MVJ22CS41 – ANALYSIS AND DESIGN OF ALGORITHMS
Module 2

Semester – IV Section - A
Academic Year : 2024-2025(EVEN)
By
NIKITHA G S

MVJ COLLEGE OF ENGINEERING
Since 1982
Engineering A Better Tomorrow

An Autonomous Institute

Brute force : Selection sort

Selection sort is an in-place comparison sorting algorithm that uses brute force to sort an array.

It's called a "brute force" algorithm because it uses the simplest and most ineffective way of calculating the solution. However, it does makes up for it with its straightforward implementation.

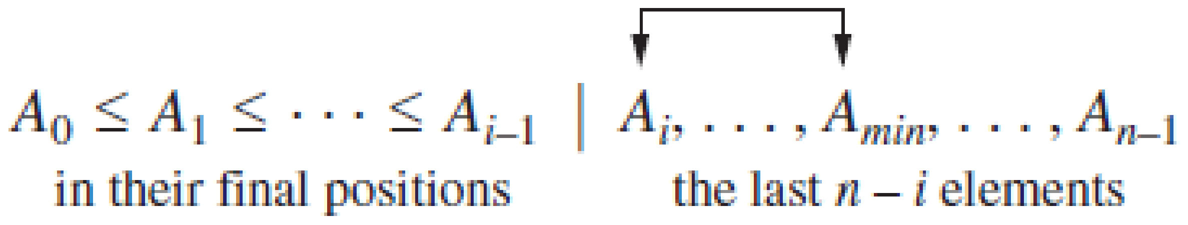
The algorithm divides the array into two subarrays:
A sorted subarray
An unsorted subarray

The sorted subarray is empty in the beginning. In every iteration, the smallest element of the unsorted array will be appended to the end of the sorted array by swapping. This way, the sorted array will eventually contain all the elements of the original array.

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

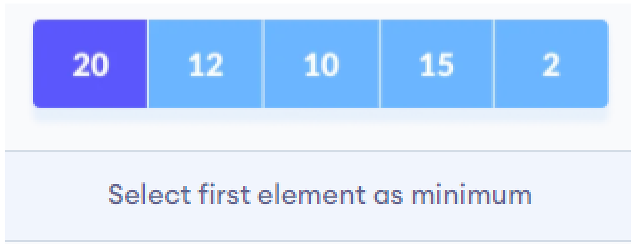
Selection sort, by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.

Then we scan the list, starting with the second element, to find the smallest among the last $n - 1$ elements and exchange it with the second element, putting the second smallest element in its final position Generally, on the ith pass through the list, which we number from 0 to $n - 2$, the algorithm searches for the smallest item among the last $n - i$ elements and swaps it with A_i. After n − 1 passes, the list is sorted.


$A_0 \le A_1 \le \cdots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$
in their final positions
the last $n - i$ elements

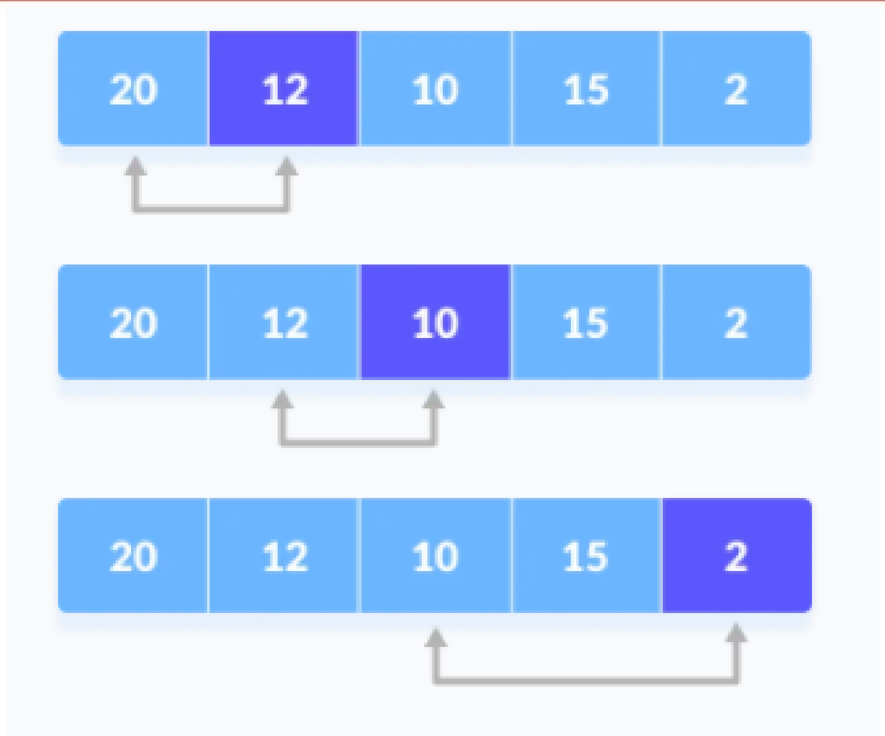
Working of Selection Sort

1. Set the first element as minimum



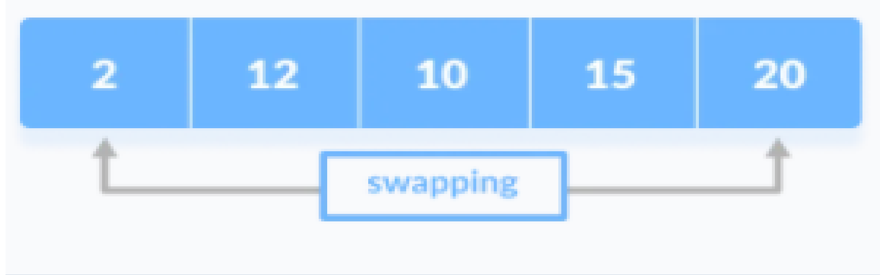
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

3. Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



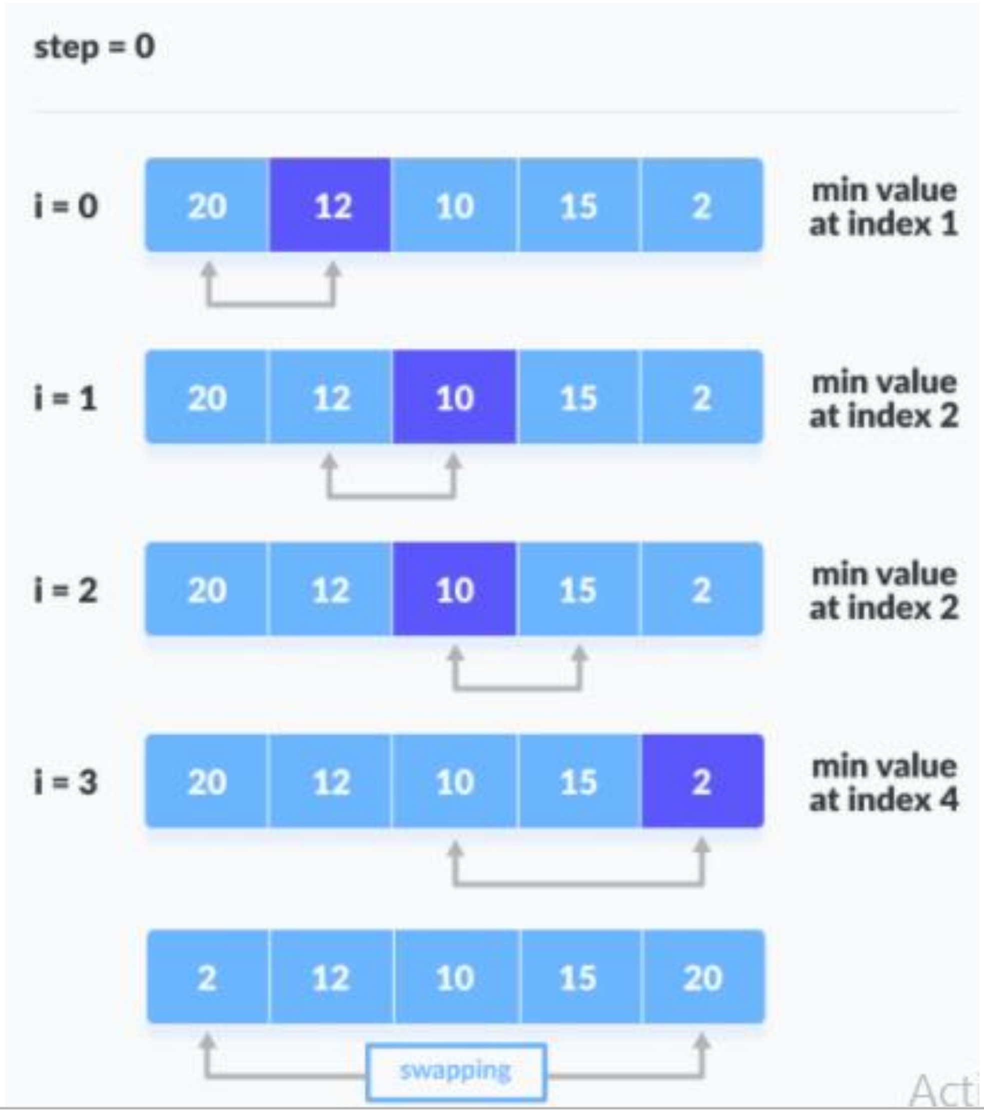
Compare minimum with the remaining elements

4. After each iteration, minimum is placed in the front of the unsorted list

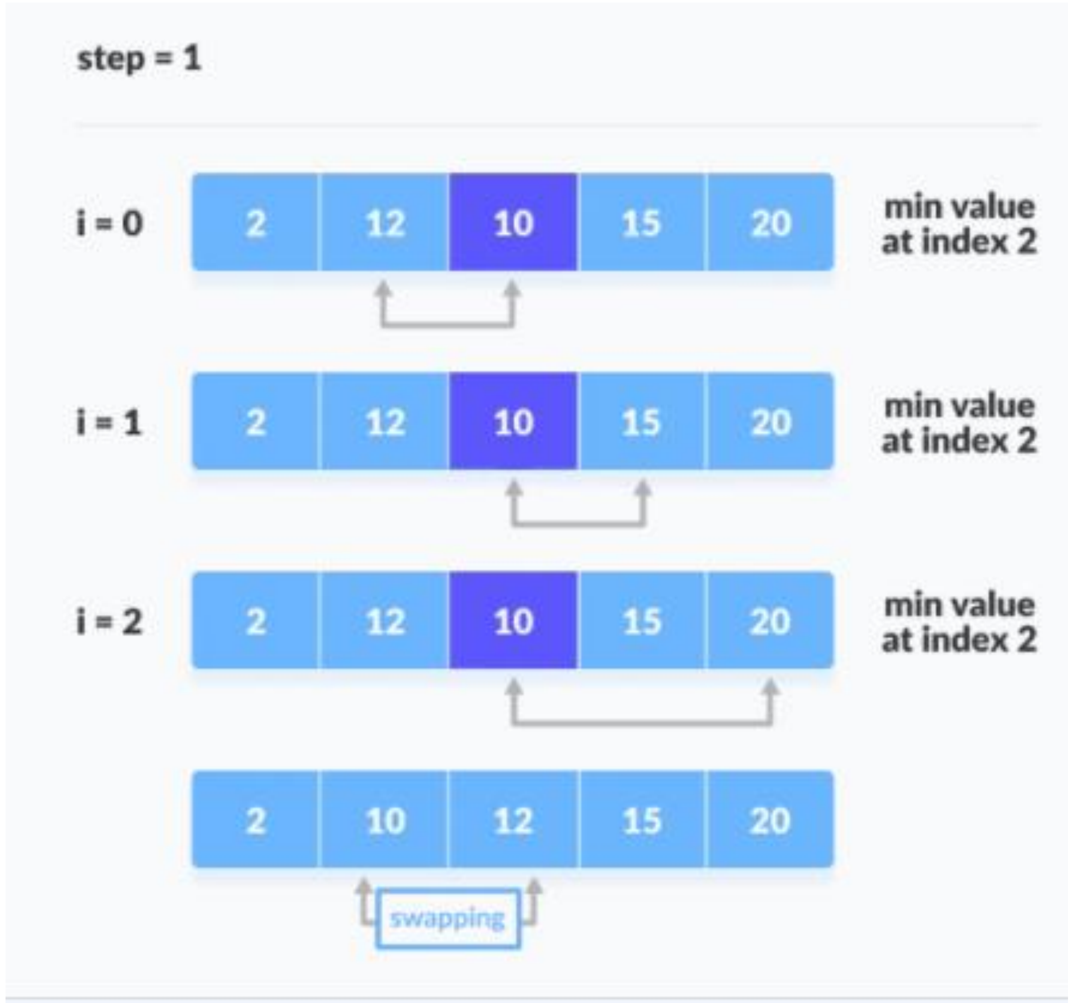


Swap the first with minimum

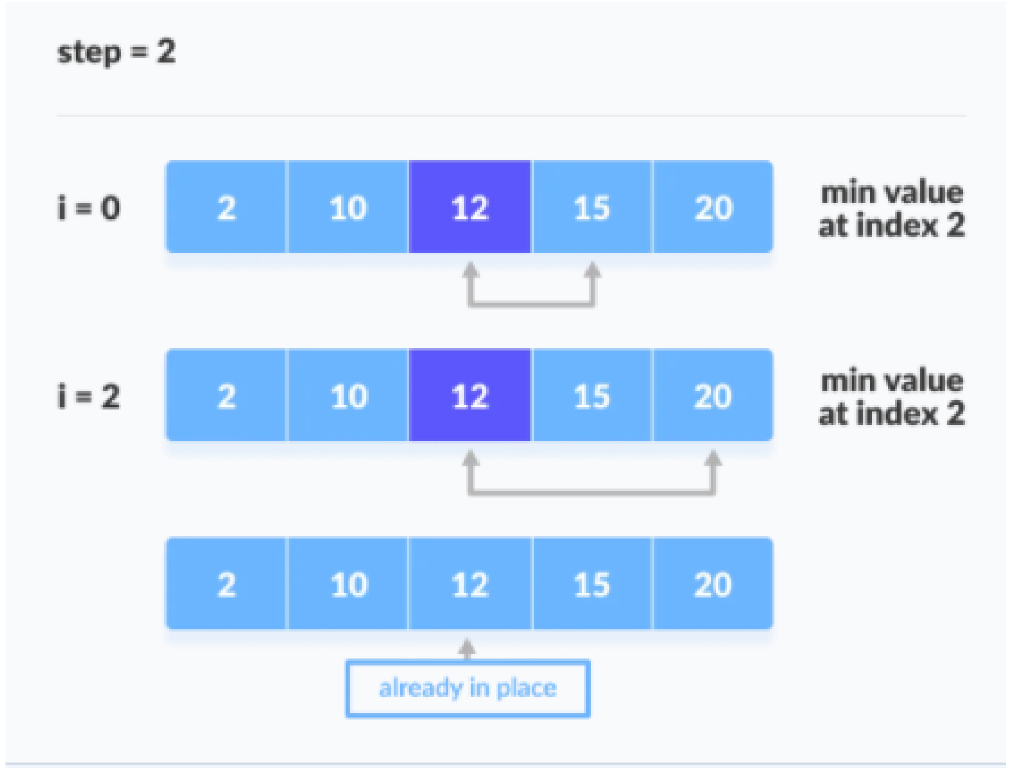
5. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



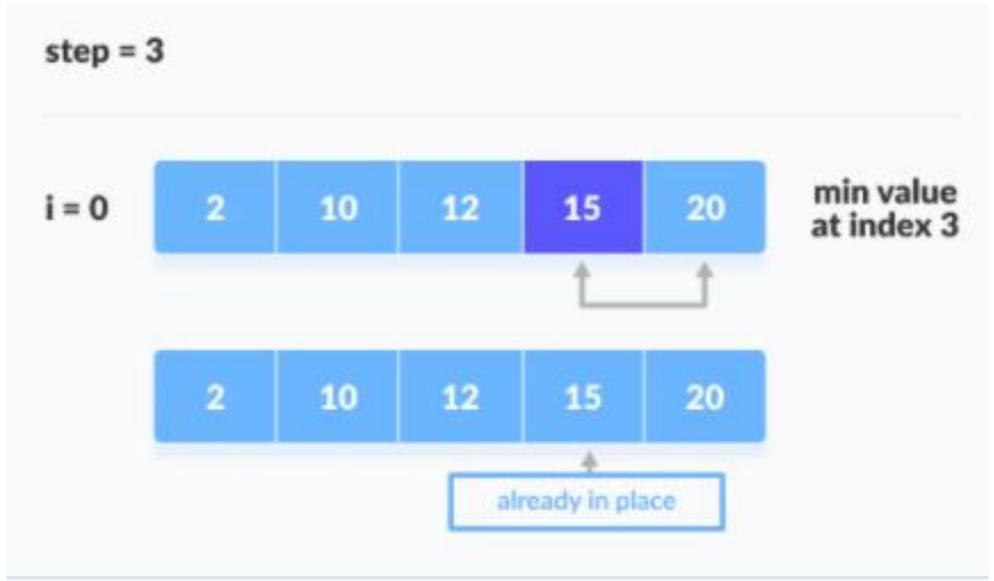
For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



