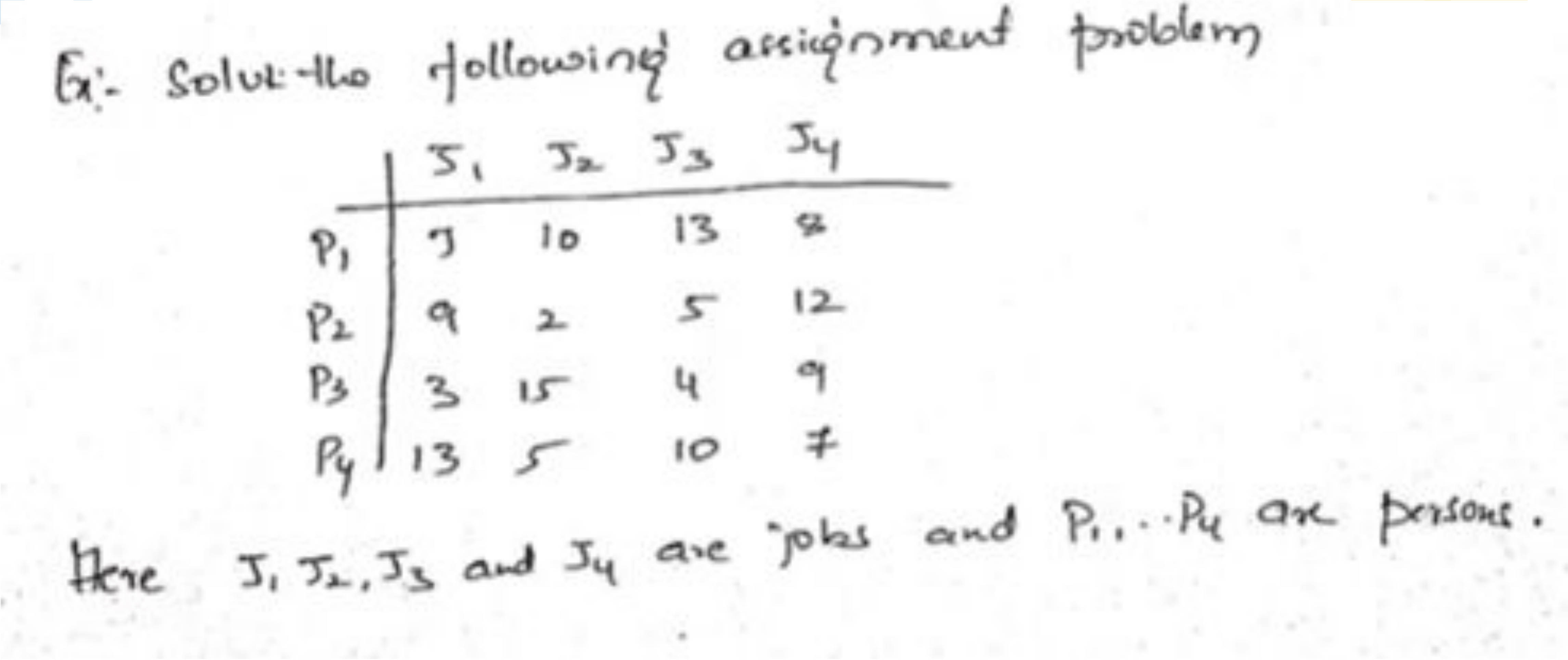


Assignment problem :-

This involves assigning 'n' different jobs to n different people. There will be cost assigned for a job to a person. we have to assign jobs to each person so that the total cost is minimum.

The cost incurred for assigning i^{th} person to the j^{th} job is donoted as C_{ij}, for $i = 1, 2, \cdots n$
$j = 1, 2, \cdots n$.