The second iteration

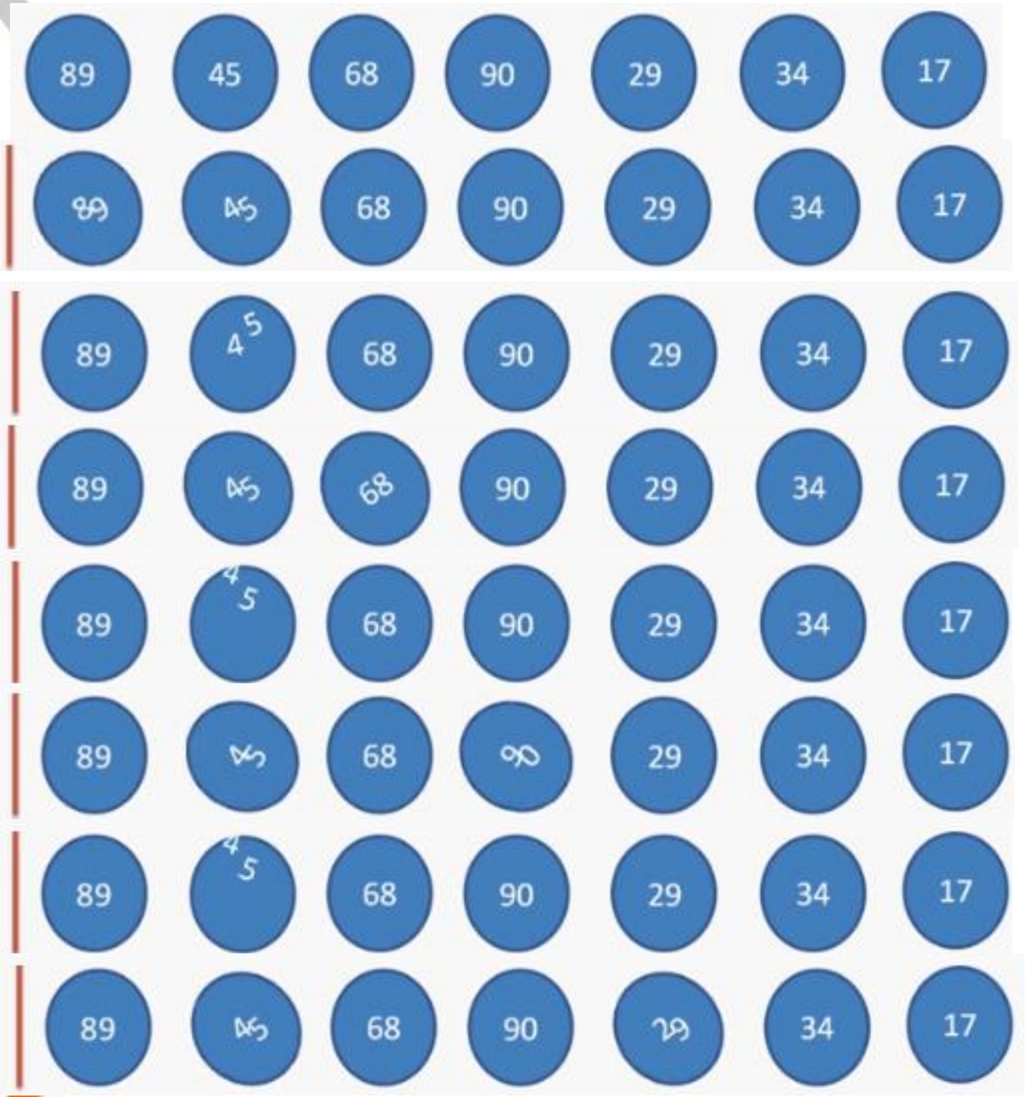


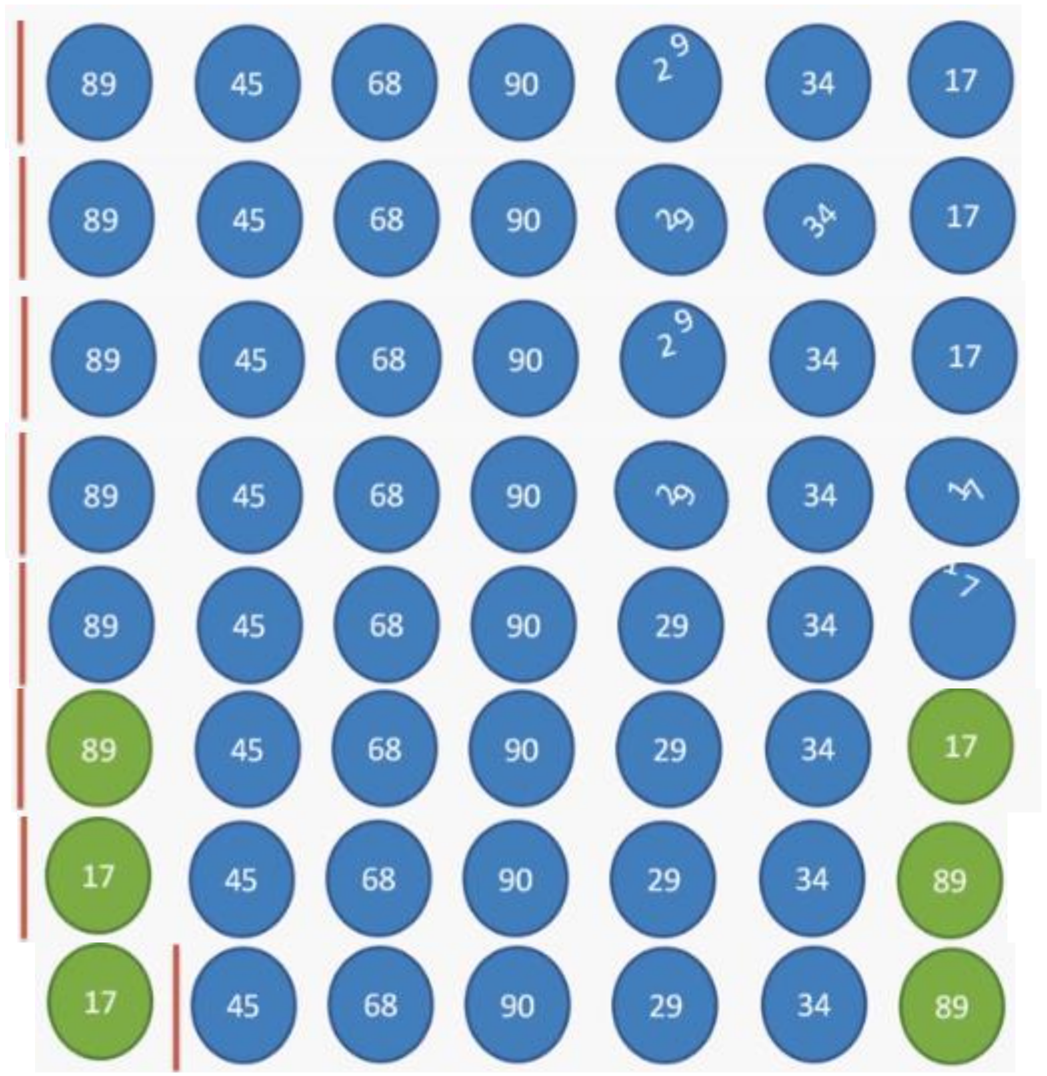
The third iteration

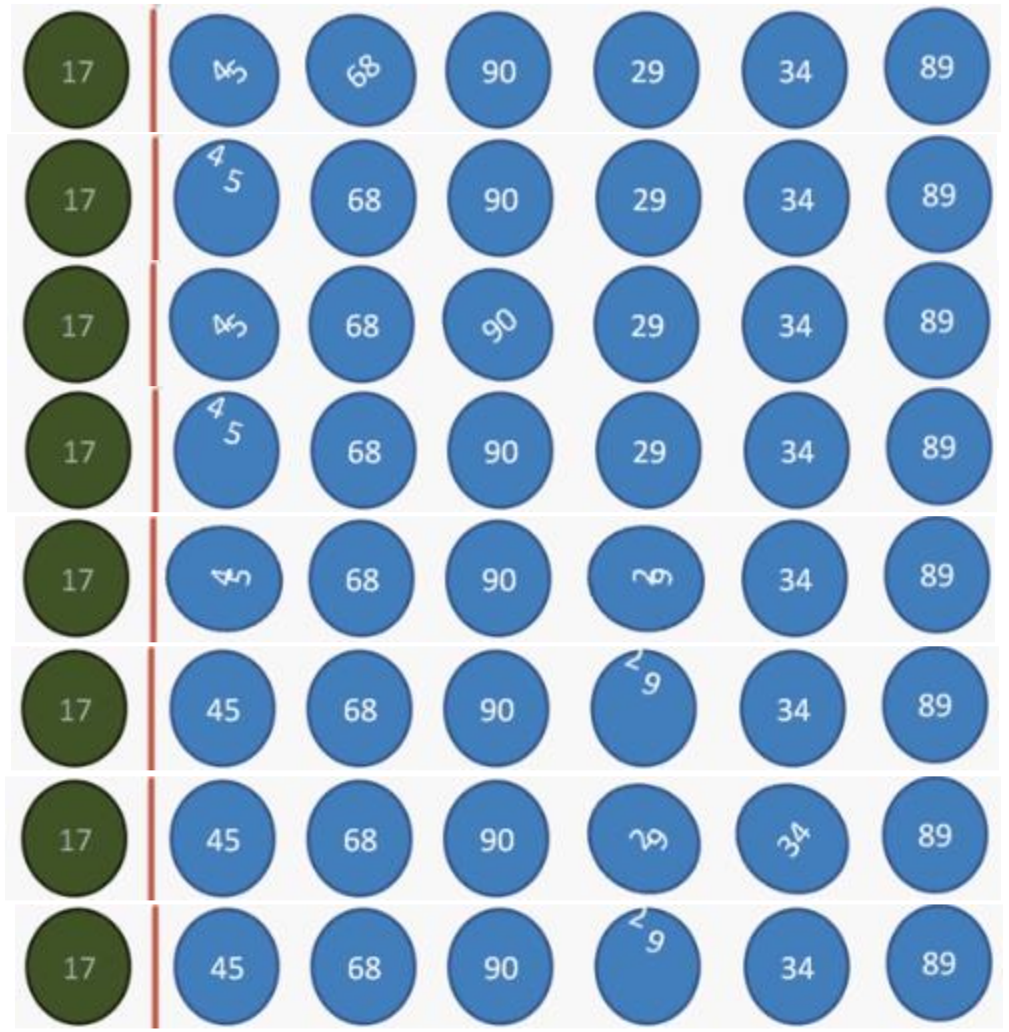


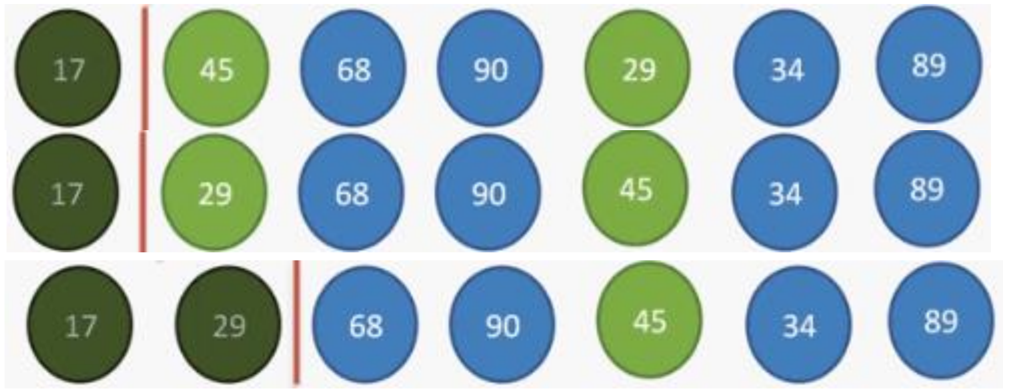
The fourth iteration

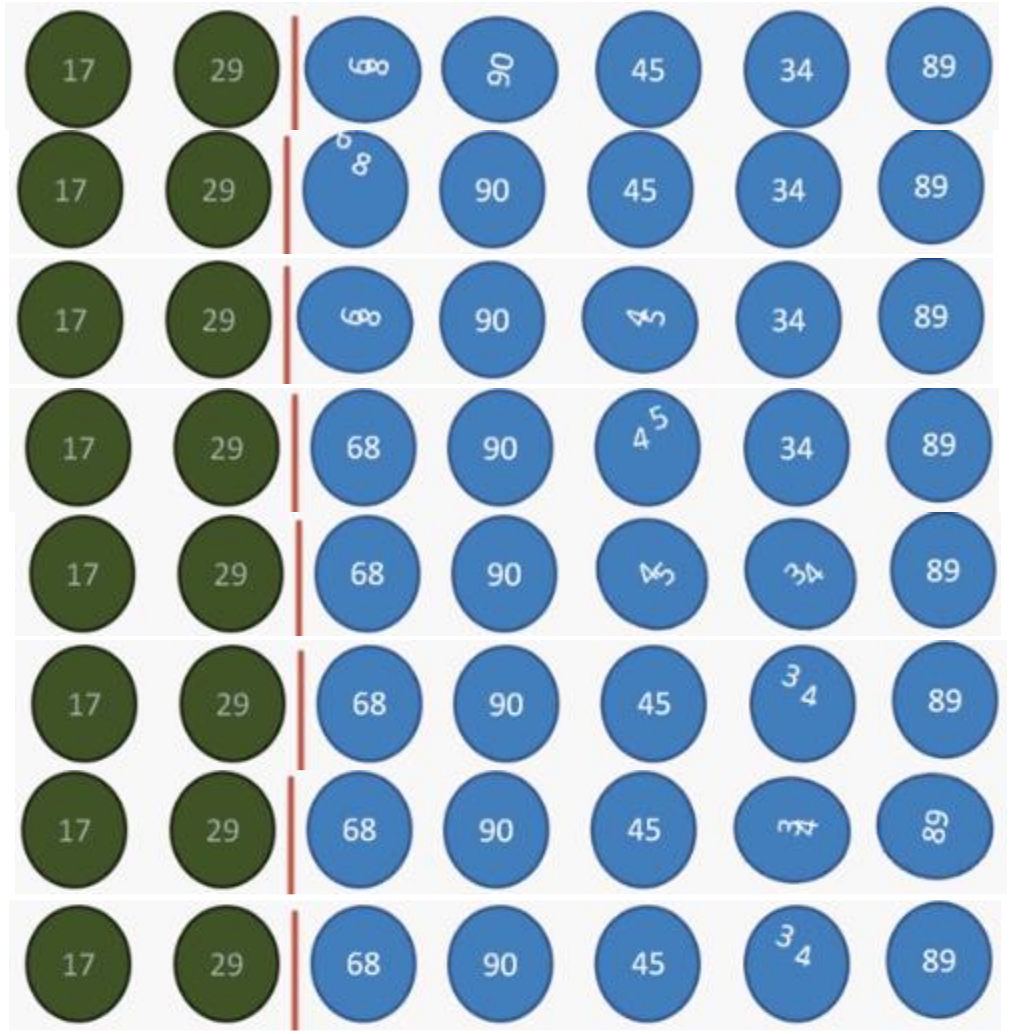
Example : 2

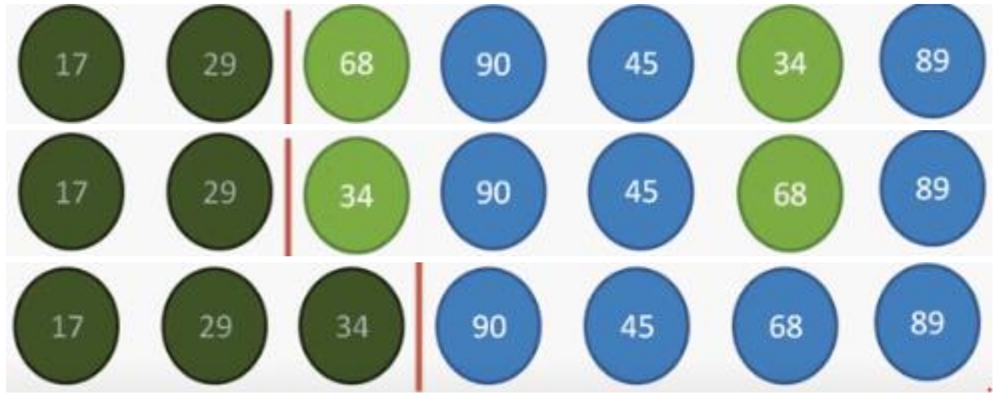

89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17


89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
89 45 68 90 29 34 17
17 45 68 90 29 34 89
17 45 68 90 29 34 89


17
45
68
90
29
34
89


17
45
68
90
29
34
89
17
29
68
90
45
34
89
17
29
68
90
45
34
89


17
29
68
90
45
34
89


17
29
68
90
45
34
89
17
29
34
90
45
68
89

```
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 $i \leftarrow 0$ to $n-2$ do
        $min \leftarrow i$
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[j] < A[min]$  $min \leftarrow j$
        swap $A[i]$ and $A[min]$
```

How Long does it takes ?

How many times do we swap ?
n-1 times

How many times do we find min ?
n-1 times

What is the basic operation ?
if A[j] < A[min]

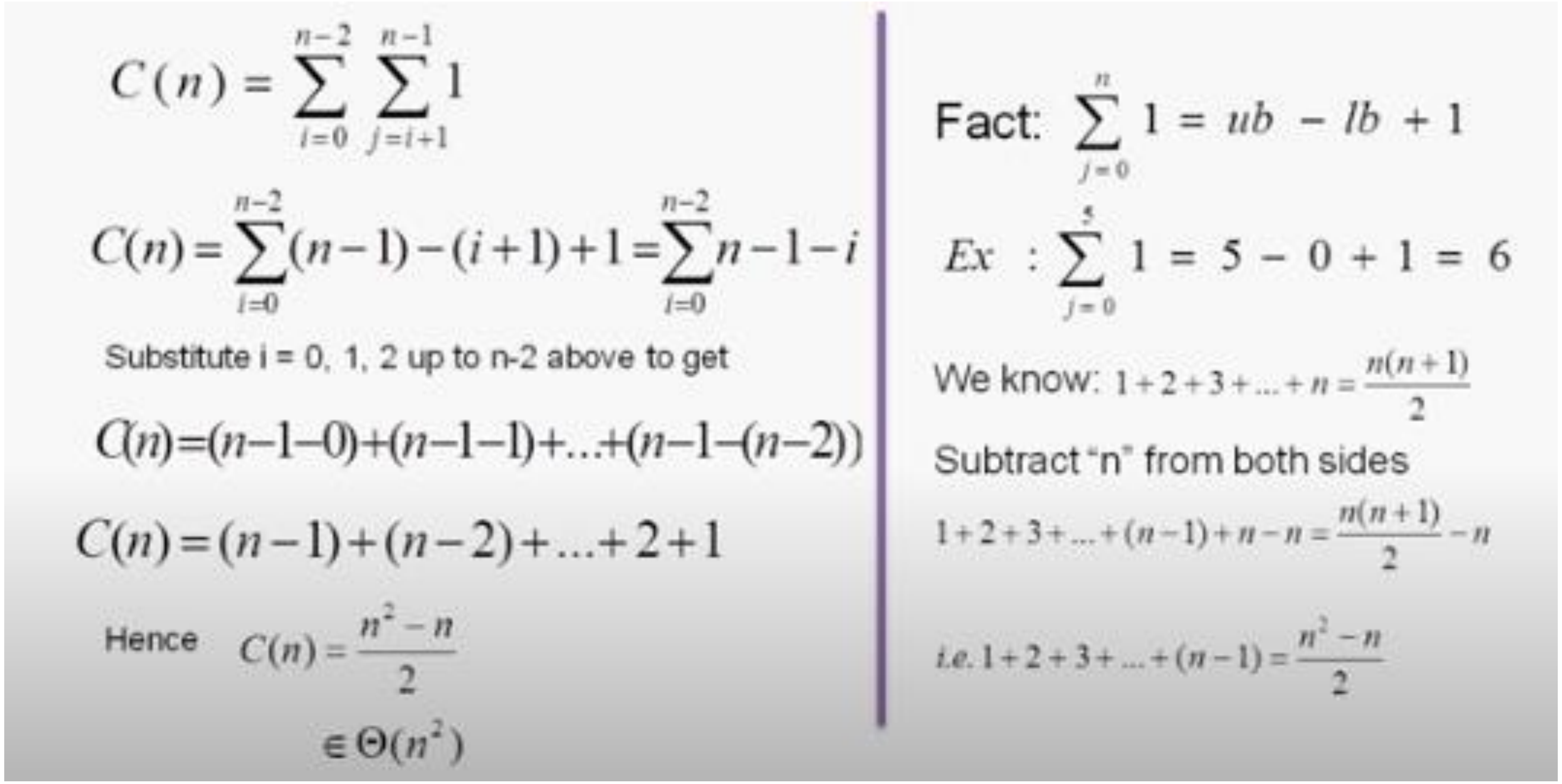
How long does it take to find min?
n-1 + n-2 + n-3 +………..1

Time Complexity:

Time Complexity is the number of times the basic operation is executed

To find the minimum element from the array of N elements, N−1 comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to N−1 and then N−2 comparisons are required to find the minimum in the unsorted array.

(n-1) + (n-2) + (n-3) +……..……..1 => (n(n-1))/2 comparisons.



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

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

Substitute i = 0, 1, 2 up to n-2 above to get

$$C(n) = (n-1-0) + (n-1-1) + \ldots + (n-1-(n-2))$$

$$C(n) = (n-1) + (n-2) + \ldots + 2 + 1$$

Hence $C(n) = \frac{n^2 - n}{2}$

$\in \Theta(n^2)$

Fact: $\sum_{j=0}^{n} 1 = ub - lb + 1$

$Ex : \sum_{j=0}^{5} 1 = 5 - 0 + 1 = 6$

We know: $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

Subtract "n" from both sides

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

$$i.e.\ 1 + 2 + 3 + \ldots + (n-1) = \frac{n^2 - n}{2}$$

An Autonomous Institute

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Because it always takes the same number of time irrespective of order of the given input elements.

Brute Force Approach - Bubble sort

Bubble sort is one of the simple sorting algorithms and also popularly known as a **Brute Force Approach**.

The logic of the algorithm is very simple as it works by repeatedly iterating through a list of elements, comparing two elements at a time and swapping them if necessary until all the elements are swapped to an order.

For e.g. if we have a list of 10 elements, bubble sort starts by comparing the first two elements in the list. If the second element is smaller than the first element then it exchanges them. Then it compares the current second element with the third element in the list.

This continues until the second last and the last element is compared which completes one iteration through the list. By the time it completes the first iteration the largest element in the list comes to the rightmost position.

The algorithm gets its name as we start from lowest point and “bubble up” the higher elements to the highest point in the list.

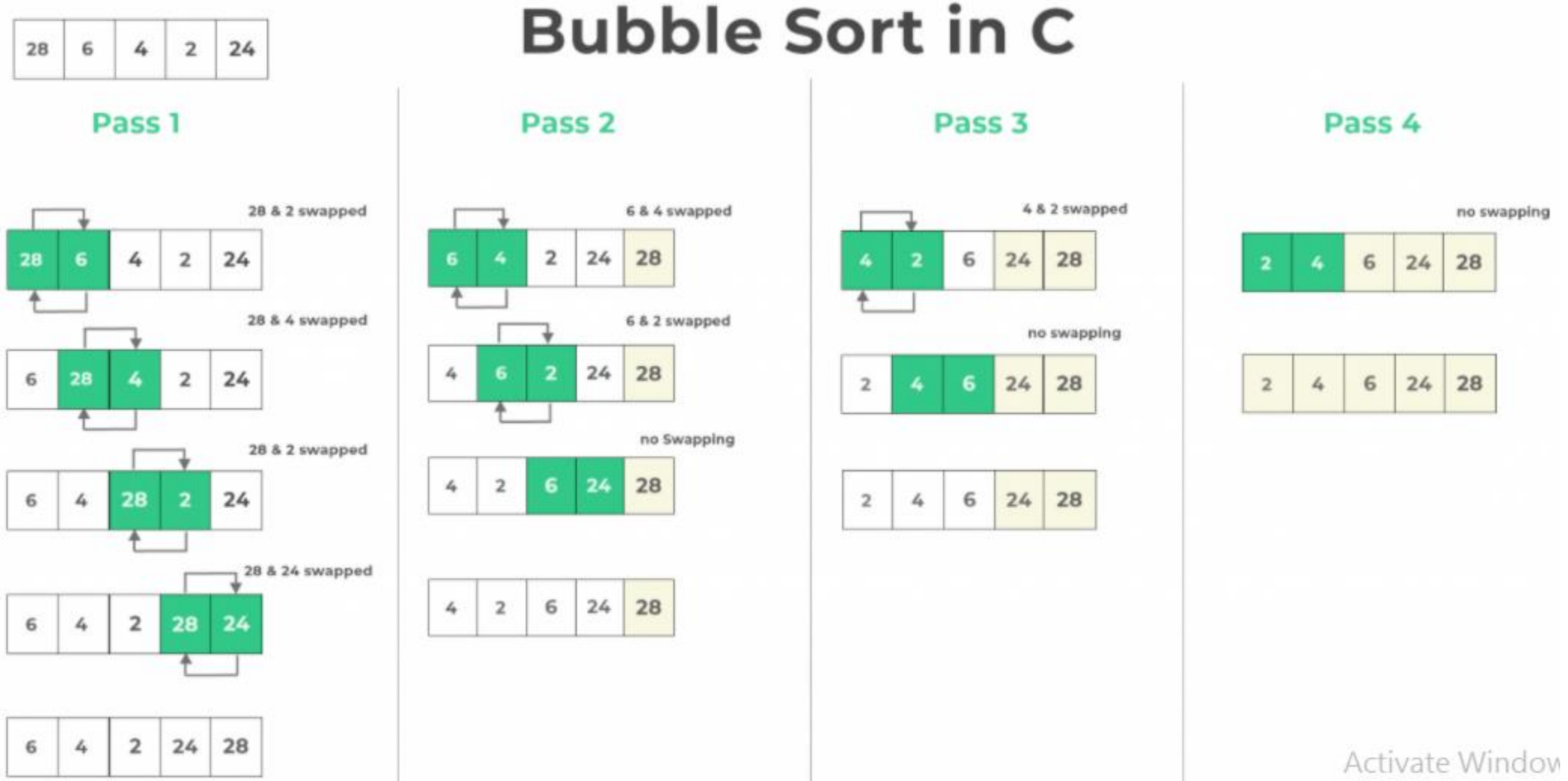
We can also follow other approach where we start from highest point and “bubble down” lowest elements in the list.

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.

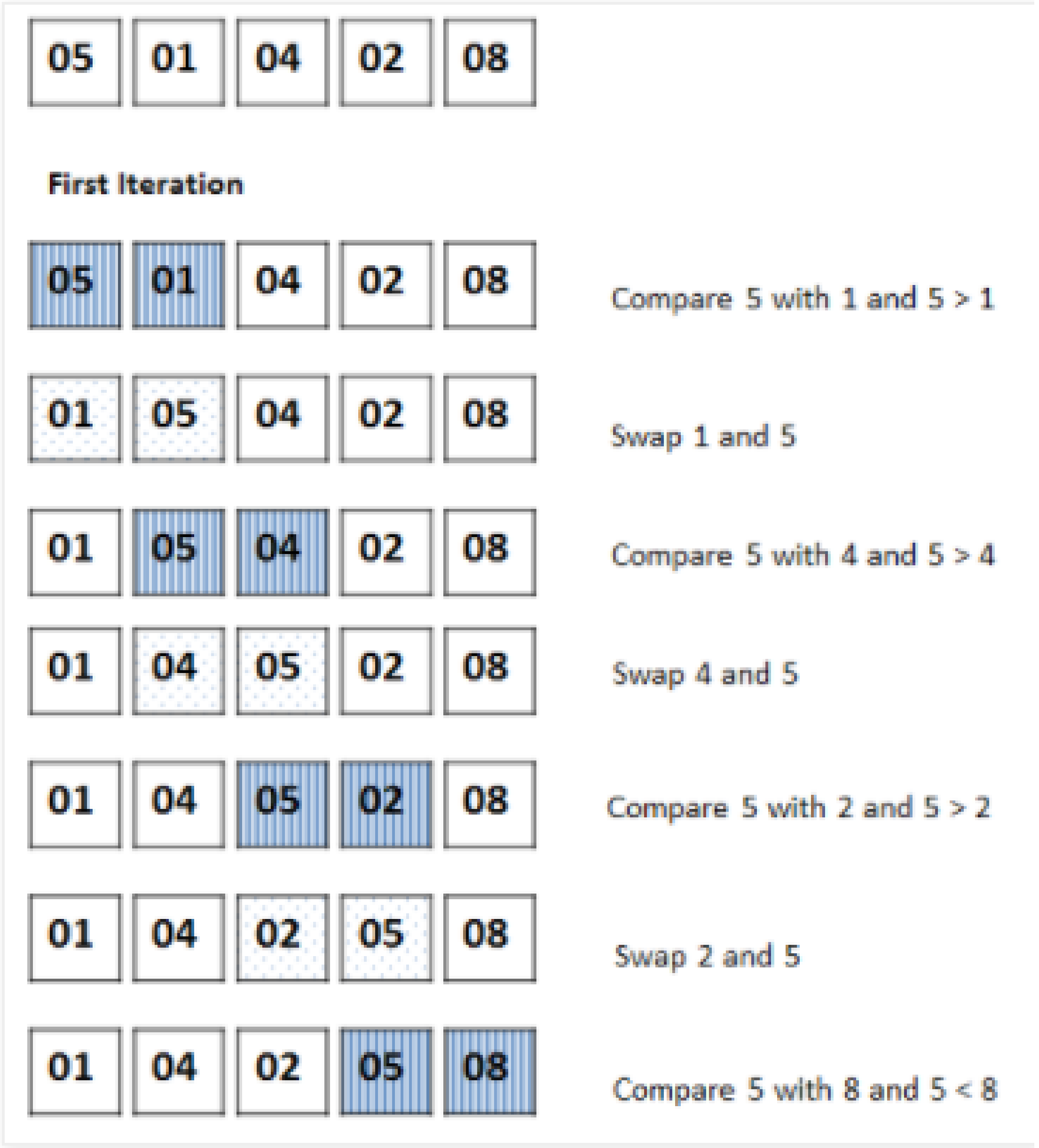
By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after n − 1 passes the list is sorted. Pass i ($0 \le i \le n - 2$) of bubble sort can be represented by the following diagram:

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

Bubble Sort in C



Graphical Example

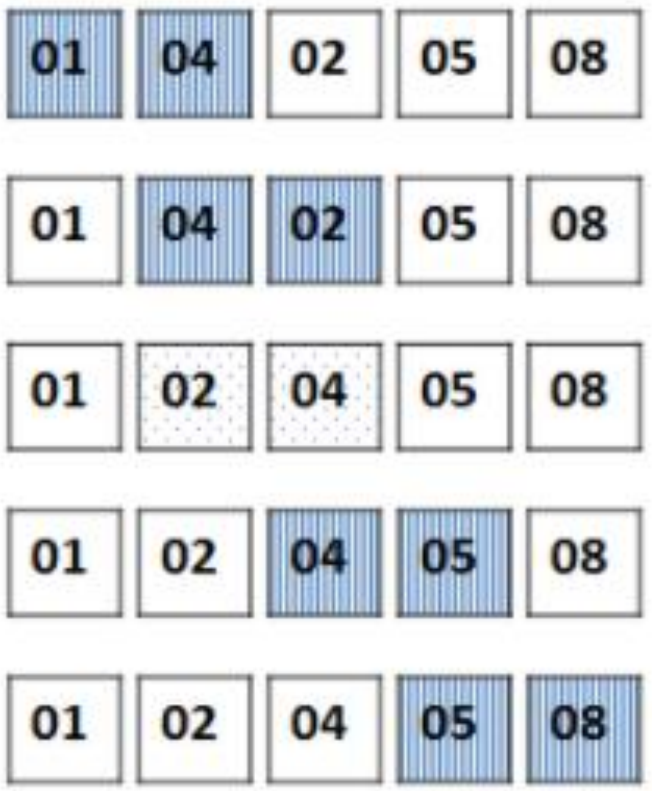


05	01	04	02	08	

First Iteration

05	01	04	02	08	Compare 5 with 1 and 5 > 1
01	05	04	02	08	Swap 1 and 5
01	05	04	02	08	Compare 5 with 4 and 5 > 4
01	04	05	02	08	Swap 4 and 5
01	04	05	02	08	Compare 5 with 2 and 5 > 2
01	04	02	05	08	Swap 2 and 5
01	04	02	05	08	Compare 5 with 8 and 5 < 8

Second Iteration



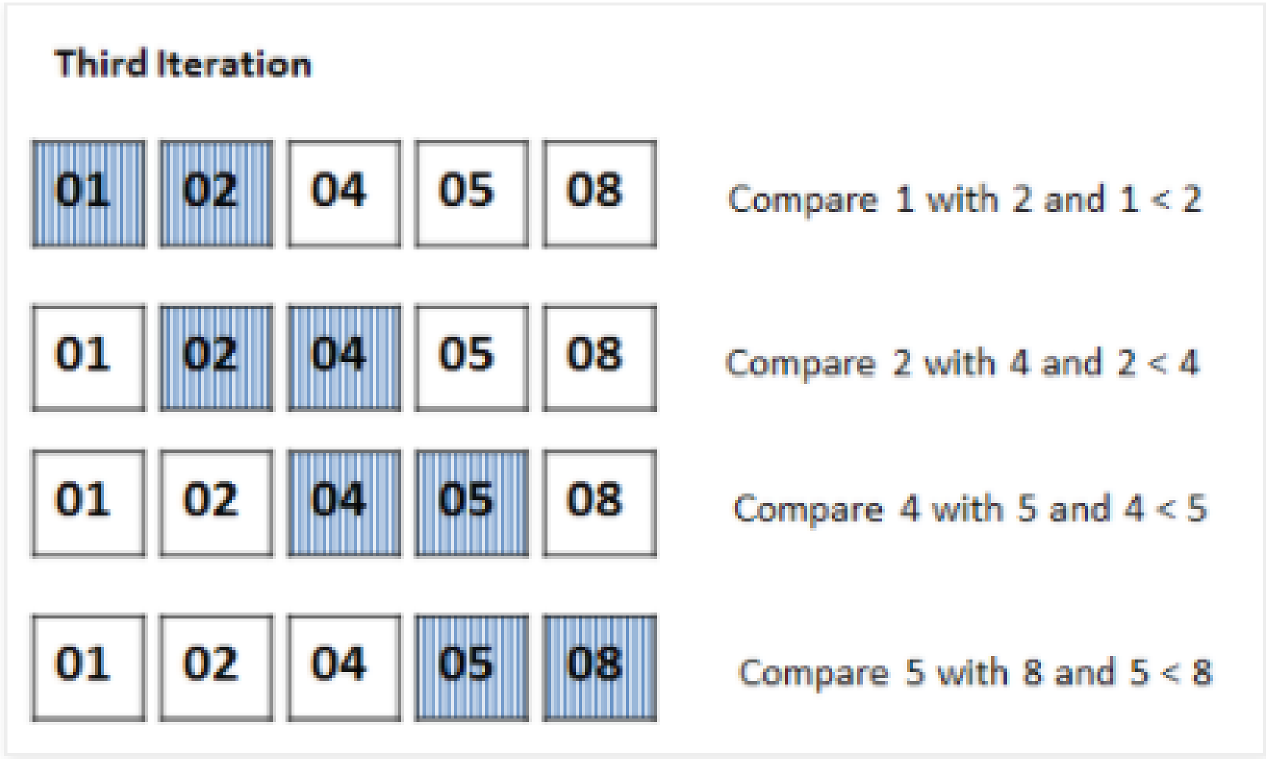
Compare 1 with 4 and 1 < 4

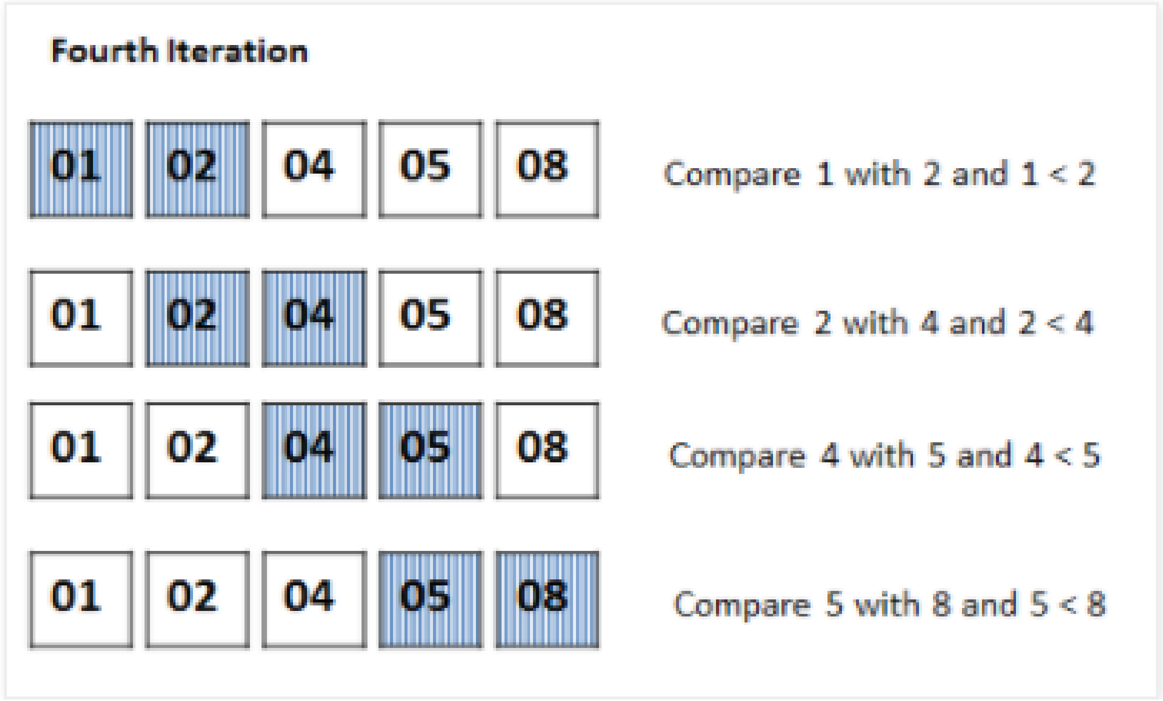
Compare 4 with 2 and 4 > 2

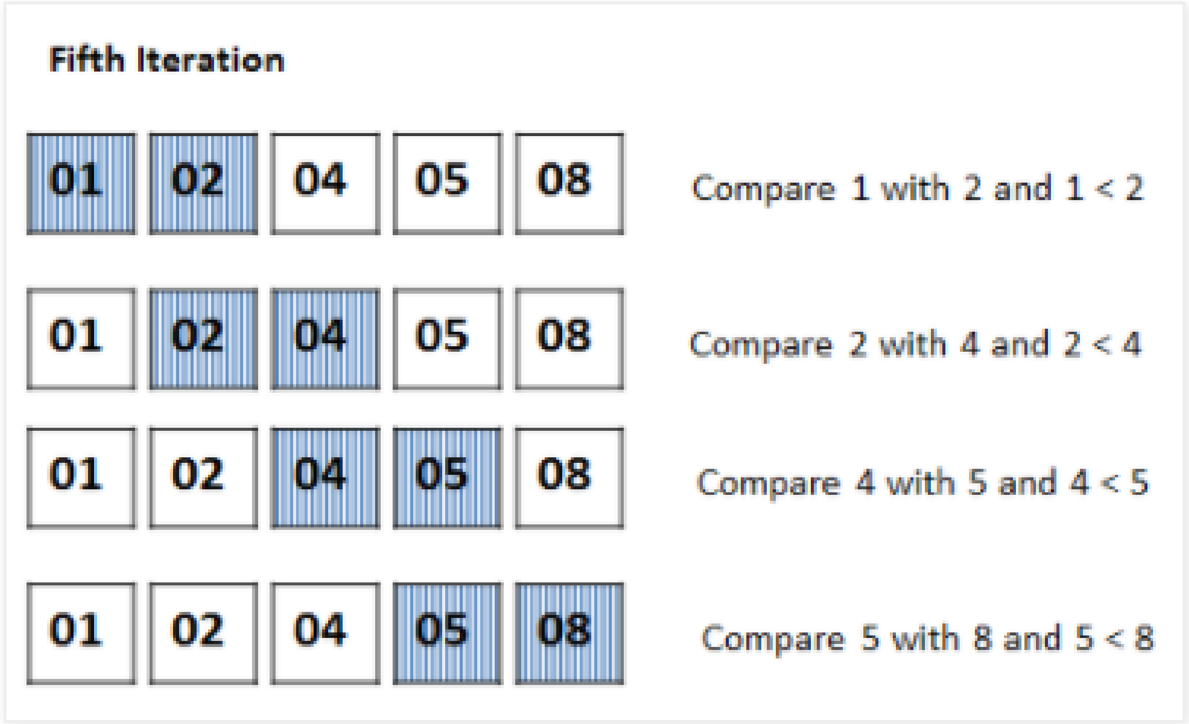
Swap 2 and 4

Compare 4 with 5 and 4 < 5

Compare 5 with 8 and 5 < 8


Third Iteration
01 02 04 05 08
Compare 1 with 2 and 1 < 2
01 02 04 05 08
Compare 2 with 4 and 2 < 4
01 02 04 05 08
Compare 4 with 5 and 4 < 5
01 02 04 05 08
Compare 5 with 8 and 5 < 8


Fourth Iteration
01 02 04 05 08
Compare 1 with 2 and 1 < 2
01 02 04 05 08
Compare 2 with 4 and 2 < 4
01 02 04 05 08
Compare 4 with 5 and 4 < 5
01 02 04 05 08
Compare 5 with 8 and 5 < 8

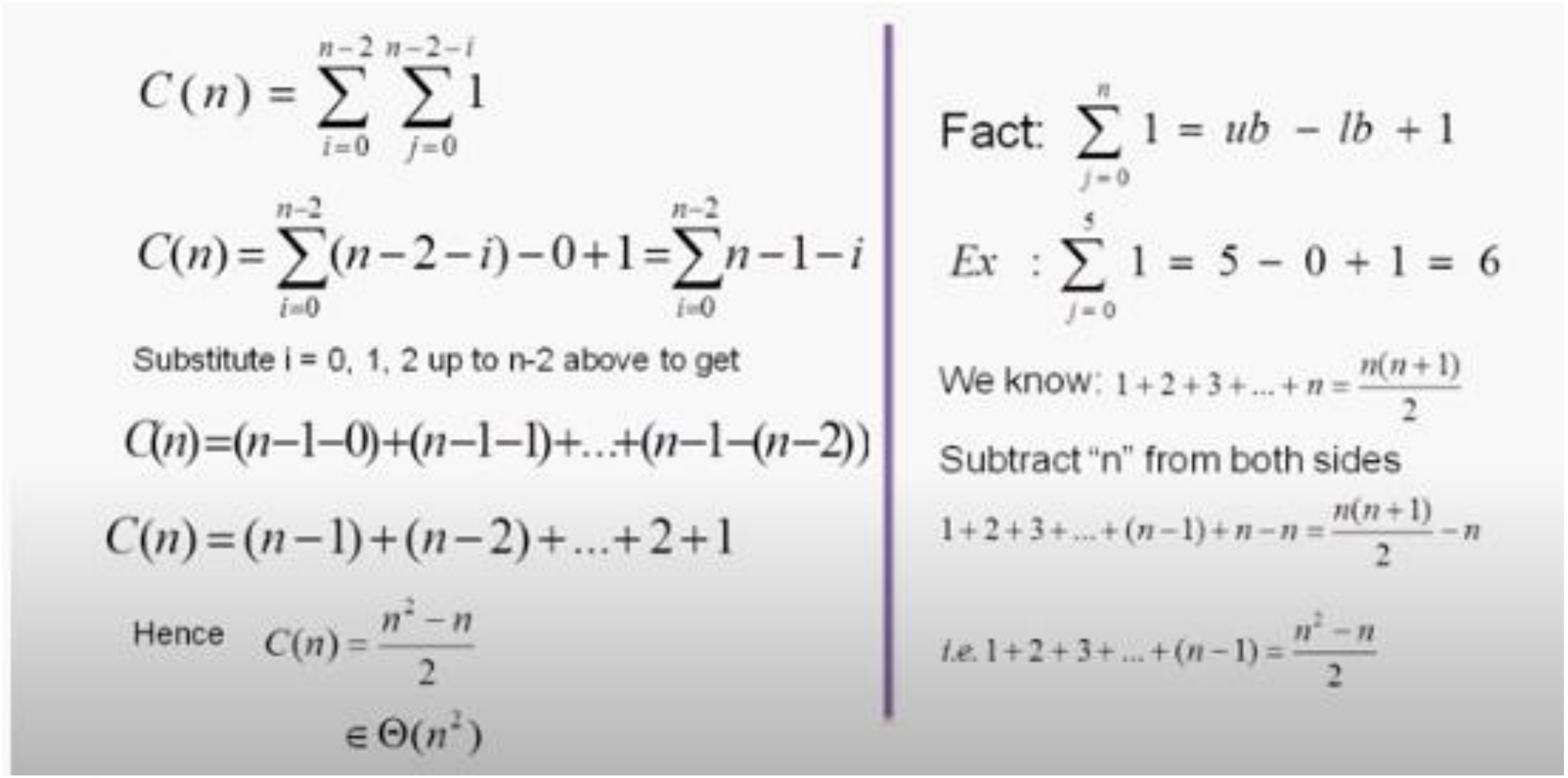

Fifth Iteration

01 ▨	02 ▨	04	05	08	Compare 1 with 2 and 1 < 2
01	02 ▨	04 ▨	05	08	Compare 2 with 4 and 2 < 4
01	02	04 ▨	05 ▨	08	Compare 4 with 5 and 4 < 5
01	02	04	05 ▨	08 ▨	Compare 5 with 8 and 5 < 8

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

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

```
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 $A[j + 1]$
```



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

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

Substitute i = 0, 1, 2 up to n-2 above to get

$$C(n) = (n-1-0) + (n-1-1) + \ldots + (n-1-(n-2))$$

$$C(n) = (n-1) + (n-2) + \ldots + 2 + 1$$

Hence $C(n) = \frac{n^2 - n}{2}$

$\in \Theta(n^2)$

Fact: $\sum_{j=0}^{n} 1 = ub - lb + 1$

$Ex : \sum_{j=0}^{5} 1 = 5 - 0 + 1 = 6$

We know: $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

Subtract "n" from both sides

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

$i.e.\ 1 + 2 + 3 + \ldots + (n-1) = \frac{n^2 - n}{2}$

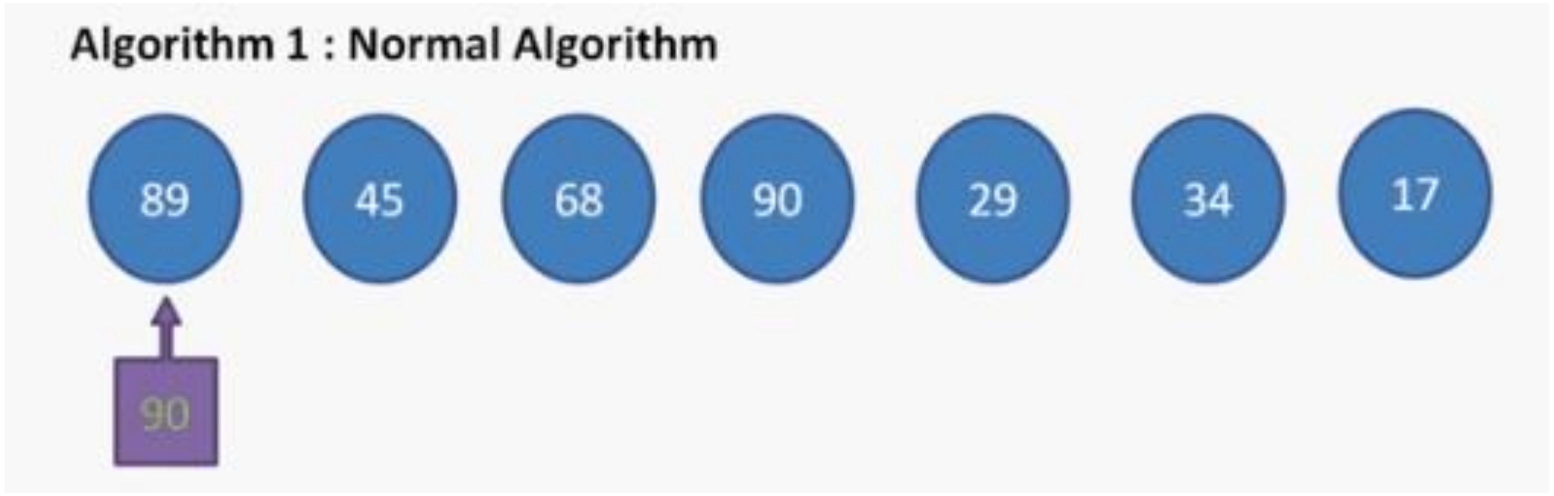
Sequential Search

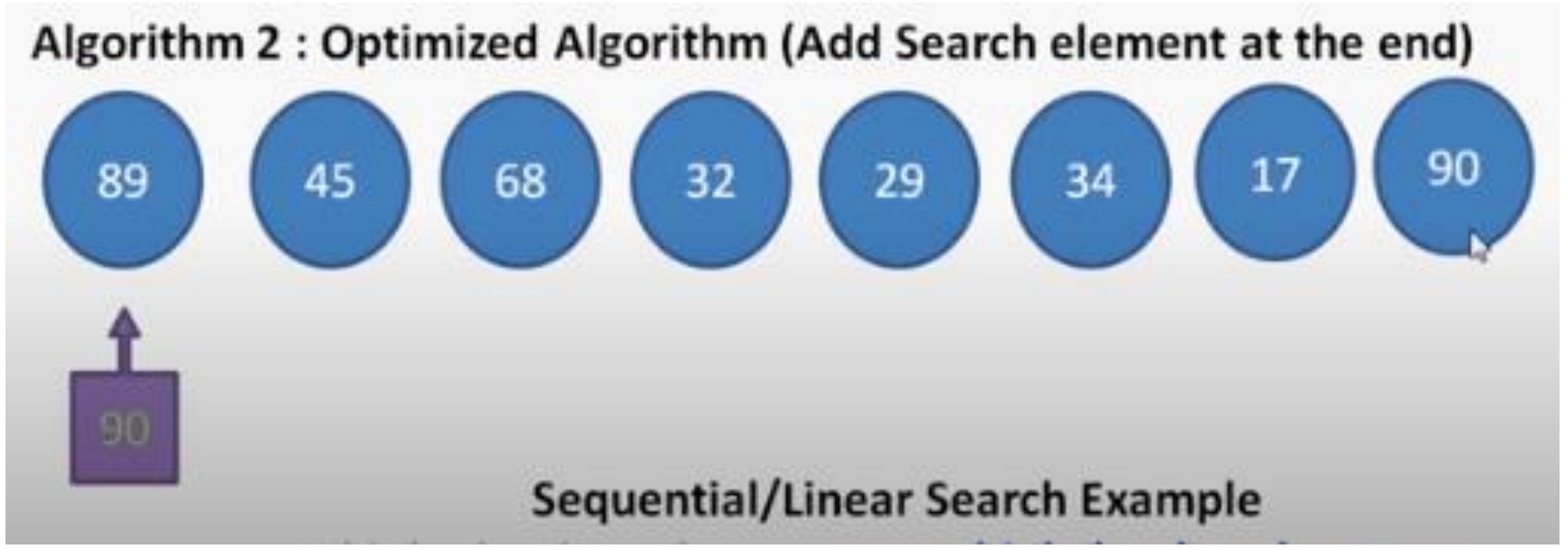
Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

To repeat, the algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

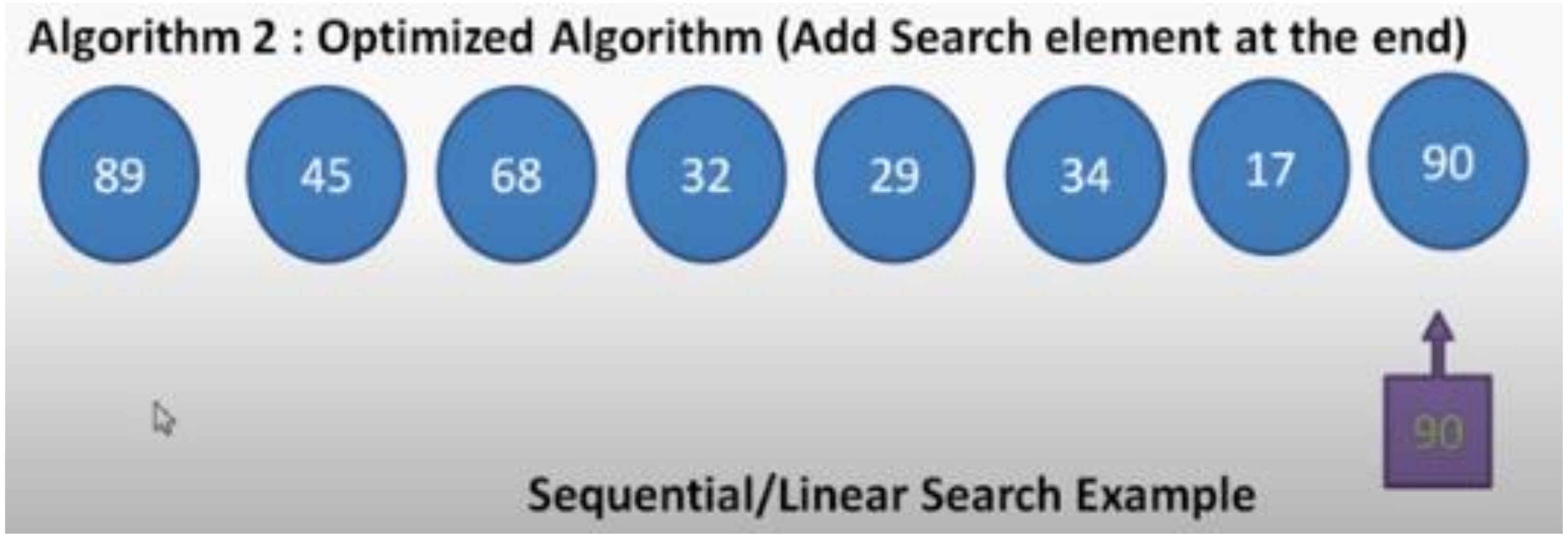
A simple extra trick is often employed in implementing sequential search:

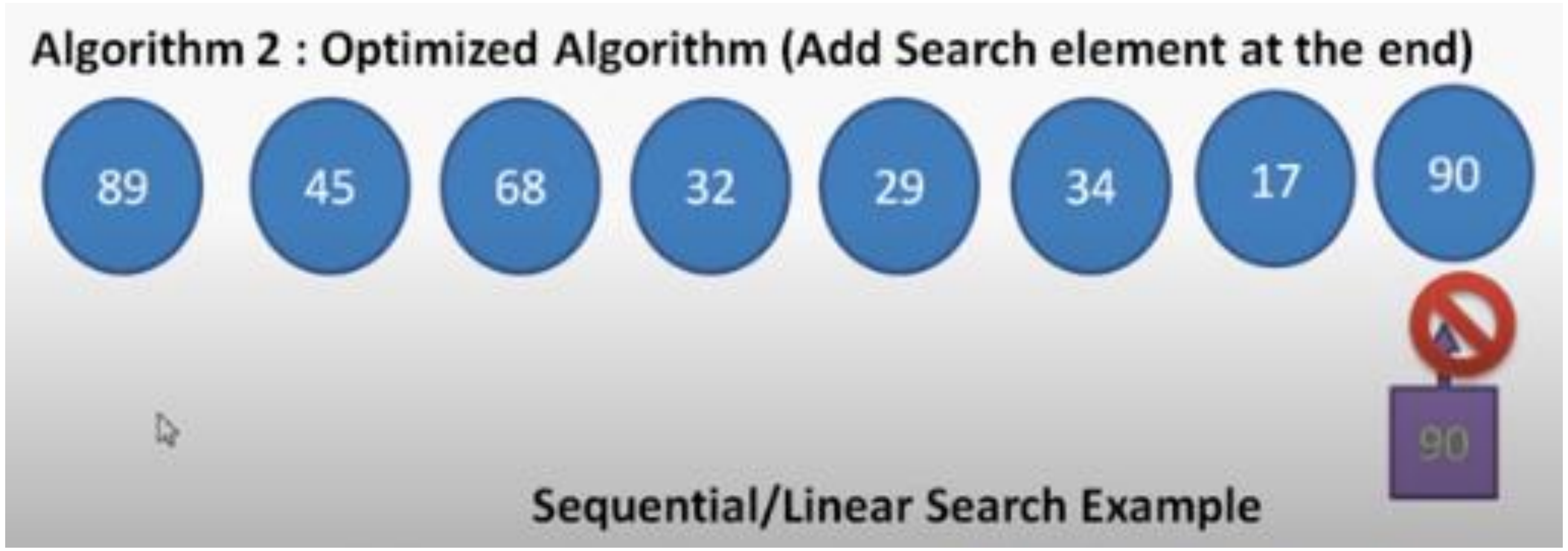
if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether.