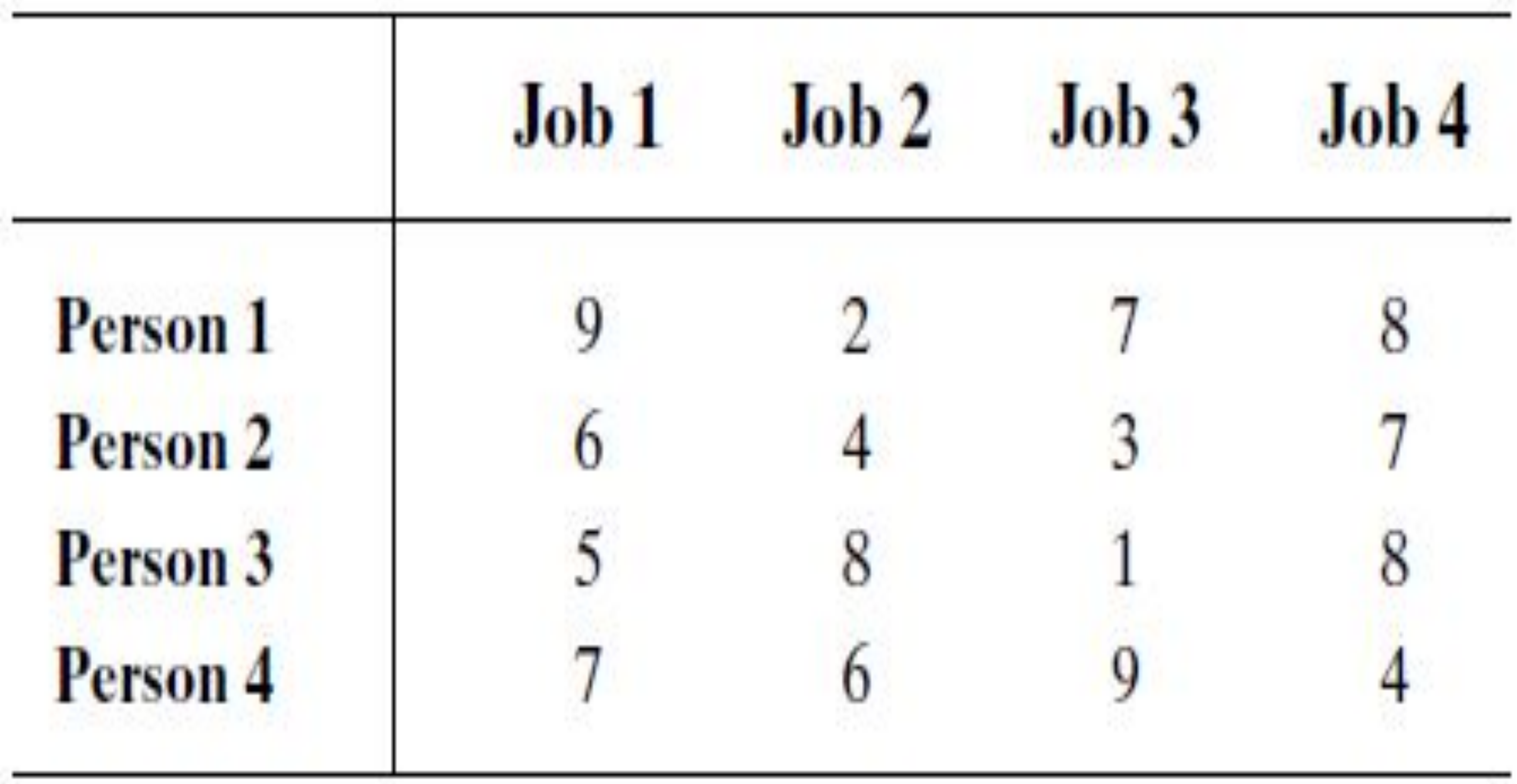
Hence in general, for any assignment problem with 'n' jobs, we have n! number of permutation of n-tuples. ∴ time Complexity is O(n!)



Ex:- Solve the following assignment problem

	J_1	J_2	J_3	J_4
P_1	7	10	13	8
P_2	9	2	5	12
P_3	3	15	4	9
P_4	13	5	10	7

Here J_1, J_2, J_3 and J_4 are jobs and $P_1 \cdots P_4$ are persons.