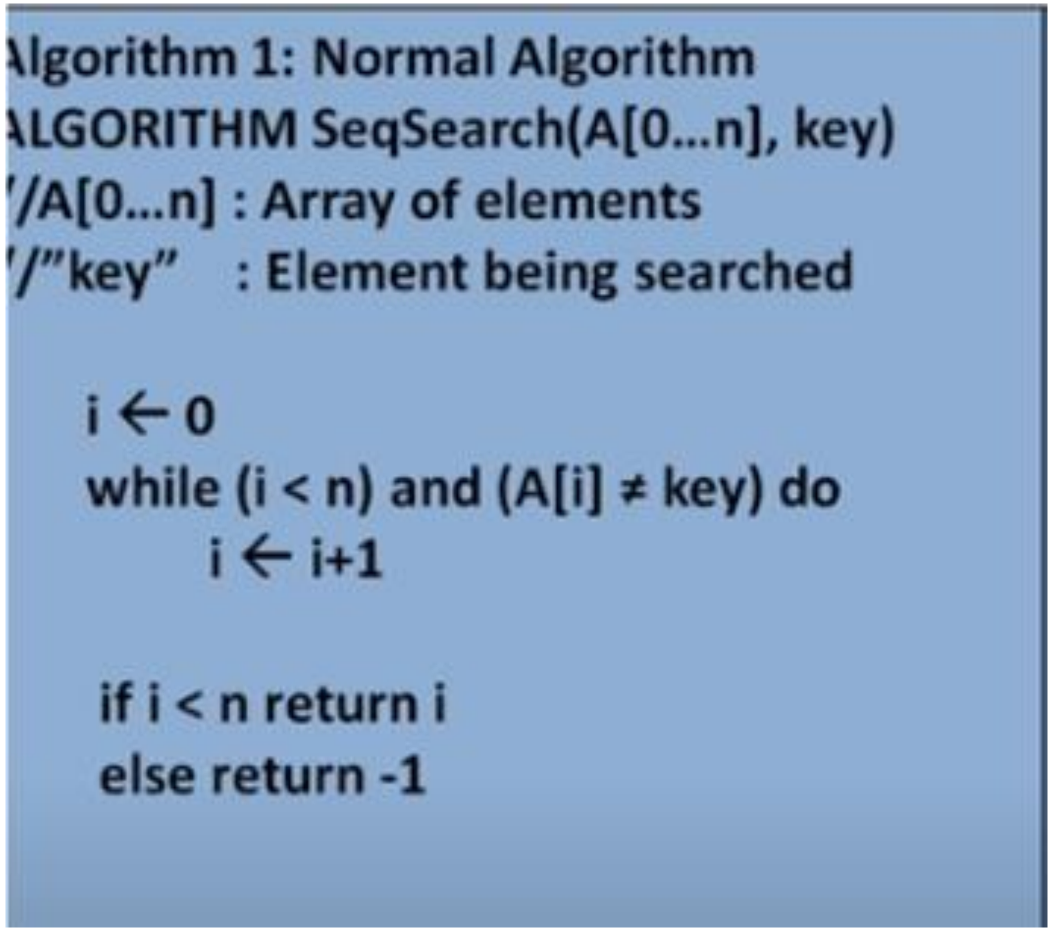

Algorithm 1 : Normal Algorithm


Algorithm 2 : Optimized Algorithm (Add Search element at the end)

Sequential/Linear Search Example


Algorithm 2 : Optimized Algorithm (Add Search element at the end)
89
45
68
32
29
34
17
90
90
Sequential/Linear Search Example

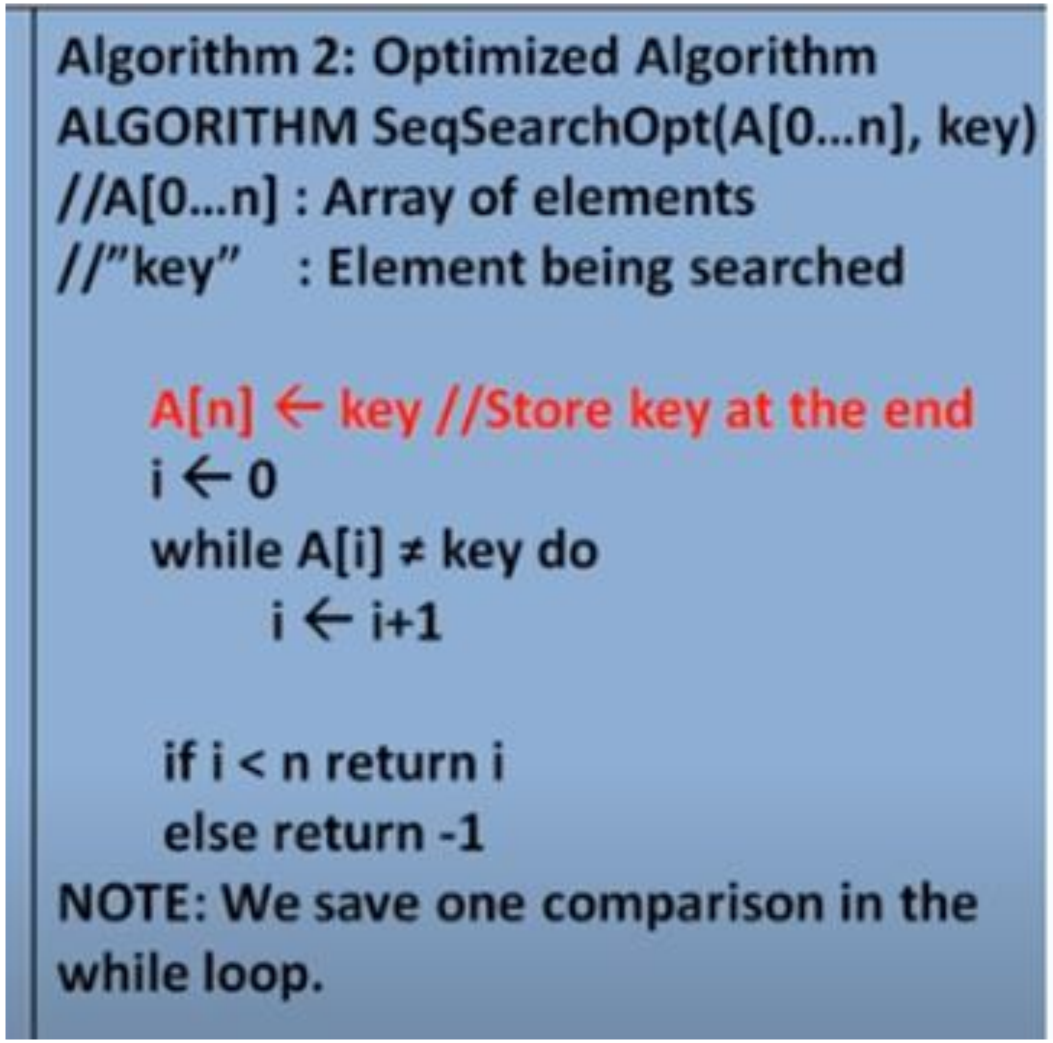

Algorithm 2 : Optimized Algorithm (Add Search element at the end)
89
45
68
32
29
34
17
90
90
Sequential/Linear Search Example



Algorithm 1: Normal Algorithm

ALGORITHM SeqSearch(A[0…n], key)

```
//A[0…n] : Array of elements
//"key"    : Element being searched

  i ← 0
  while (i < n) and (A[i] ≠ key) do
        i ← i+1

  if i < n return i
  else return -1
```



Algorithm 2: Optimized Algorithm

```
ALGORITHM SeqSearchOpt(A[0…n], key)
//A[0…n] : Array of elements
//"key"    : Element being searched

    A[n] ← key //Store key at the end
    i ← 0
    while A[i] ≠ key do
        i ← i+1

    if i < n return i
    else return -1
```

NOTE: We save one comparison in the while loop.

```
ALGORITHM SequentialSearch2(A[0..n], K)
    //Implements sequential search with a search key as a sentinel
    //Input: An array A of n elements and a search key K
    //Output: The index of the first element in A[0..n − 1] whose value is
    //         equal to K or −1 if no such element is found
    $A[n] \leftarrow K$
    $i \leftarrow 0$
    while $A[i] \neq K$ do
        $i \leftarrow i + 1$
    if $i < n$ return $i$
    else return $-1$
```

How Long does it takes ?

How many times do we compare ?
n+1 times   (n times for the n elements of the array, 1 time for extra element that we added at the end)

$$\left.\begin{array}{l}\text{n+1 : unsuccessful search}\\ \text{n : successful search}\end{array}\right\}\quad \text{Worst case Analysis} = O(n)$$

Best Case Analysis - $\Omega(1)$

Average Case Analysis - $\theta(n)$

Brute-Force String Matching

String matching is something crucial for database development and text processing software.

Fortunately, every modern programming language and library is full of functions for string processing that help us in our everyday work.

String algorithms can typically be divided into several categories. One of these categories is string matching.

When it comes to string matching, the most basic approach is what is known as brute force, which simply means to check every single character from the text to match against the pattern. Applications: Text editors, Search engines, Biological research

Applications of String Matching Algorithms:

It is applicable to wide variety of problems, particularly If we have large input size, we may use this Brute force.

Plagiarism Detection: The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.

Bioinformatics and DNA Sequencing: Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.

String Searching: DNA alphabet

DNA alphabet contains only four "letters", forming fixed pairs in the double-helical structure of DNA

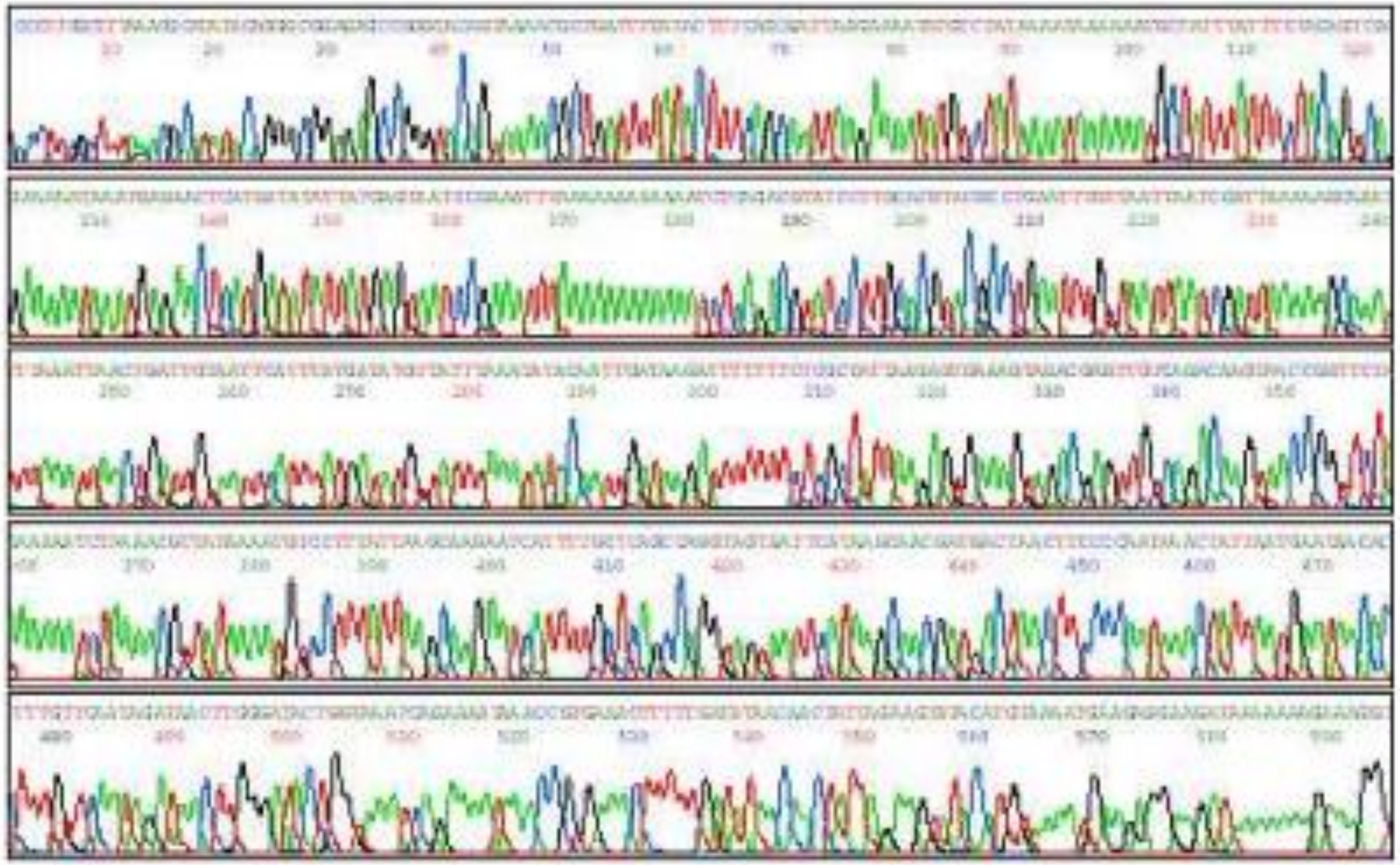
- A – adenine: A pairs with T
- C – cytosine: C pairs with G
- G – guanine: G pairs with C
- T - thymine: T pairs with A

http://www.biotechnologyonline.gov.au/popups/img_helix.html

[illegible]

http://www.insectscience.org/2.10/ref/fig5a.gif

String Searching: DNA alphabet



http://biology.kenyon.edu/courses/biol114/Chap08/longread_sequence.gif

Spelling Checker: Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance state.

Search engines or content search in large databases: To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.

Given a string of n characters called the ***text*** and a string of m characters $(m \leq n)$ 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 $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

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

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text.

Note that the last position in the text that can still be a beginning of a matching substring is $n - m$(provided the text positions are indexed from 0 to $n - 1$).

Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
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 $-1$
```

	0	1	2	3	4	
T	a	b	a	c	a	n=5

	0	1	2	
P	a	b	a	m=3

```
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 $-1$
```

	0	1	2	3	4	
T	b	d	a	c	a	n=5

	0	1	2	
P	a	c	a	m=3

```
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 $-1$
```

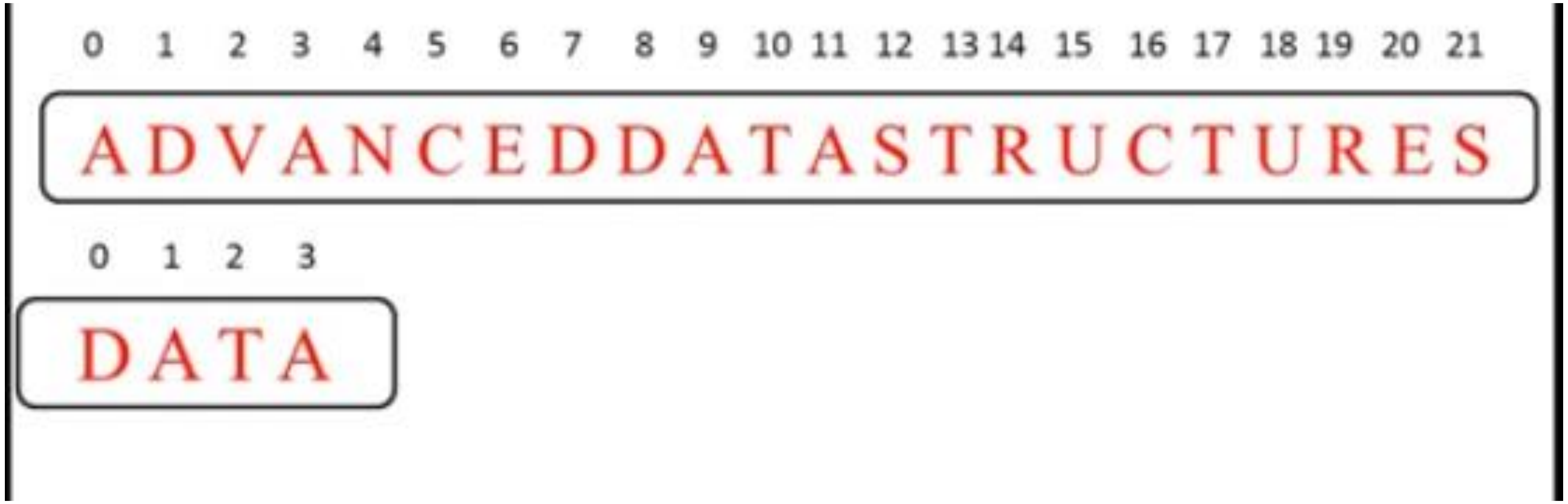
Assignment – 1

Question : 3

	0	1	2	3	4	
T	a	t	e	n	a	n=5

	0	1	2	
P	t	e	n	m=3