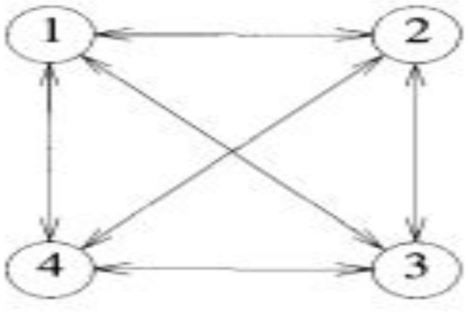


	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

1. TSP:

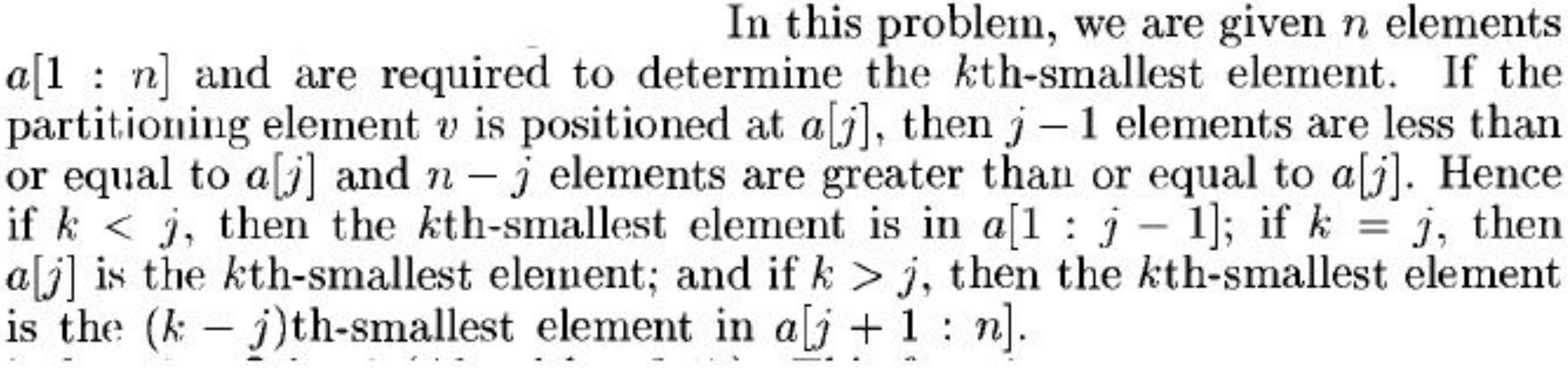


$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

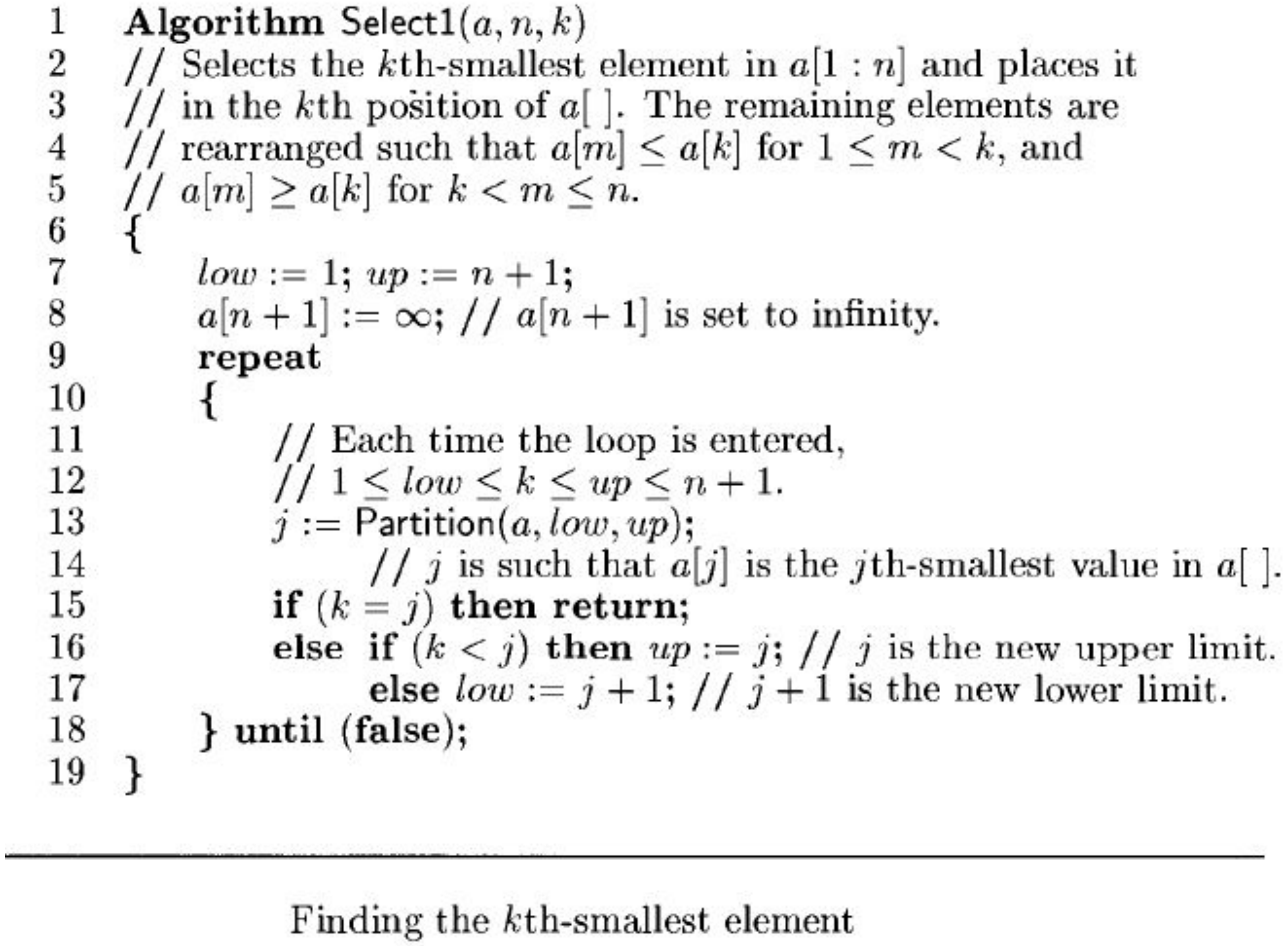
2.Consider the knapsack instance n=3,(P1,P2,P3)=(1,2,5),(W1,W2,W3)=(2,3,4) and capacity of the knapsack is m=6.