```
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 $-1$
```


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
A D V A N C E D D A T A S T R U C T U R E S
0 1 2 3
D A T A

The brute force algorithm consists in checking, at all positions in the text between 0 and *n*-*m*, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The time complexity of brute force string matching is **O(mn)**, which is sometimes written as **O(n*m)** . So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n * m tries.

Worst Case:

Outer loop : (n-m+1) => n
Inner loop : m

=> O(n*m)

Exhaustive Search

Exhaustive search is simply a brute-force approach to combinatorial problems. It is a method obtained by searching each element of given problem.

Exhaustive search applied it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem.

Traveling Salesman Problem : The traveling salesman problem (TSP) 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.

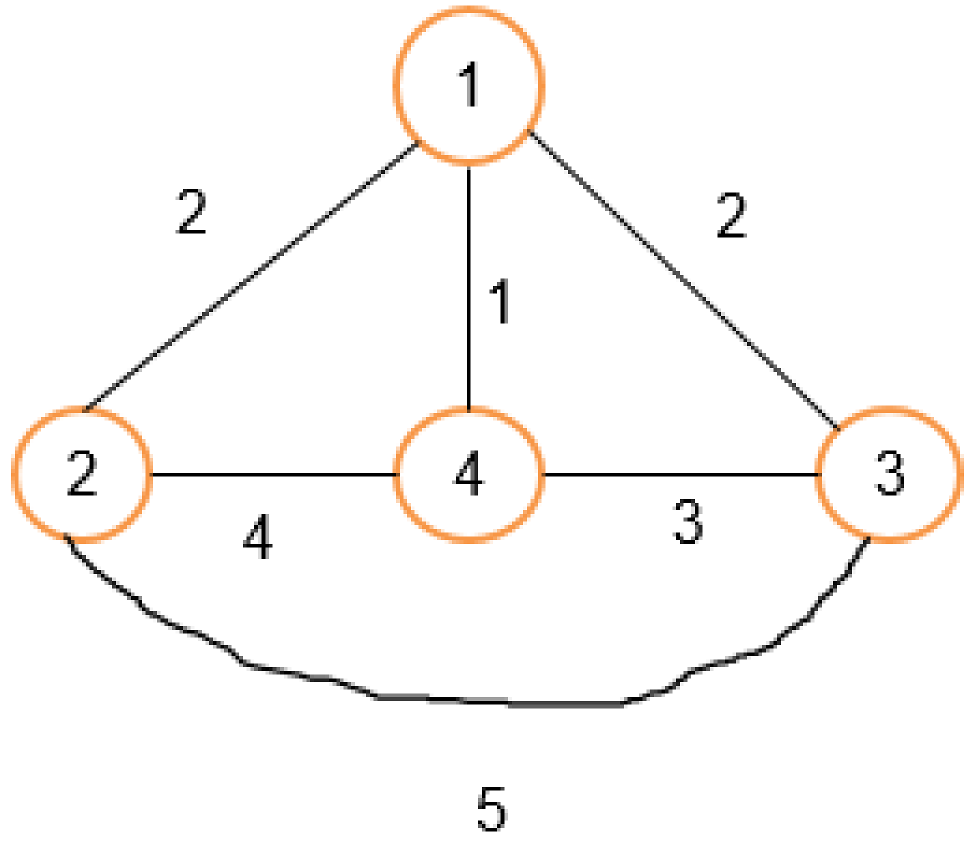
The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

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

Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

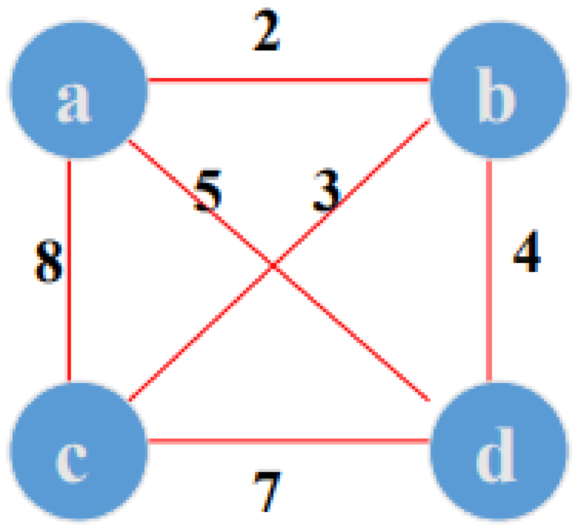
It is easy to see that a Hamiltonian circuit can also be defined as a sequence of n + 1 adjacent vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$. where the first vertex of the sequence is the same as the last one and all the other n − 1 vertices are distinct.

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.

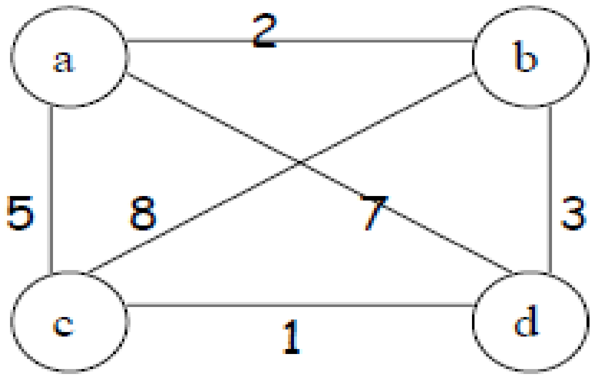


1->3->4->2->1	2+3+4+2=11	Optimal
1->2->3->4->1	2+5+3+1=11	Optimal
1->3->2->4->1	2+5+4+1=12	
1->2->4->3->1	2+4+3+2=11	Optimal
1->4->2->3->1	1+4+5+2=12	
1->4->3->2->1	1+3+5+2=11	Optimal

Traveling salesman by exhaustive search



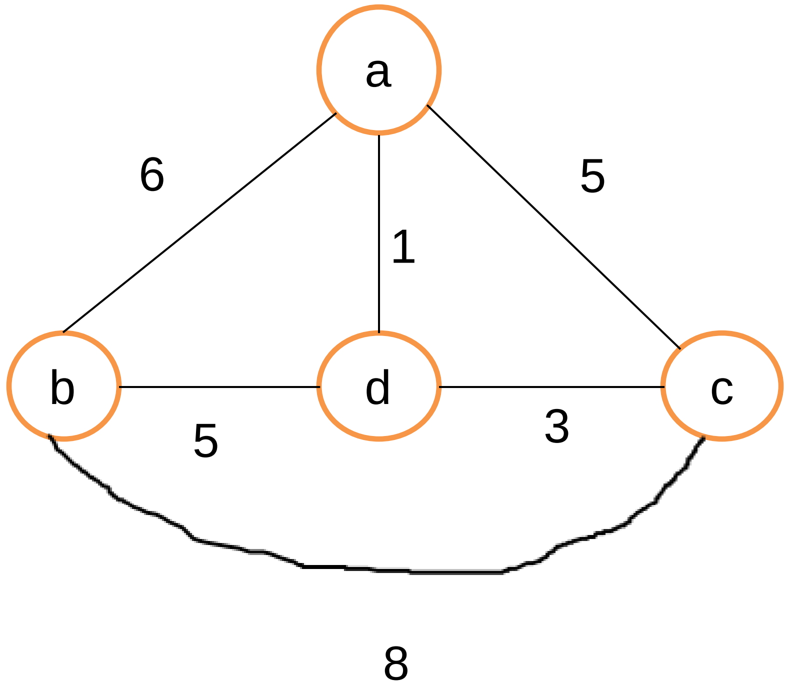
Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	l = 2 + 8 + 1 + 7 = 18	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	l = 2 + 3 + 1 + 5 = 11	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	l = 5 + 8 + 3 + 7 = 23	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	l = 5 + 1 + 3 + 2 = 11	optimal
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	l = 7 + 3 + 8 + 5 = 23	

Assignment : 1

4. Traveling Salesman Problem

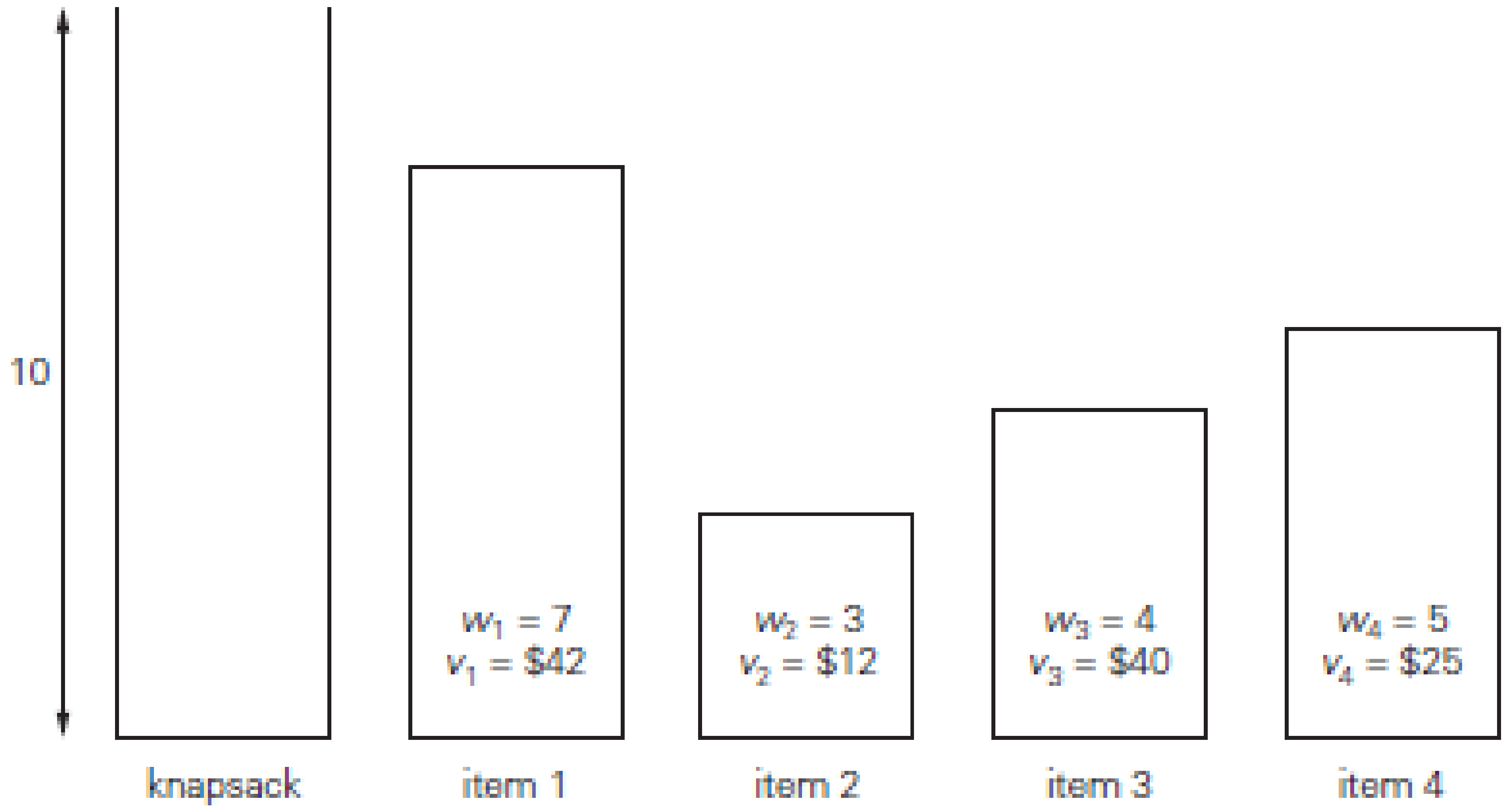


KNAPSACK PROBLEM

The problem states that: given n items of known weights $w^1, w^2, \ldots w^n$ and values $v^1, v^2, \ldots, v^n$ and a knapsack of capacity w, find the most valuable subset of the items that fit into the knapsack.

Eg Items in the shopping basket

Problem Statement: To fill up the sack with items, such a way that getting the maximum profit out of that.


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

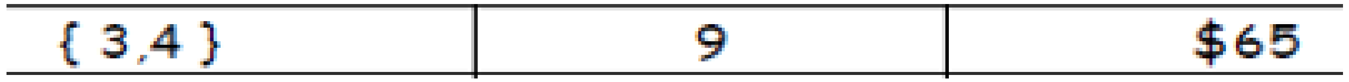
(a)

Example:

$W=10$, $w_1, w_2, w_3, w_4 = \{ 7,3,4,5 \}$ and $v_1, v_2, v_3, v_4 = \{ 42,12,40,25 \}$

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

The above figure presents a small instance of the knapsack problem in which 65 is the maximum value. Considering that as a optimal solution,



{ 3,4 }	9	$65

This problem considers all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets and finding the largest among the values, which is an optimal solution.

- The number of subsets of an n-element set is 2^n the search leads to a $\Omega(2^n)$ algorithm, which is not based on the generation of individual subsets.

Thus, for both TSP and knapsack, exhaustive search leads to algorithms that are inefficient on every input.

- These two problems are the best known examples of NP-hard problems.
- No polynomial-time algorithm is known for any NP-hard problem.
- The two methods Backtracking and Branch & bound enable us to solve this problem in less than exponential time.

- Algorithm:
  - We go through all combinations and find the one with maximum value and with total weight less or equal to *W*

- **Efficiency:**
  - Since there are *n* items, there are 2^n possible combinations of items.
  - Thus, the running time will be $O(2^n)$

ASSIGNMENT PROBLEM

The problem is: given 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 if the ith person is assigned to the jth job is a known quantity c[i,j] for each pair i, j=1,2,….,n. The problem is to find an assignment with the minimum total cost.

	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

It is easy to see that an instance of the assignment problem is completely specified by its cost matrix *C*.

In terms of this matrix, the problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

Note that no obvious strategy for finding a solution works here. For example, we cannot select the smallest element in each row, because the smallest elements may happen to be in the same column.

In fact, the smallest element in the entire matrix need not be a component of an optimal solution. Thus, opting for the exhaustive search.

We can describe feasible solutions to the assignment problem as *n*-tuples *j*1, . . . , *jn* in which the *i*th component, *i* = 1, . . . , *n*, indicates the column of the element selected in the *i*th row (i.e., the job number assigned to the *i*th person).

For example, for the cost matrix above, 2, 3, 4, 1 indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1.

The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first *n* integers.

	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

Therefore, 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

Divide and Conquer

Divide-and-conquer is probably the best-known general algorithm design technique

Divide-and-conquer algorithms work according to the following general plan:

1. Divide: Divide the problem into a number of smaller sub-problems ideally of about the same size.

2. Conquer: The smaller sub-problems are solved, typically recursively. If the sub-problem sizes are small enough, just solve the sub-problems in a straight forward manner.

3. Combine: If necessary, the solution obtained the smaller problems are connected to get the solution to the original problem.

As an example, let us consider the problem of computing the sum of *n* numbers

$a_0, \ldots, a_{n-1}$. If $n > 1$

we can divide the problem into two instances of the same problem: to compute the sum of the first [n/2] numbers and to compute the sum of the remaining [n/2] numbers.

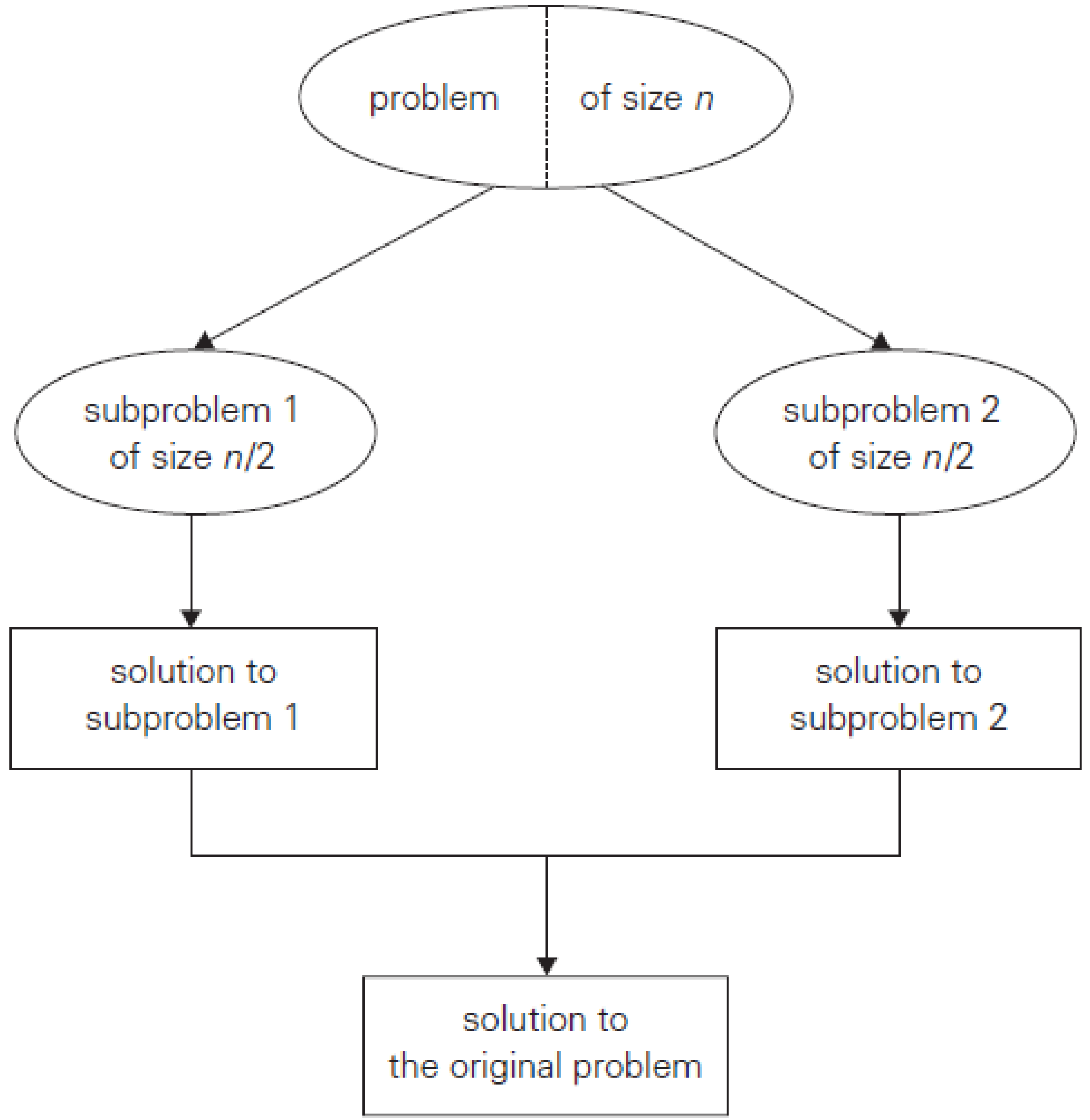
Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

Ex:

	Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]
Arr	2	3	4	5	1

Divide-and-conquer technique :


problem of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
solution to subproblem 1
solution to subproblem 2
solution to the original problem

GENERAL METHOD

Suppose we consider the divide-and-conquer strategy, when it splits the input into two sub problem of the same kind as the original problem.

A **control abstraction** is a procedure whose flow of **control** is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.

The **control abstraction for divide and conquer** technique is DANDC(P), where P is the problem to be solved.

Control Abstraction for Divide and Conquer

```
Algorithm DAndC
{
if small(p) then return s(p);
else
{
        Divide p into smaller instructions $p_1$, $p_2$, $p_3$….. $p_k$;   where k>1

        Apply DAndC  to each of the sub problems;

        return combine(DAndC($p_1$),DAndC($p_2$)….D&C($p_k$));
}
}
```

Algorithm: Control abstraction for divide-and-conquer DandC(p) is the divide-and-conquer algorithm, where P is the problem to be solved.

Small(p) is a Boolean valued function(i.e., either true or false) that determines whether the input size is small enough that the answer can be computed without splitting.

If this, is so the function S is invoked. Otherwise the problem P is divided into smaller sub-problems. These sub-problems P1, P2, P3……..Pk, are solved by receive applications of DandC.

Combine is a function that combines the solution of the K sub-problems to get the solution for original problem ‘P. If the size of p is n and the sizes of the k sub problems are n_1, n_2, ... $n_{k.}$

Computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

T(n) is the time for DAndC on any input of size n and g(n) is the time to compute the answer directly for small inputs. The function f(n) is the time for dividing P and combining the solutions to sub problems

For divide and conquer based algorithms that produce sub problems of same type as original problem, The complexity of divide and conquer can be given by,

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

where a and b are known constants.

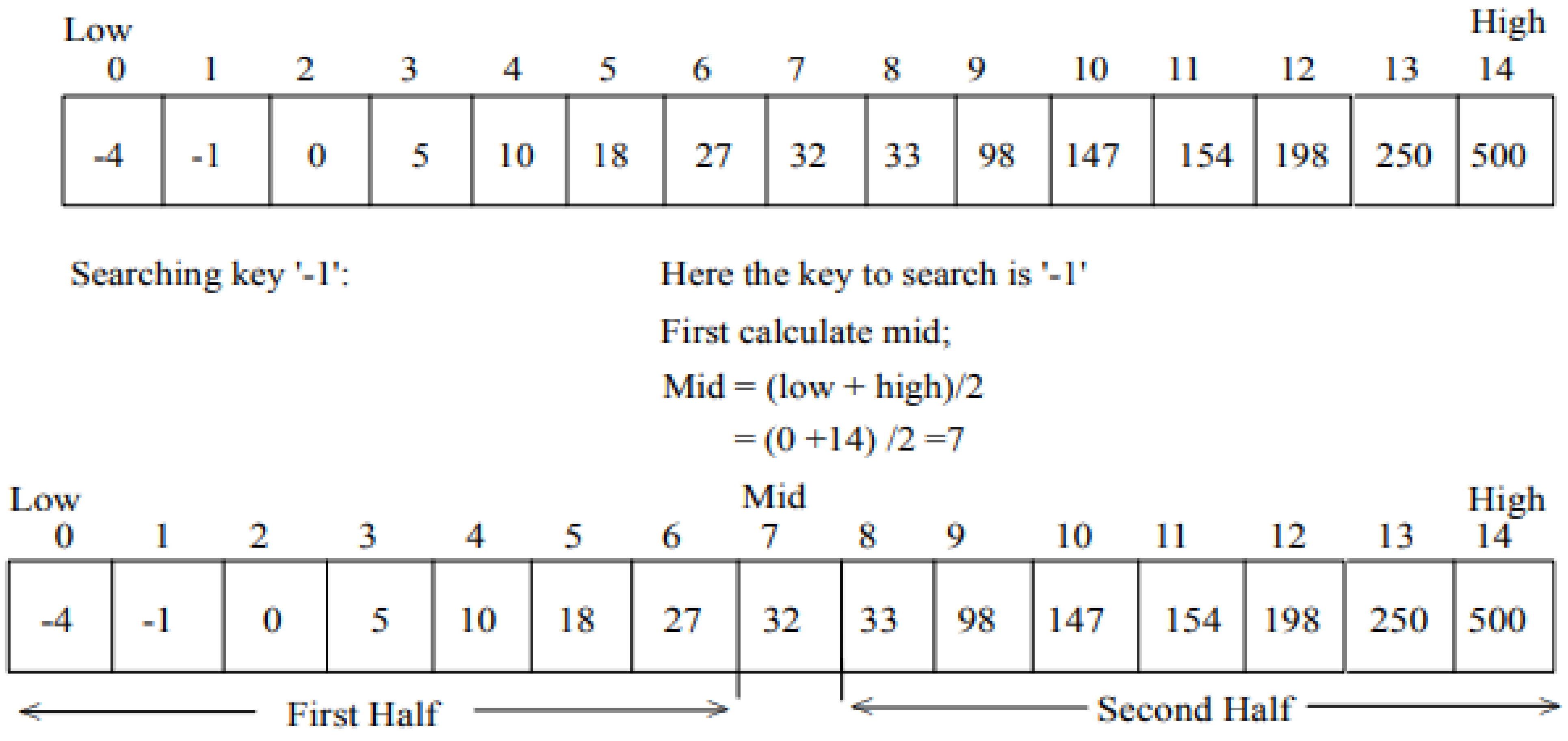
BINARY SEARCH

Binary search is an efficient searching technique that works with only sorted lists. So the list must be sorted before using the binary search method. Binary search is based on divide-and conquer technique.

The process of binary search is as follows: The method starts with looking at the middle element of the list. If it matches with the key element, then search is complete. Otherwise, the key element may be in the first half or second half of the list. If the key element is less than the middle element, then the search continues with the first half of the list.

If the key element is greater than the middle element, then the search continues with the second half of the list. This process continues until the key element is found or the search fails indicating that the key is not there in the list

Consider the list of elements: -4, -1, 0, 5, 10, 18, 32, 33, 98, 147, 154, 198, 250, 500. Trace the binary search algorithm searching for the element -1.



Low														High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Searching key '-1':

Here the key to search is '-1'

First calculate mid;

Mid = (low + high)/2

= (0 +14) /2 =7

Low							Mid							High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half →　　　← Second Half →

Here, the search key -1 is less than the middle element (32) in the list. So the search process continues with the first half of the list.

Low						High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid = (0+7)/2
=3.

Low			Mid			High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

$\leftarrow$ First Half $\rightarrow$ $\leftarrow$ Second Half $\rightarrow$

The search key '-1' is less than the middle element (5) in the list. So the search process continues with the first half of the list.

Low		High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid= (0+2)/2
=1

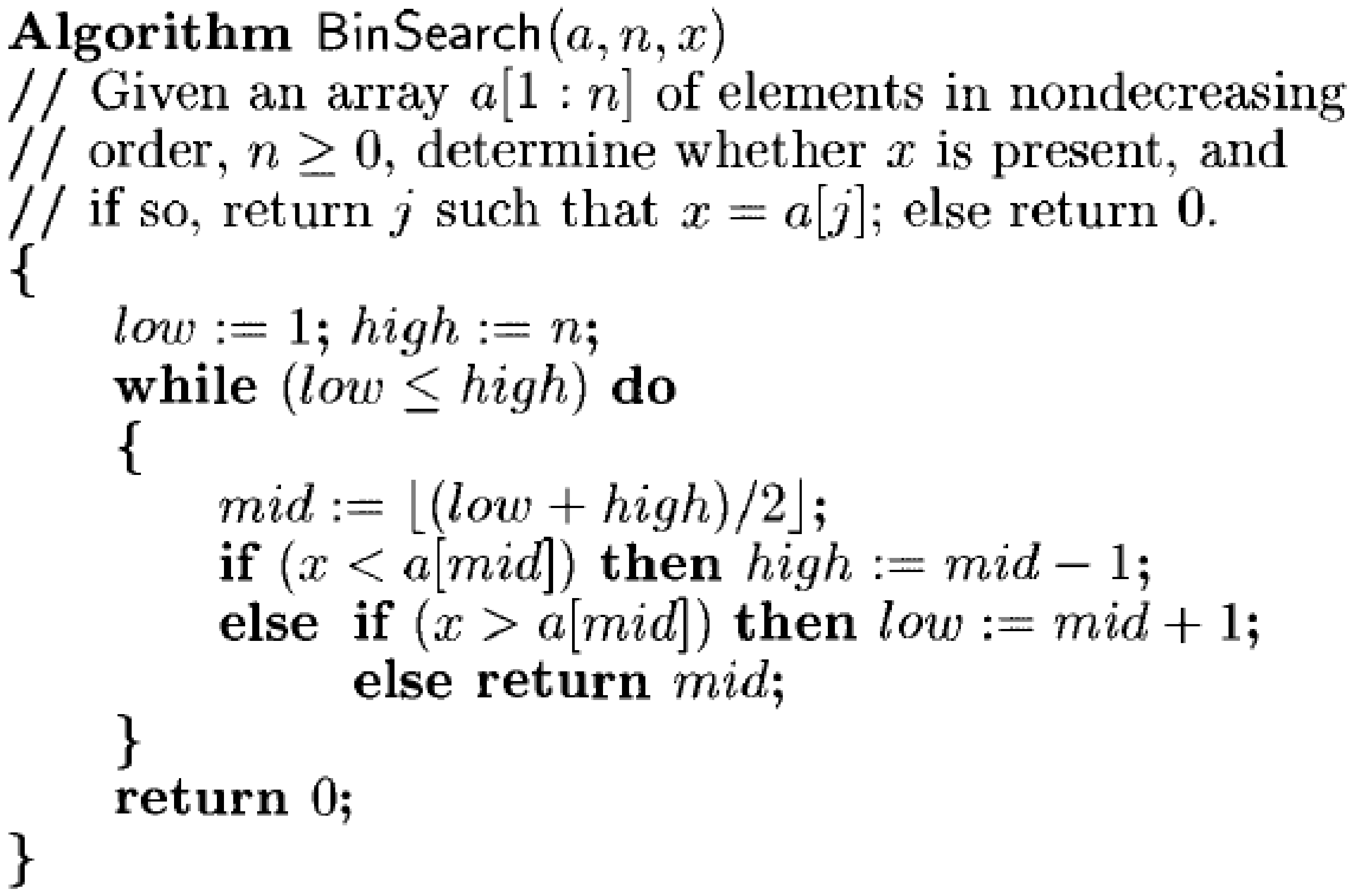
Low	Mid	High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Here, the search key -1 is found at position 1.

Recursive Binary Search:

```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i:l]$ of elements in nondecreasing
// order, $1 \le i \le l$, 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($a, mid + 1, l, x$);
    }
}
```

Iterative Binary Search:



```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// order, n ≥ 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 ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid − 1;
        else if (x > a[mid]) then low := mid + 1;
             else return mid;
    }
    return 0;
}
```

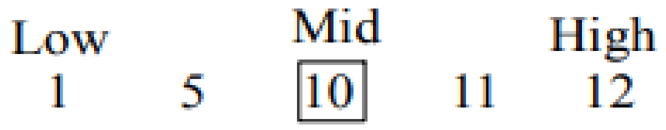
Advantages of Binary Search: The main advantage of binary search is that it is faster than sequential (linear) search. Because it takes fewer comparisons, to determine whether the given key is in the list, then the linear search method

Disadvantages of Binary Search: The disadvantage of binary search is that can be applied to only a sorted list of elements. The binary search is unsuccessful if the list is unsorted.

Efficiency of Binary Search: To evaluate binary search, count the number of comparisons in the best case, average case, and worst case.

Efficiency of Binary Search: To evaluate binary search, count the number of comparisons in
the best case, average case, and worst case.

Best Case: The best case occurs if the middle element happens to be the key element. Then only one comparison is needed to find it. Thus the efficiency of binary search is Ω(1).
Ex: Let the given list is: 1, 5, 10, 11, 12. Let key = 10.



Since the key is the middle element and is found at our first attempt

Worst Case: Assume that in worst case, the key element is not there in the list. So the process of divides the list in half continues until there is only one item left to check.

Items left to search	*Comparisons so far*
16	0
8	1
4	2
2	3
1	4

For a list of size 16, there are 4 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half.

In general, if n is the size of the list and c is the number of comparisons, then

$$C = \log_2 n$$

Eficiency in worst case = O(log n)

Average Case: In binary search, the average case efficiency is near to the worst case efficiency.

So the average case efficiency will be taken as O(log n).

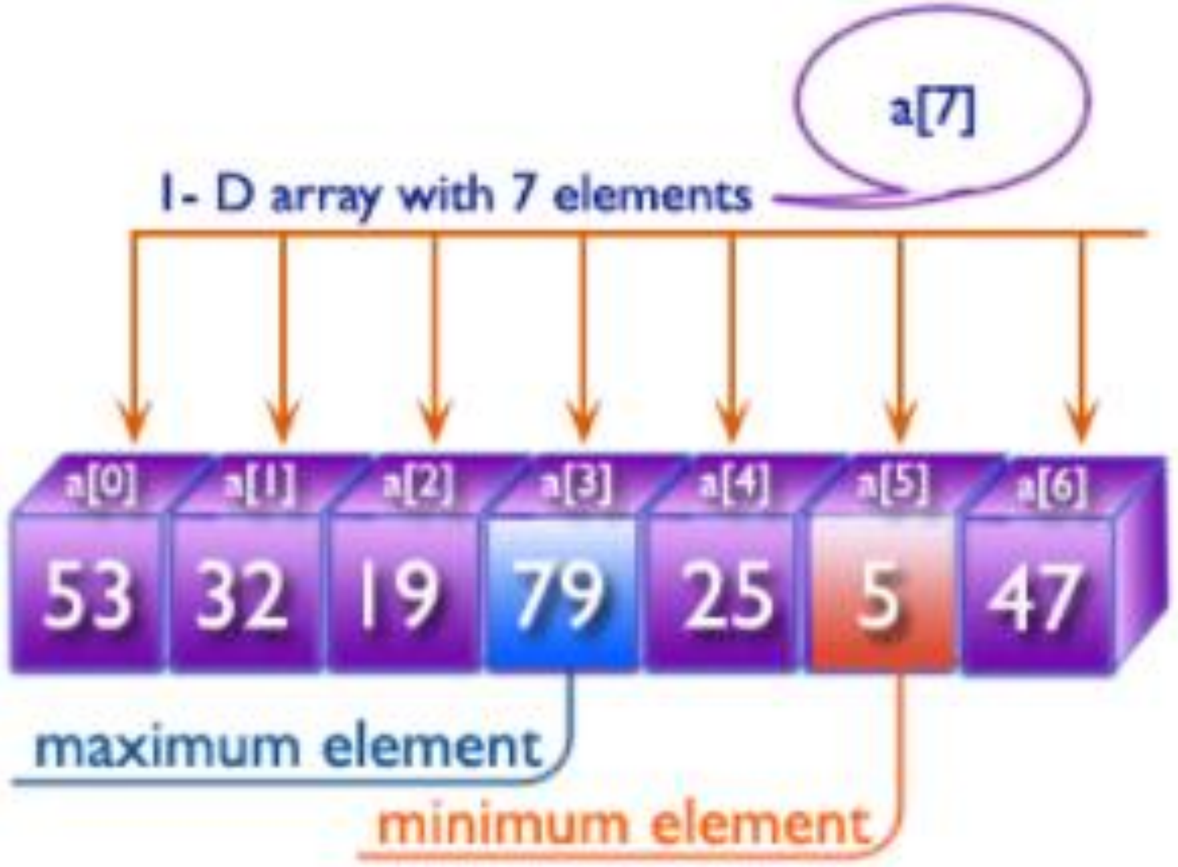
Efficiency in average case = O (log n).

The Time complexity of Binary Search is O (log n)

FINDING THE MAXIMUM AND MINIMUM

Problem statement: Given a list of n elements, the problem is to find the maximum and minimum items.

Example :


a[7]
I- D array with 7 elements
a[0] a[1] a[2] a[3] a[4] a[5] a[6]
53 32 19 79 25 5 47
maximum element
minimum element

Straight forward Maximum & Minimum

```
Algorithm StraightMaxMin($a, n, max, min$)
// Set $max$ to the maximum and $min$ to the minimum of $a[1:n]$.
{
    $max := min := a[1]$;
    for $i := 2$ to $n$ do
    {
        if $(a[i] > max)$ then $max := a[i]$;
        if $(a[i] < min)$ then $min := a[i]$;
    }
}
```

	a[1]	a[2]	a[3]	a[4]	a[5]
a	2	3	4	5	1

Divide & Conquer Method:

1. Divide the list into small groups.
2. Then find max and min of each group.
3. The max/min of result must be one of maxs and mins of the groups.

Example:

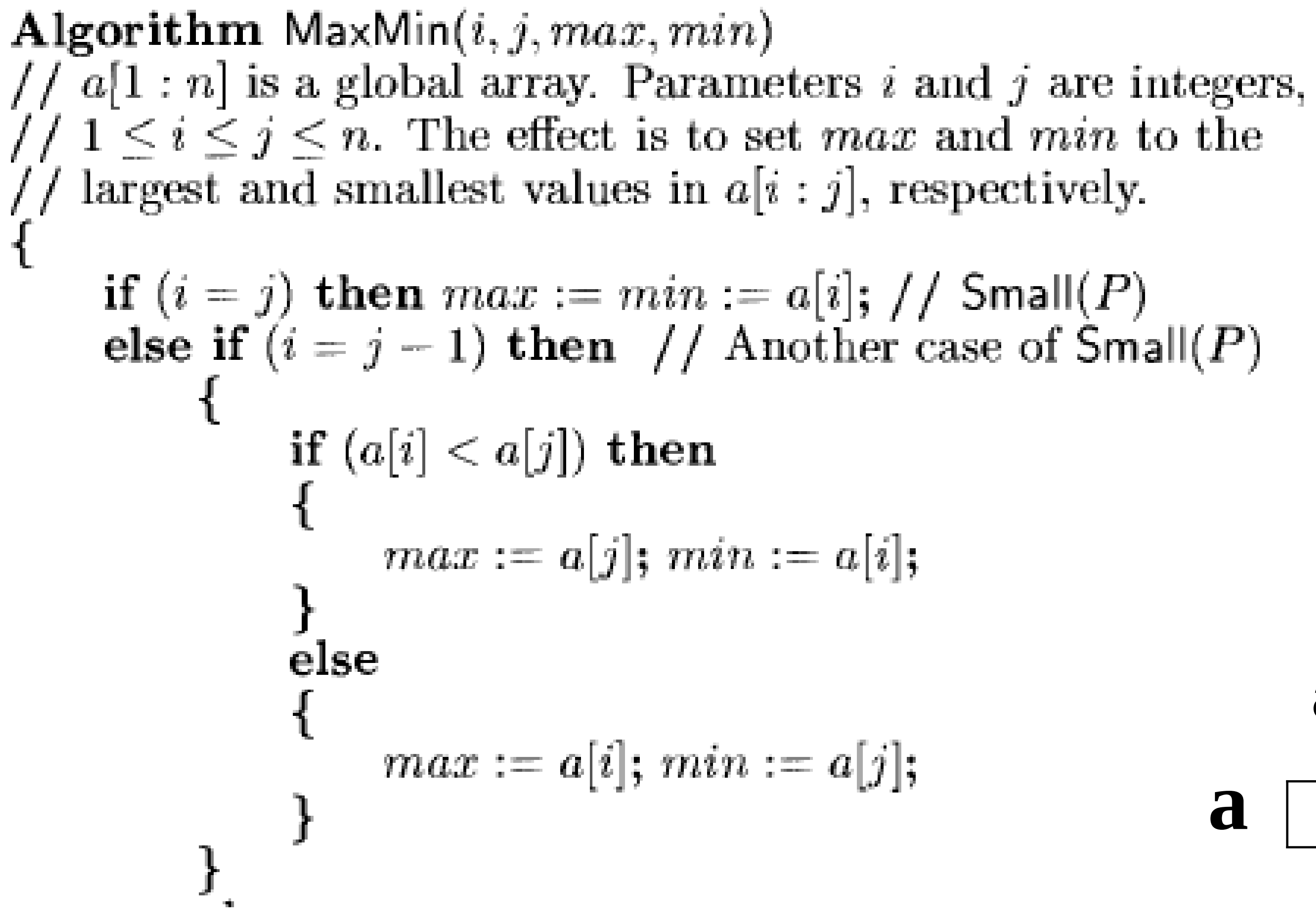
$$A = [5, 7, 1, 4, 10, 6]$$

$$A_1 = [5, 7, 1],\ \max(A_1) = 7,\ \min(A_1) = 1$$

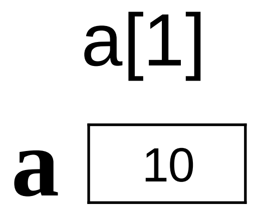
$$A_2 = [4, 10, 6],\ \max(A_2) = 10,\ \min(A_2) = 4$$

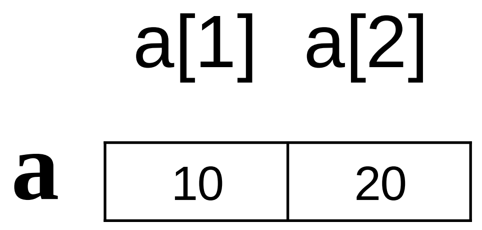
So the min and max of A is $\min(1, 4)$ and $\max(7, 10)$, i.e 1 and 10.

Recursively finding the Maximum & Minimum



```
Algorithm MaxMin($i, j, max, min$)
// $a[1:n]$ is a global array. Parameters $i$ and $j$ are integers,
// $1 \le i \le j \le n$. The effect is to set $max$ and $min$ to the
// largest and smallest values in $a[i:j]$, respectively.
{
    if ($i = j$) then $max := min := a[i]$; // Small($P$)
    else if ($i = j - 1$) then  // Another case of Small($P$)
        {
            if ($a[i] < a[j]$) then
            {
                $max := a[j]$; $min := a[i]$;
            }
            else
            {
                $max := a[i]$; $min := a[j]$;
            }
        }.
```





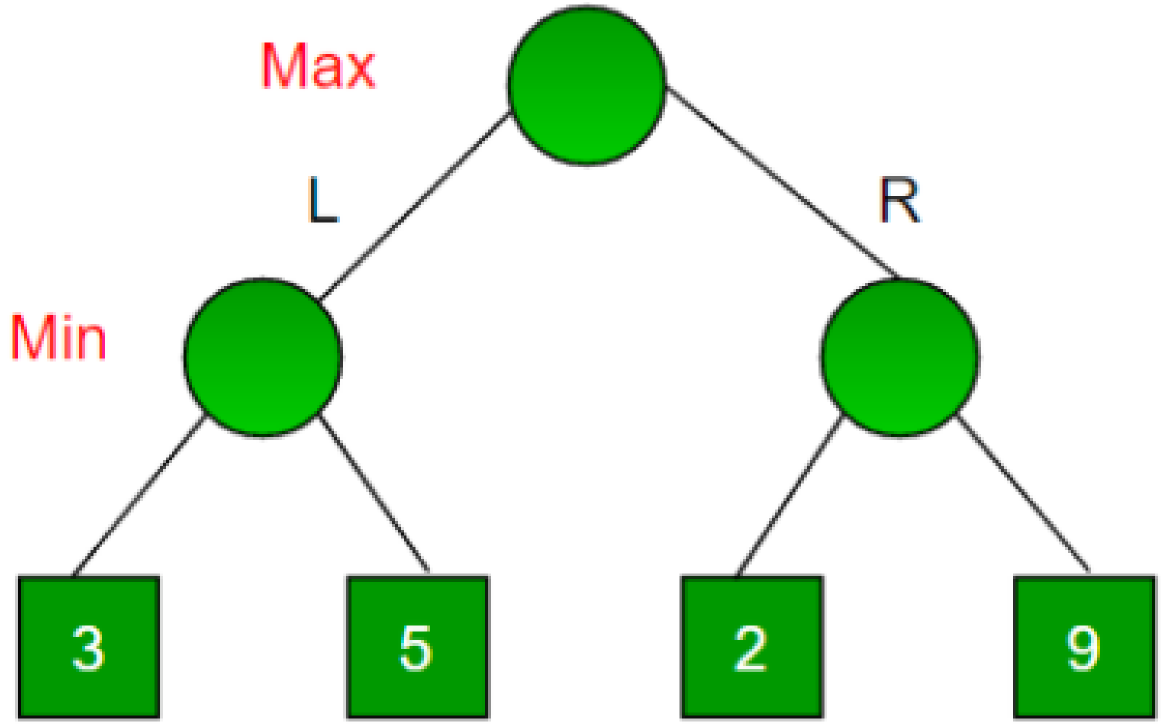
```
    else
    {   // If $P$ is not small, divide $P$ into subproblems.
        // Find where to split the set.
            $mid := \lfloor (i+j)/2 \rfloor$;
        // Solve the subproblems.
            MaxMin($i, mid, max, min$);
            MaxMin($mid+1, j, max1, min1$);
        // Combine the solutions.
            if ($max < max1$) then $max := max1$;
            if ($min > min1$) then $min := min1$;
    }
}
```

$A = [5, 7, 1, 4, 10, 6]$

$A_1 = [5, 7, 1]$, $\max(A_1) = 7$, $\min(A_1) = 1$

$A_2 = [4, 10, 6]$, $\max(A_2) = 10$, $\min(A_2) = 4$

So the min and max of A is $\min(1, 4)$ and $\max(7, 10)$, i.e 1 and 10.


Max
L
R
Min
3
5
2
9

Analysis of MaxMin Algorithm

Analyzing the time complexity of the algorithm , Concentrates on the number of element comparisions

If only one element is present, then comparison is not required.
Time Complexity T(n)=0, n = 1

If there are two elements in the list then we require 1 comparision
Time Complexity T(n)=1, n = 2

If there are more than two elements in the given list ,

T(n) = T(n/2) + T(n/2) +2 ……………… n>2

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

$= 2[2T(n/4)+2]+2$

$= 4T(n/4)+4+2$

$= 4\ [2T(n/8)+2]+4+2$

$= 8T(n/8)+ 8+4+2$

$T(n/2) = 2T(n/4)+2$

$T(n/4) = 2T(n/8)+2$

.........assume here $k=4$ and $n=2^k$

$= 2^{4-1}T(2^4/2^3) + 2^3+2^2+2^1$

$= 2^{k-1}T(2) + \sum_{1\le i \le k-1} 2^i$

$T(n) = 2^{k-1}T(2) + \sum_{1 \le i \le k-1} 2^i$

$= 2^{k-1} + 2^k - 2$ ....$T(2)=1$

$=(2^k/2) + 2^k - 2$

$=(n/2) + n - 2$ ....$n = 2^k$

$T(n)=(3n/2)-2$

Time Complexity for MaxMin Algorithm using Divide & Conquer Method is (3n/2)-2

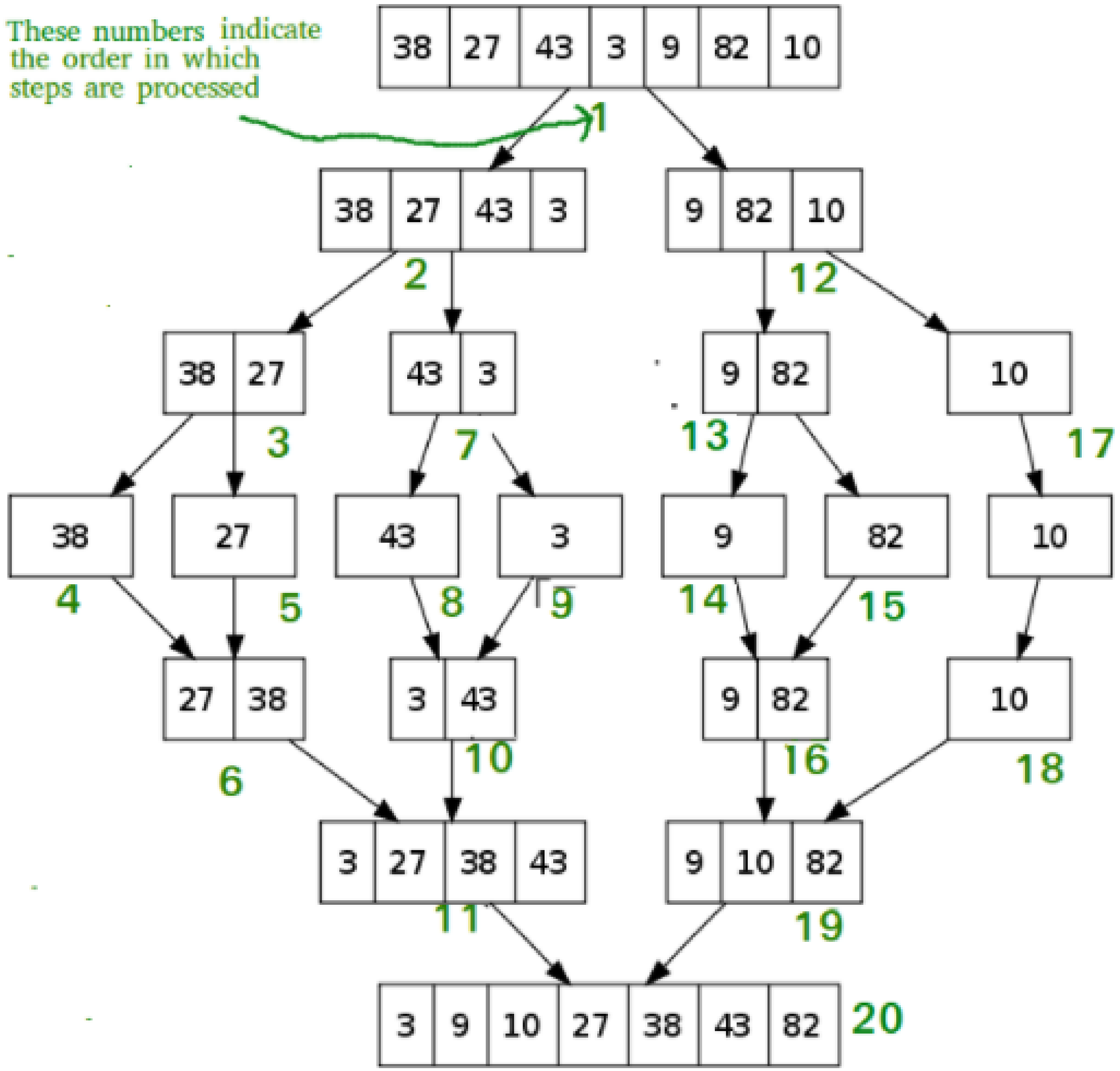
Merge sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

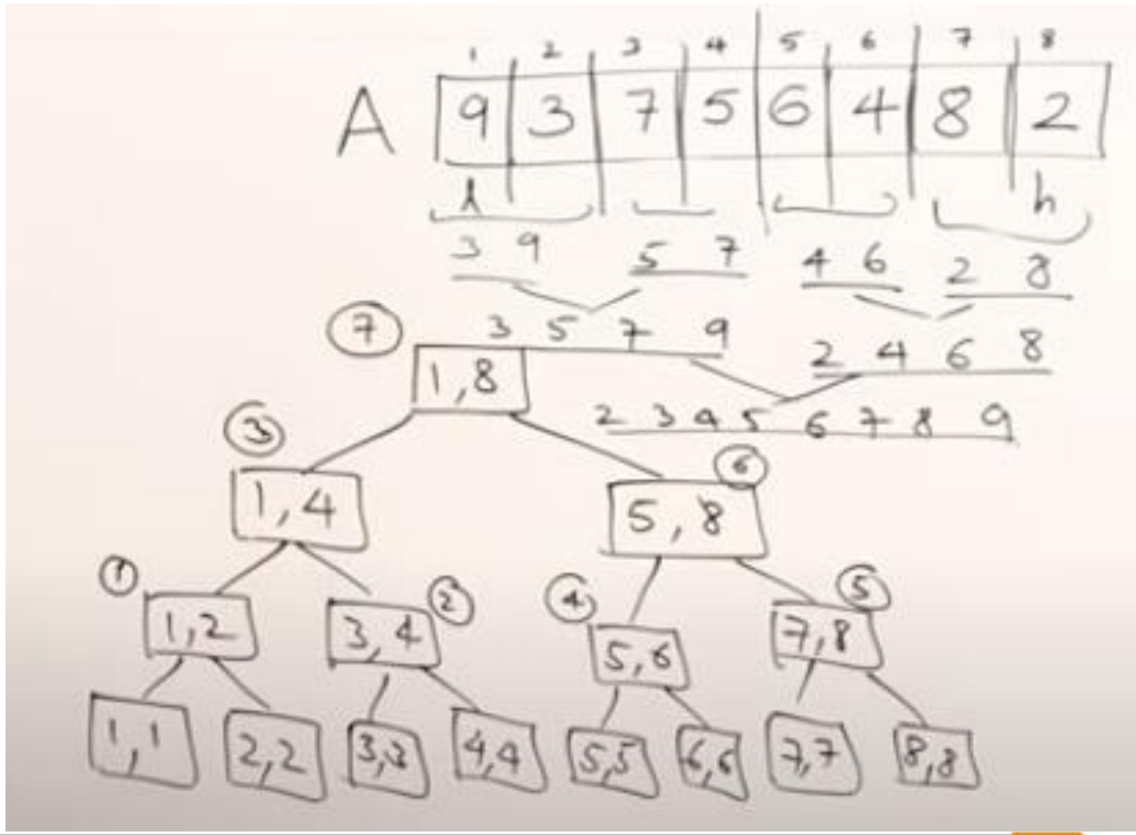
It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..n/2-1]$ and $A[n/2..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Mergesort Example:1

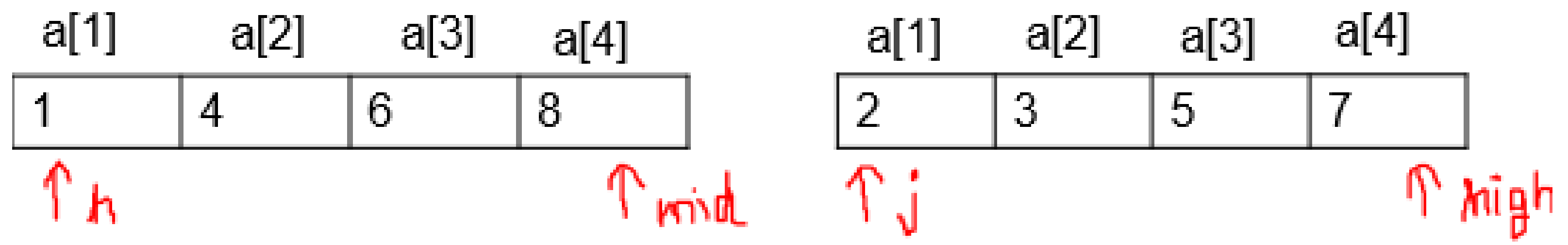


```
Algorithm MergeSort($low, high$)
// $a[low : high]$ is a global array to be sorted.
// Small($P$) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if ($low < high$) then  // If there are more than one element
    {
        // Divide $P$ into subproblems.
            // 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($low, mid, high$);
    }
}
```

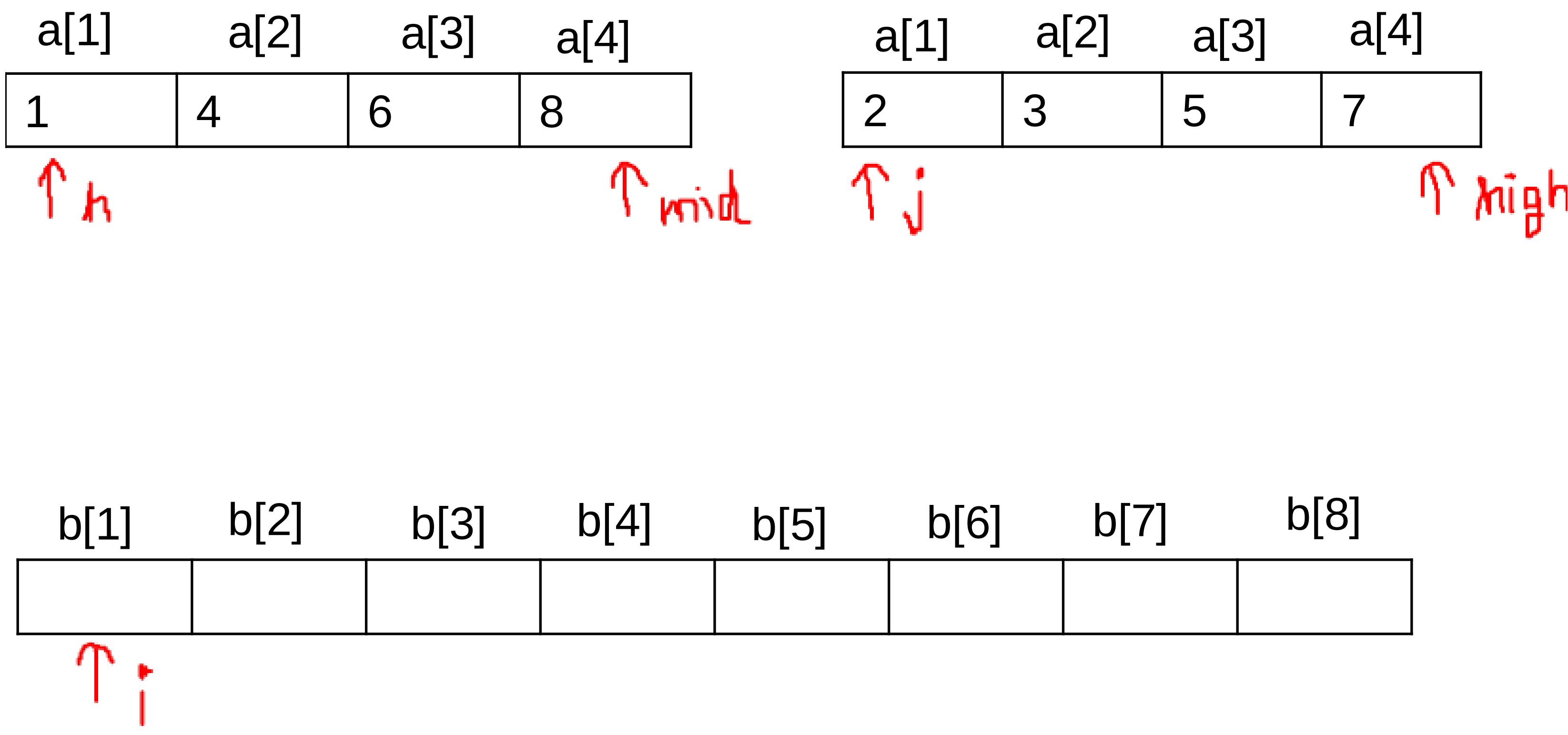