3.Consider the knapsack instance n=4,(P1,P2,P3,P4)=(10,10,12,18),(W1,W2,W3,W4)=(2,4,6,9) and capacity of the knapsack is m=15.

SELECTION



In this problem, we are given n elements $a[1:n]$ and are required to determine the kth-smallest element. If the partitioning element v is positioned at $a[j]$, then $j-1$ elements are less than or equal to $a[j]$ and $n-j$ elements are greater than or equal to $a[j]$. Hence if $k < j$, then the kth-smallest element is in $a[1:j-1]$; if $k = j$, then $a[j]$ is the kth-smallest element; and if $k > j$, then the kth-smallest element is the $(k-j)$th-smallest element in $a[j+1:n]$.



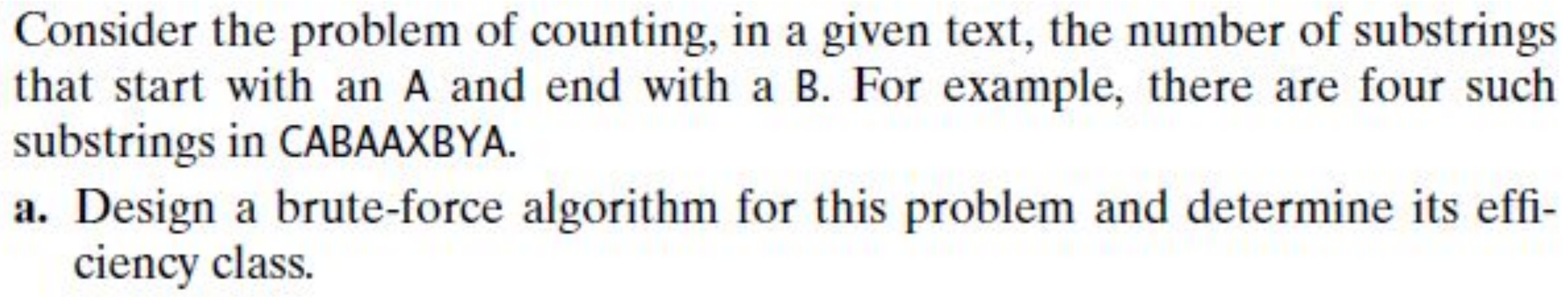
```
Algorithm Select1($a, n, k$)
// Selects the $k$th-smallest element in $a[1 : n]$ and places it
// in the $k$th position of $a[\,]$. The remaining elements are
// rearranged such that $a[m] \leq a[k]$ for $1 \leq m < k$, and
// $a[m] \geq a[k]$ for $k < m \leq n$.
{
    $low := 1$; $up := n + 1$;
    $a[n + 1] := \infty$; // $a[n + 1]$ is set to infinity.
    repeat
    {
        // Each time the loop is entered,
        // $1 \leq low \leq k \leq up \leq n + 1$.
        $j :=$ Partition($a, low, up$);
             // $j$ is such that $a[j]$ is the $j$th-smallest value in $a[\,]$.
        if ($k = j$) then return;
        else if ($k < j$) then $up := j$; // $j$ is the new upper limit.
             else $low := j + 1$; // $j + 1$ is the new lower limit.
    } until (false);
}
```

Finding the kth-smallest element

```
**Algorithm** Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    **repeat**
    {
        **repeat**
            $i := i + 1$;
        **until** $(a[i] \geq v)$;

        **repeat**
            $j := j - 1$;
        **until** $(a[j] \leq v)$;

        **if** $(i < j)$ **then** Interchange($a, i, j$);

    } **until** $(i \geq j)$;

    $a[m] := a[j]$; $a[j] := v$; **return** $j$;
}
```

- 65 70 75 80 85 60 55 50

- K=5



Consider the problem of counting, in a given text, the number of substrings that start with an A and end with a B. For example, there are four such substrings in CABAAXBYA.

a. Design a brute-force algorithm for this problem and determine its efficiency class.