```
ALGORITHM  $Mergesort(A[0..n-1])$
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2\rfloor - 1]$ to $B[0..\lfloor n/2\rfloor - 1]$
        copy $A[\lfloor n/2\rfloor..n-1]$ to $C[0..\lceil n/2\rceil - 1]$
        $Mergesort(B[0..\lfloor n/2\rfloor - 1])$
        $Mergesort(C[0..\lceil n/2\rceil - 1])$
        $Merge(B, C, A)$    //see below
```

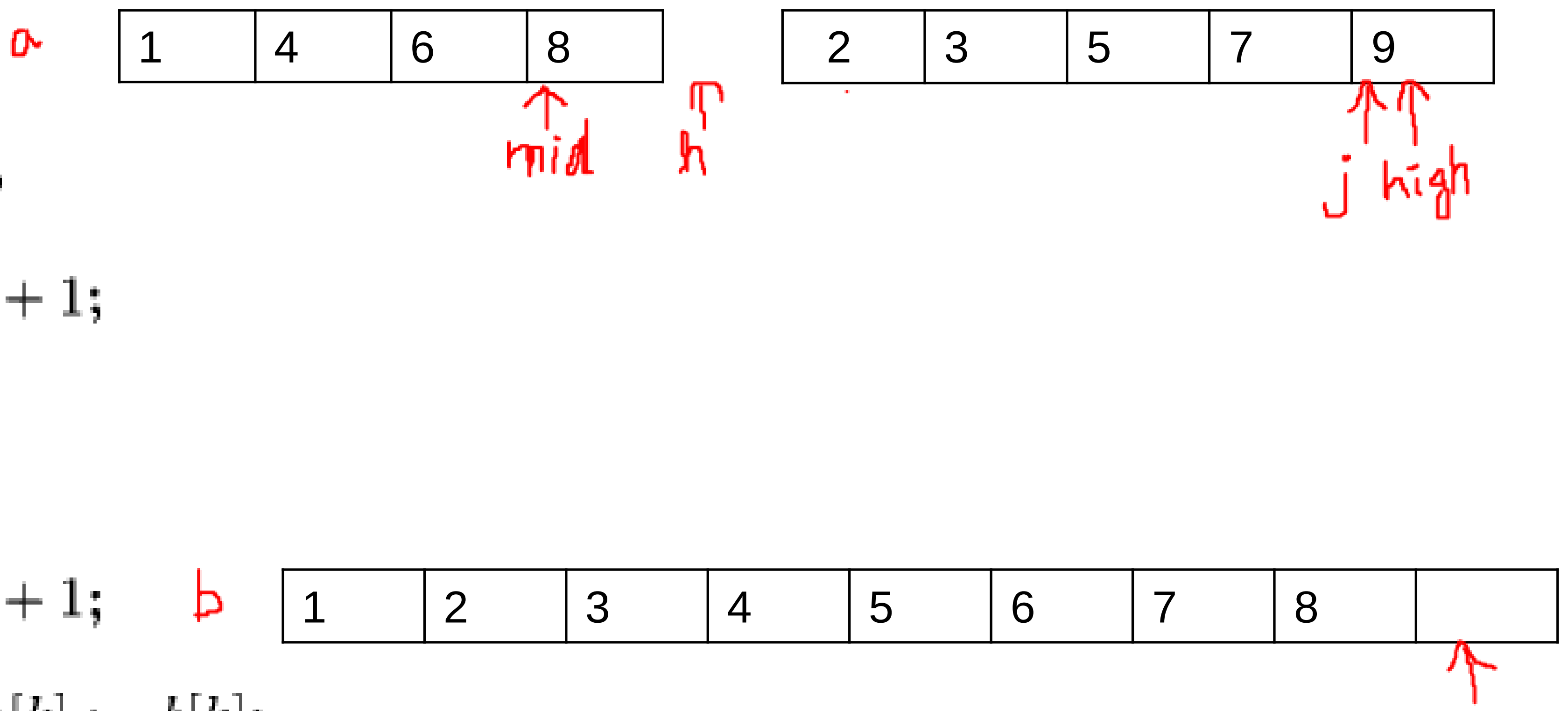
```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
    $h := low$; $i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



```
while (($h \le mid$) and ($j \le high$)) do
{
    if ($a[h] \le a[j]$) then
    {
        $b[i] := a[h]; h := h + 1;$
    }
    else
    {
        $b[i] := a[j]; j := j + 1;$
    }
    $i := i + 1;$
}
```



```
    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 $a[k] := b[k];$
}
```



whenever we divide a number into half in every step, it can be represented using a logarithmic function, which is log n and the number of steps can be represented by log n + 1(at most)

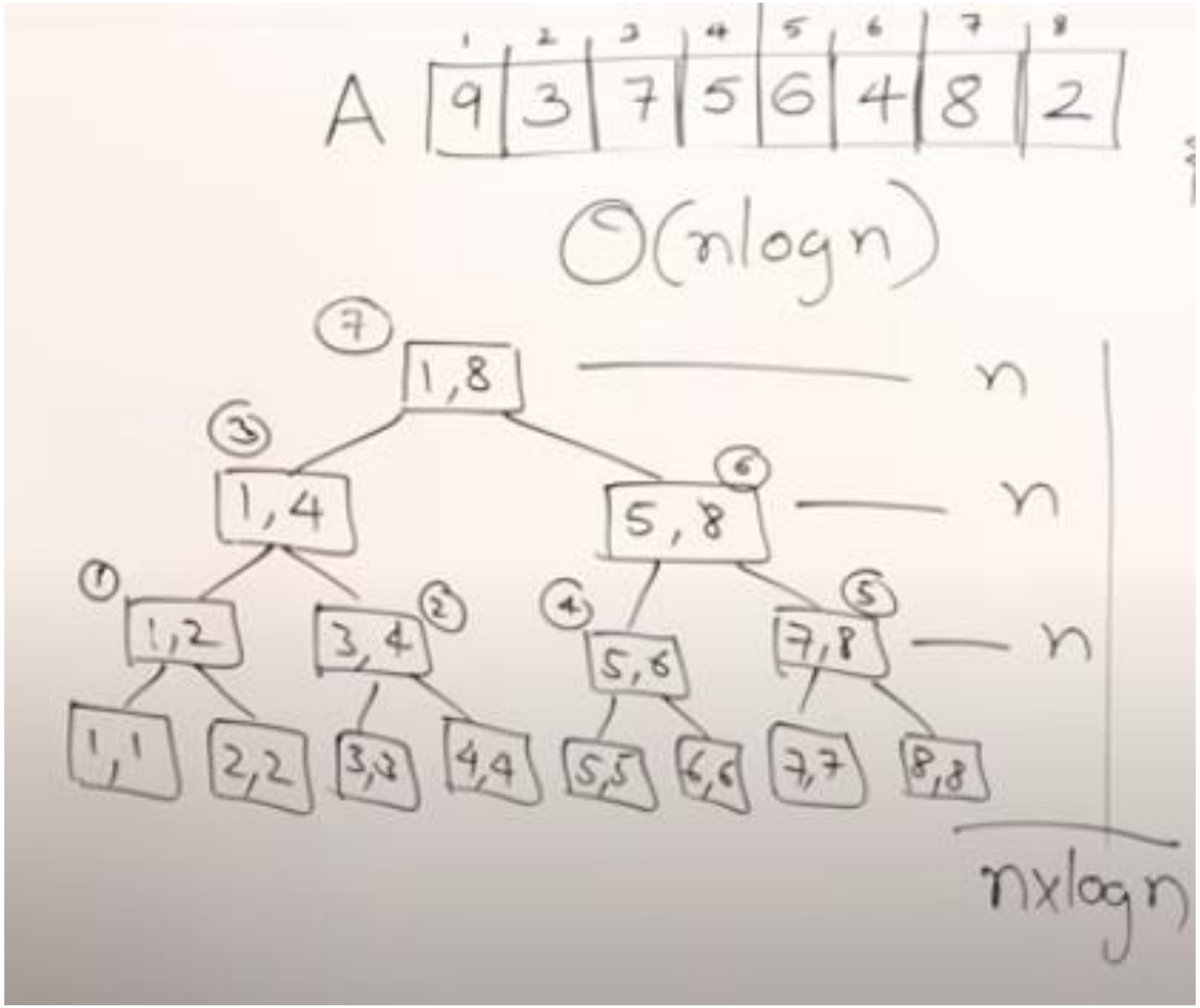
Also, we perform a single step operation to find out the middle of any subarray, i.e. O(1).

And to **merge** the subarrays, made by dividing the original array of n elements, a running time of O(n) will be required.

Hence the total time for mergeSort function will become n(log n + 1), which gives us a time complexity of O(n*log n).

Worst Case Time Complexity [Big-O]: **O(n*log n)**
Best Case Time Complexity [Big-omega]: **O(n*log n)**
Average Time Complexity [Big-theta]: **O(n*log n)**
Space Complexity: O(n)

Time Complexity of Merge Sort : ɵ(n log n)



Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

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

But we ignore '-1' because the element will take some time to be copied in merge lists.

$T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1 $\quad$ T(1)=0

Putting n=$\frac{n}{2}$ in place of n in ....equation 1

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$..................equation2

Put 2 equation in 1 equation

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$$

$T(n) = 2^2\, T\left(\frac{n}{2^2}\right)$+2n....................equation 3

Putting n= $\frac{n}{2^2}$ in equation 1

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$$ equation4

Putting 4 equation in 3 equation

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

$$T(n) = 2^3\ T\left(\frac{n}{2^3}\right) + n + 2n$$

$$T(n) = 2^3\ T\left(\frac{n}{2^3}\right) + 3n$$ equation5

From eq 1, eq3, eq 5.......we get

$$T(n) = 2^i\ T\left(\frac{n}{2^i}\right) + in$$ equation6

From Stopping Condition: $T(n) = 2^i\, T\left(\frac{n}{2^i}\right) + in.$

$\frac{n}{2^i} = 1$ And $T\left(\frac{n}{2^i}\right) = 0$

$n = 2^i$

Apply log both sides:

$\log n = \log_2 i$

$\log n = i \log 2$

$\frac{\log n}{\log 2} = i$

$\log_2 n = i$

From 6 equation

$T(n) = 2^i\, T\left(\frac{n}{2^i}\right) + in$

$= 2^i \times 0 + \log_2 n \,.\, n$

= T (n) = n.log n

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

QUICKSORT

The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.

In merge sort, the file a[1:n] was divided at its midpoint into subarrays which were independently sorted and later merged. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later.

- Quick Sort is a famous sorting algorithm.
- It sorts the given data items in ascending order.
- It uses the idea of divide and conquer approach.
- It follows a recursive algorithm.

How Does Quick Sort Works?

- Quick Sort follows a recursive algorithm.
- It divides the given array into two sections using a partitioning element called as pivot.

The division performed is such that-

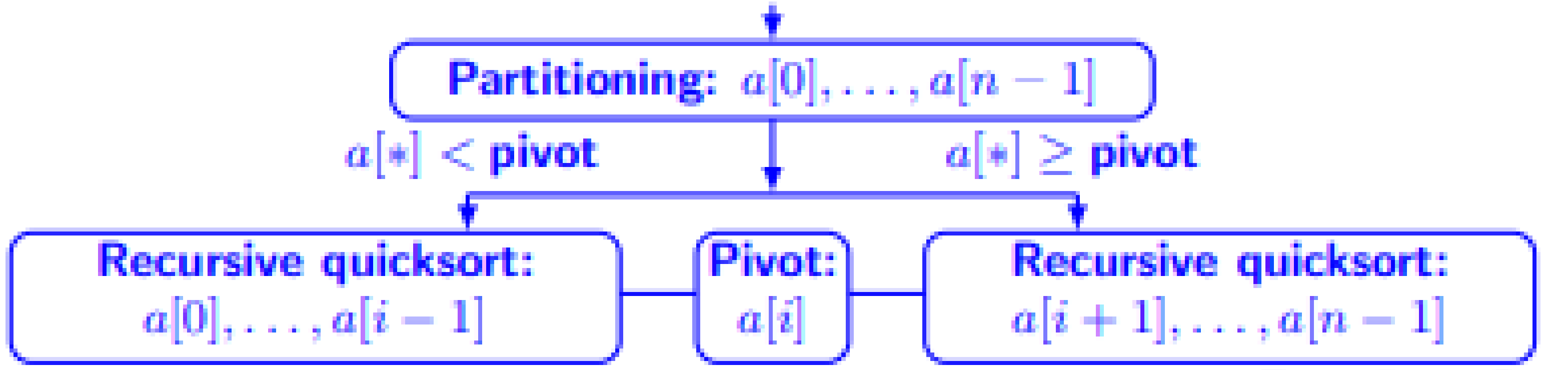
- All the elements to the left side of pivot are smaller than pivot.
- All the elements to the right side of pivot are greater than pivot.

After dividing the array into two sections, the pivot is set at its correct position.
Then, sub arrays are sorted separately by applying quick sort algorithm recursively.

Basic Recursive Quicksort

If the size, n, of the list, is 0 or 1, return the list. Otherwise:

1. Choose one of the items in the list as a **pivot**.
2. Next, **partition** the remaining items into two disjoint sublists, such that all items greater than the pivot follow it, and all elements less than the pivot precede it.
3. Finally, return the result of quicksort of the "head" sublist, followed by the pivot, followed by the result of quicksort of the "tail" sublist.

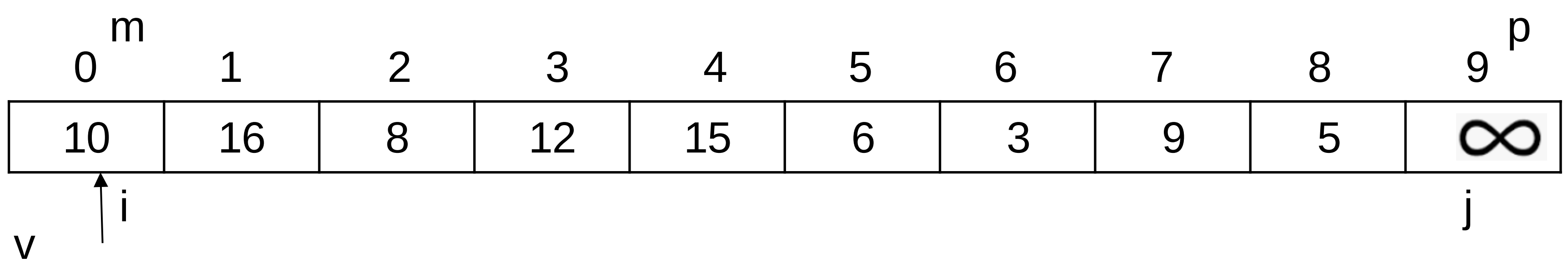

Partitioning: $a[0], \ldots, a[n-1]$
$a[*] < $ pivot
$a[*] \geq $ pivot
Recursive quicksort: $a[0], \ldots, a[i-1]$
Pivot: $a[i]$
Recursive quicksort: $a[i+1], \ldots, a[n-1]$

m									p
0	1	2	3	4	5	6	7	8	9
10	16	8	12	15	6	3	9	5	∞
v ↑									

Pivot = 10

m									p
0	1	2	3	4	5	6	7	8	9
10	16	8	12	15	6	3	9	5	∞
↑ i									j

Pivot = 10



Pivot = 10

Here,

i will search for the element which is greater than 10. ie Pivot

j will search for the element which is less than 10. ie Pivot

So that the numbers can be exchanged.

i atmost stop at infinity and j atmost stop at pivot

Partitioning Procedure: STEP - 1

	m									p
	0	1	2	3	4	5	6	7	8	9
	10	16	8	12	15	6	3	9	5	∞
Pivot = 10 v	↑	i							j	

Here,

increment i until we find the element which is greater than 10. ie Pivot

decrement j until we find the element which is less than 10. ie Pivot

Exchange 5 and 16.

	0	1	2	3	4	5	6	7	8	9
	10	5	8	12	15	6	3	9	16	∞
Pivot = 10	↑	i							j	

Partitioning Procedure: STEP - 2

m									p
0	1	2	3	4	5	6	7	8	9
10	5	8	12	15	6	3	9	16	∞
↑			i				j		

Pivot = 10 v

increment i, 8 is not greater than 10.
increment i,12 is greater than 10.
decrement j until we find the element which is less than 10. ie Pivot
9 is less than 10. ie pivot

Exchange 12 and 9.

0	1	2	3	4	5	6	7	8	9
10	5	8	9	15	6	3	12	16	∞
↑			i				j		

Pivot = 10

Partitioning Procedure: STEP - 3

	m									p
	0	1	2	3	4	5	6	7	8	9
	10	5	8	9	15	6	3	12	16	∞
Pivot = 10 v	↑				i		j			

increment i,15 is greater than 10

decrement j, 3 is less than 10. ie pivot

Exchange 15 and 3.

	0	1	2	3	4	5	6	7	8	9
	10	5	8	9	3	6	15	12	16	∞
Pivot = 10	↑				i		j			

Partitioning Procedure: STEP - 3

m									p
0	1	2	3	4	5	6	7	8	9
10	5	8	9	3	6	15	12	16	∞
↑					j	i			

Pivot = 10   v

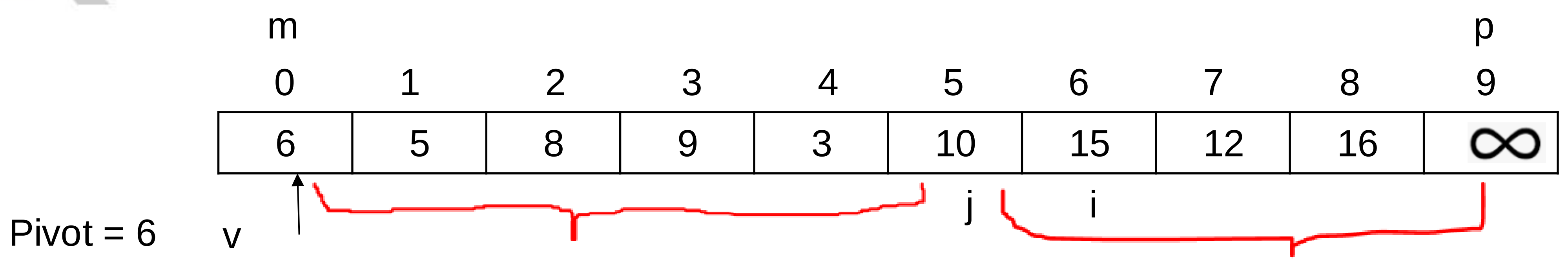
increment i,6 is not greater than 10
increment i,15 is greater than 10
decrement j, 6 is less than 10. ie pivot
Here, i> j, so don’t interchange i and j, interchange pivot and j

Interchange 10 and 6.

0	1	2	3	4	5	6	7	8	9
6	5	8	9	3	10	15	12	16	∞
↑					j	i			

Pivot = 6

Partitioning Procedure: STEP - 3



Here all the elements at the LHS are less than 10 and all the elements at the RHS are greater than 10. But both are not sorted.
j is the partitioning position.
Perform quick sort recursively on each side.

Data to sort; pivot $p = a[7] = 31$										**Description**
25	8	2	91	15	50	20	*31*	70	65	Initial list
										$L = 0; R = 10$
31	8	2	91	15	50	20	25	70	65	Move pivot to head
31	8	2	91	15	50	20	25	70	65	Stop R
31	8	2	91	15	50	20	25	70	65	Stop L
31	8	2	*25*	15	50	20	*91*	70	65	Swap $a[R]$ and $a[L]$
31	8	2	25	15	50	20	91	70	65	Stop R
31	8	2	25	15	50	20	91	70	65	Stop L
31	8	2	25	15	*20*	*50*	91	70	65	Swap $a[R]$ and $a[L]$
31	8	2	25	15	20	50	91	70	65	Stop $R = L$
20	8	2	25	15	*31*	50	91	70	65	Swap $a[L]$ with pivot

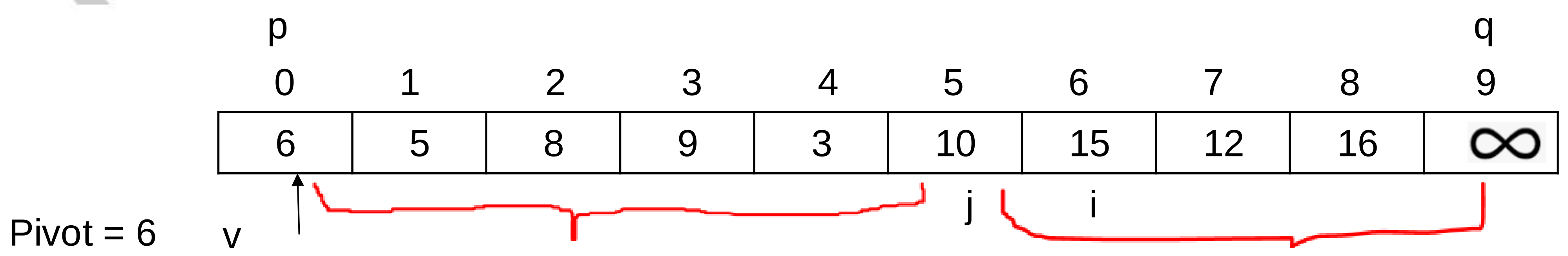
Head (left) sublist $\leq$ p $\leq$ Tail (right) sublist

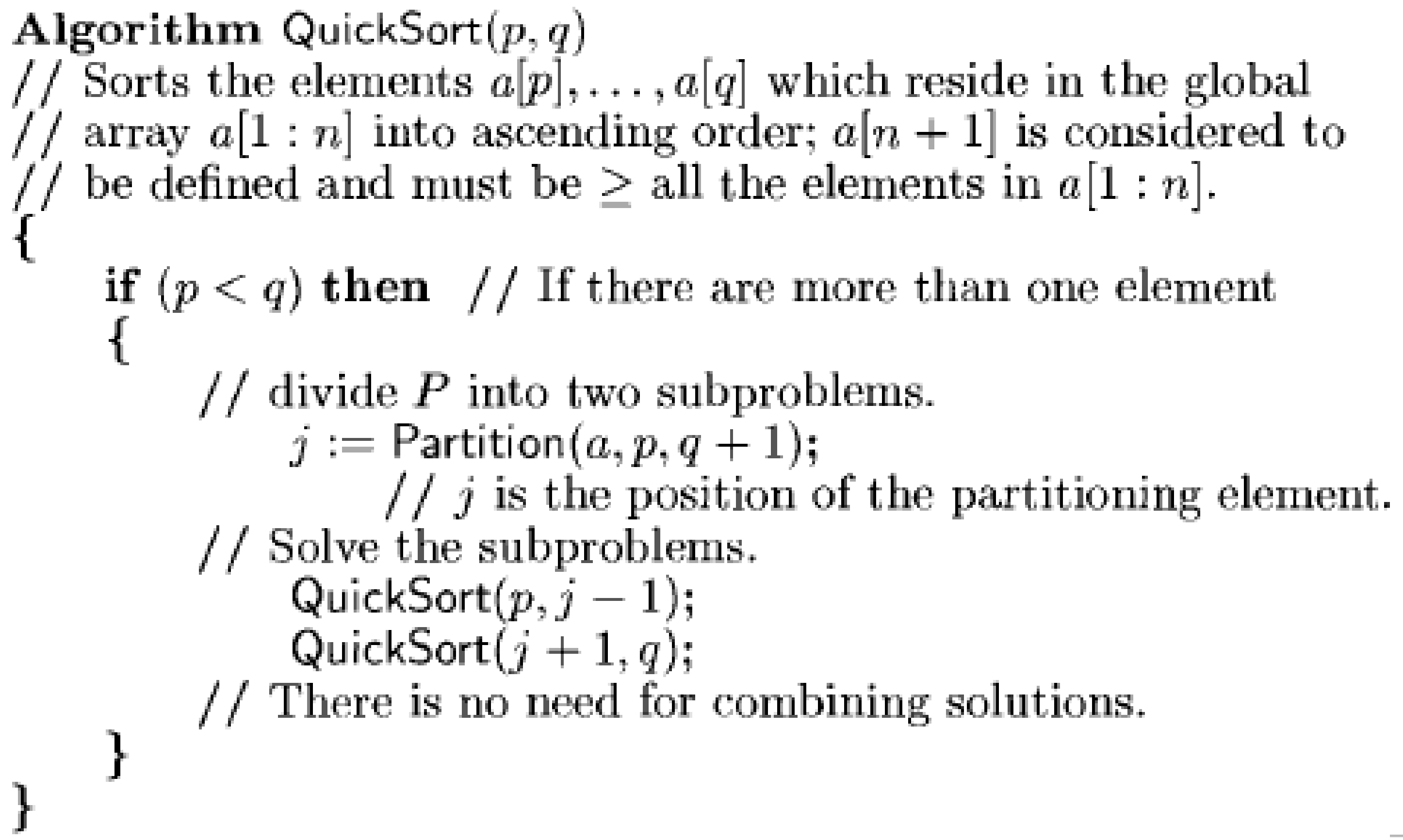
```
Algorithm Partition(a, m, p)
// Within a[m], a[m+1], ..., a[p-1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p-1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
  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

```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

MVJ COLLEGE OF ENGINEERING Since 1982
Engineering A Better Tomorrow
An Autonomous Institute

Partitioning Procedure: STEP - 3





```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., 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 ≥ 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.
    }
}
```

Worst Case-

- Quick Sort is sensitive to the order of input data.
- It gives the worst performance when elements are already in the ascending order.
- It then divides the array into sections of 1 and (n-1) elements in each call.
- Then, there are (n-1) divisions in all.
- Therefore, here total comparisons required are $f(n) = n \times (n-1) = O(n^2)$.

Worst Case Analysis:

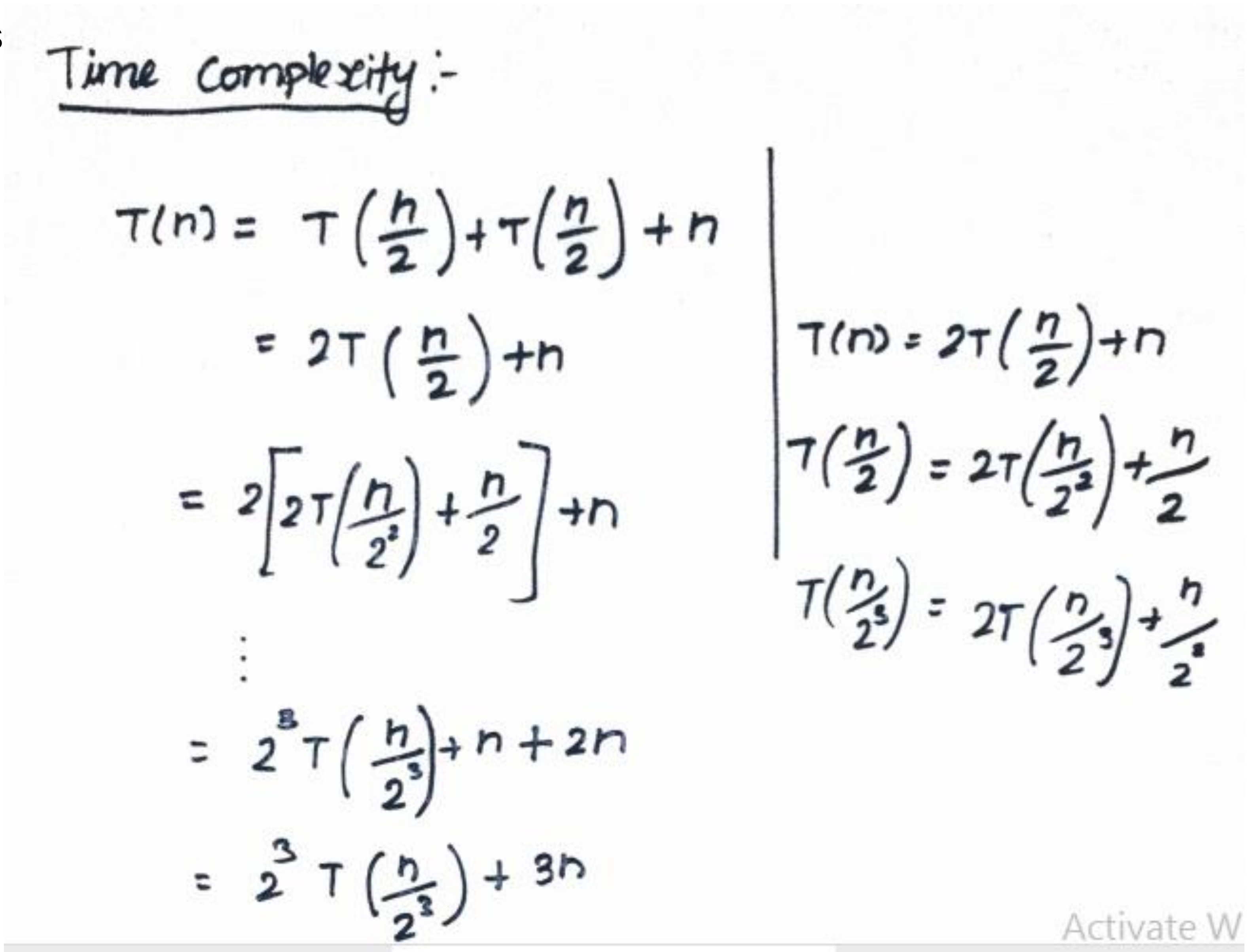
The pivot is the smallest element, all the time

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + C\,n && n > 1 \\
&= T(n-1) + C.\,n && \text{since } T(0)=0 \\
&= [T(n-2) + C(n-1)] + C\,n \\
&= [T(n-3) + C(n-2)] + C(n-1) + C\,n \\
&= T(n-3) + C[(n-2)+(n-1)+n] \\
&. \\
&. \\
&. \\
&= T(n-k) + C[(n-(k-1)) + \ldots. + (n-1) + n]
\end{aligned}
$$

Let $n-k=1 \Rightarrow$

$$
\begin{aligned}
T(n) &= T(1) + C[2+3+\ldots.+n] \\
&= 1 + C[2+3+\ldots.+n] \\
&\leq C[1+2+3+\ldots.n] \\
&\leq C.\,n(n+1)/2 \\
&= O(n^2)
\end{aligned}
$$

Average Case Analysis



Time complexity :-

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

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

$$= 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$\vdots$$

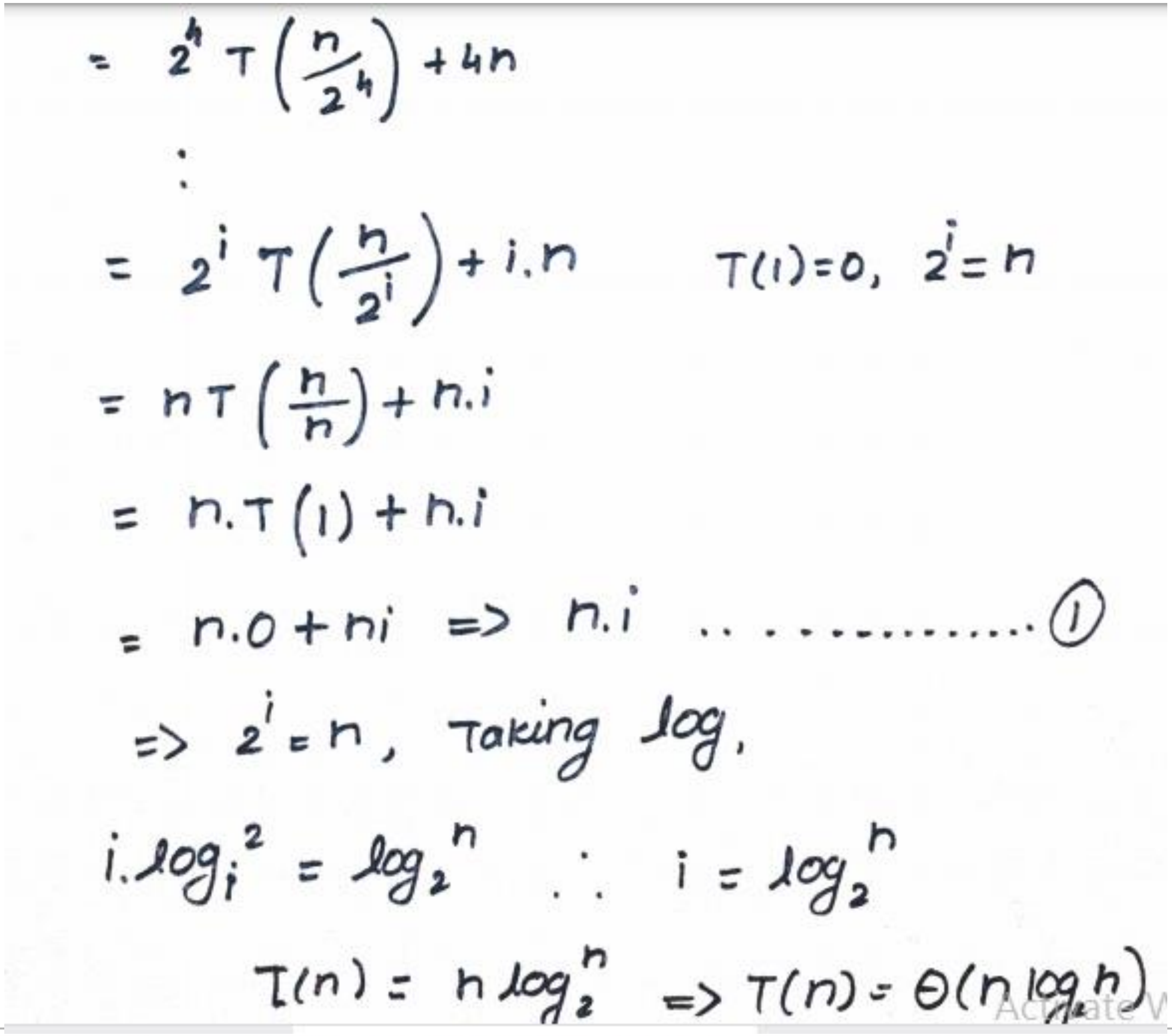
$$= 2^3 T\left(\frac{n}{2^3}\right) + n + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

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

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$$

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$$



$$= 2^4 T\left(\frac{n}{2^4}\right) + 4n$$

$$\vdots$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i.n \qquad T(1)=0,\ 2^i = n$$

$$= nT\left(\frac{n}{n}\right) + n.i$$

$$= n.T(1) + n.i$$

$$= n.0 + ni \Rightarrow n.i \ \ldots\ldots\ldots\ldots \textcircled{1}$$

$\Rightarrow 2^i = n$, Taking log,

$$i.\log_i^2 = \log_2^n \quad \therefore \quad i = \log_2^n$$

$$T(n) = n\log_2^n \Rightarrow T(n) = \Theta(n\log_2 n)$$

Worst Case Analysis:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(0) + T(n-1) + cn & \text{Otherwise} \end{cases}$$

where T(0) indicates that there are no elements towards left and it can be equated to zero. T(n-1) is the time required to sort remaining (n-1) elements and *cn* is the time required to partition the array into two sub arrays. Now, consider the relation

$$T(n) = T(0) + T(n-1) + cn$$

$$= T(n-1) + cn \qquad \text{By replacing n by n-1 in above eq.}$$

$$= T(n-2) + c(n-1) + cn \qquad \text{By replacing n-1 by n-2 in above eq.}$$

$$= T(n-3) + c(n-2) + c(n-1) + cn$$

....

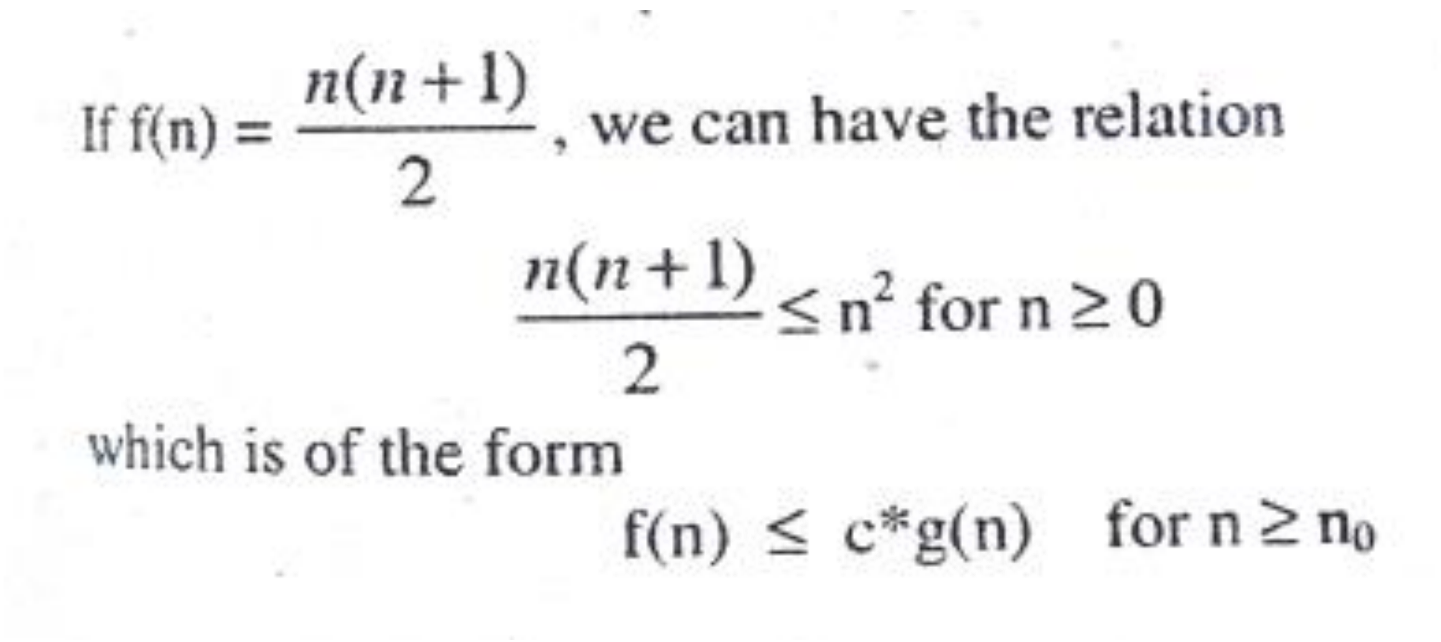
....

....

$$= T(n-n) + c(1) + c(2) + \ldots.. + c(n-2) + c(n-1) + cn$$

$$= 0 + c(1) + c(2) + \ldots.. + c(n-2) + c(n-1) + cn$$

$$= c\left(\frac{n(n+1)}{2}\right)$$



If $f(n) = \frac{n(n+1)}{2}$, we can have the relation

$$\frac{n(n+1)}{2} \leq n^2 \text{ for } n \geq 0$$

which is of the form

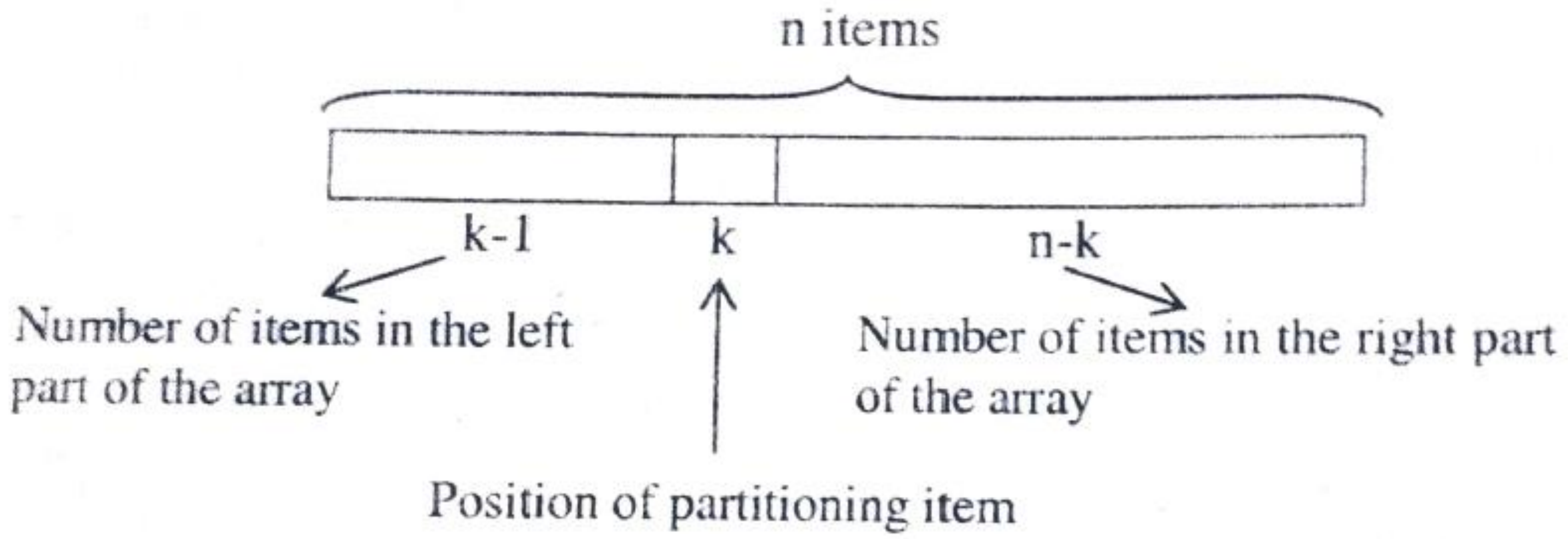
$$f(n) \leq c^*g(n) \quad \text{for } n \geq n_0$$

By definition, C=1, n_0=0,g(n)=n^2

So, T(n)= o(n^2)

Analysis (Average case)

In the average case, the given array may not be exactly partitioned into two sub arrays or may not be skewed as in case of worst case. The *key element*(also called pivot element) may be placed at any arbitrary position in the array ranging from 1 to n. Let us consider the situation shown below which gives *k* as the position of the partitioning element:


n items
k-1
k
n-k
Number of items in the left part of the array
Number of items in the right part of the array
Position of partitioning item

$$= \frac{a\left[n + (n-1) + (n-2) + \ldots\ldots 3 + 2 + 1\right]}{n} = \frac{an(n+1)}{2n} = c(n+1) \text{ where } c = a/2$$

The time complexity for the average case is obtained by taking the time to partition the array and time taken to sort the left part and the right part and is given by the following relation:

$$\therefore, T(n) = \underbrace{c(n+1)}_{\text{Time for average number of comparisons required for partitioning}} + \frac{1}{n}\sum_{k=1}^{n} \underbrace{T(k-1) + T(n-k)}_{\text{Average time required to sort the left part of the array and right part of the array.}}$$

Let us also assume T(0) = 0 and T(1) = 0. Now, let us solve the relation

$$T(n) = c(n+1) + \frac{1}{n}\sum_{k=1}^{n} T(k-1) + T(n-k) \text{---Eq 4.1}$$

Multiplying equation 4.1 by n we get

$$nT(n) \quad = cn(n+1) + \sum_{k=1}^{n}\left(T(k-1) + T(n-k)\right)$$

$$= cn(n+1) + T(0) + T(n-1) + T(1) + T(n-2) + \ldots\ldots T(n-2) + T(1) + T(n-1) + T(0)$$

$$= cn(n+1) + 2[T(0) + T(1) + T(2) + \ldots\ldots T(n-1)] \text{----------------------------Eq 4.2}$$

Replacing n by n-1 in equation 4.2, we have

$$(n-1)T(n-1) = c(n-1)n + 2[T(0) + T(1) + T(2) + \ldots\ldots T(n-2) \text{--------------------------Eq 4.3}$$

Now, subtracting equation 4.3 from equation 4.2 we have

$$nT(n) - (n-1)T(n-1) \ = cn^2 + cn - cn^2 + cn + \ 2T(n-1)$$

Now, subtracting equation 4.3 from equation 4.2 we have

$$nT(n) - (n-1)T(n-1) = cn^2 + cn - cn^2 + cn + 2T(n-1)$$
$$= 2cn + 2T(n-1)$$
$$\text{i.e.. } nT(n) = (n-1)T(n-1) + 2cn + 2T(n-1)$$
$$= T(n-1)(n-1+2) + 2cn$$
$$= T(n-1)(n+1) + 2cn$$

Dividing the above equation by n(n+1) we have

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

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

Replacing n by n-1 we have

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

Substituting for $\frac{T(n-1)}{n}$ in the above equation we have

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

Substituting for $\frac{T(n-2)}{n-1}$ in the above equation we have

Replacing n by n-1 we have

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

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

$$= \frac{T(n-4)}{n-3} + \frac{2c}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1}$$

....

....

$$= \frac{T(0)}{1} + \frac{2c}{2} + \frac{2c}{3} + \ldots.. + \frac{2c}{n} + \frac{2c}{n+1}$$

....

$$= \frac{T(0)}{1} + \frac{2c}{2} + \frac{2c}{3} + \ldots\ldots + \frac{2c}{n} + \frac{2c}{n+1}$$

$$= \frac{2c}{2} + \frac{2c}{3} + \ldots\ldots + \frac{2c}{n} + \frac{2c}{n+1} = 2c\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}\right)$$

$$= 2c\sum_{k=2}^{n+1}\frac{1}{k} = 2c\int_{2}^{n+1}\frac{1}{k}dk = 2c\left[\log k\right]_2^{n+1}$$

$$= 2c[\log_2(n+1) - \log_2 2] \approx 2c\log_2(n+1) \text{ for very large value of } n$$

i.e. $\frac{T(n)}{(n+1)} = 2c\log_2(n+1)$

$\therefore$, $T(n) = (n+1)\log_2(n+1) \approx n\log_2 n$ for very large value of n.

$\therefore$, Time complexity of quick sort in the average case is given by $T(n) = \theta(n\log_2 n)$

Matrix multiplication: Strassen’s algorithm

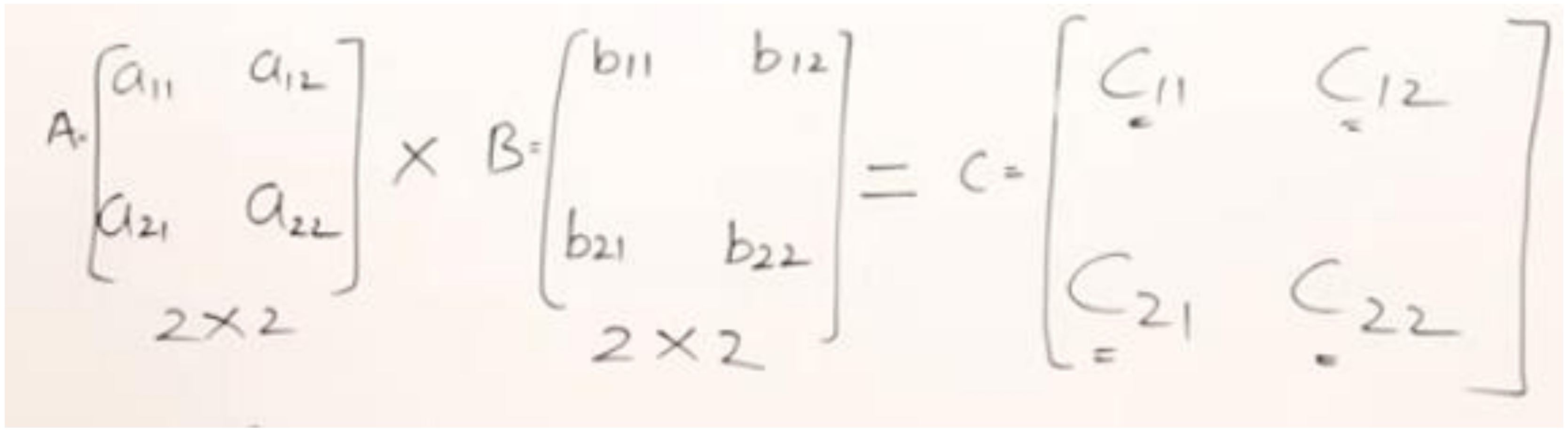
Matrix Multiplication

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 5 \\ 3 & 7 \end{bmatrix} = \begin{bmatrix} 3+12 & 15+28 \\ 2+3 & 10+7 \end{bmatrix}$$

Matrix 1 Matrix 2

$$= \begin{bmatrix} 15 & 43 \\ 5 & 17 \end{bmatrix}$$

Resultant Matrix



Multiplication can be performed by 4 equations:

$c_{11} = a_{11}b_{11} + a_{12}b_{21}$

$c_{11} = a_{11}b_{12} + a_{12}b_{22}$

$c_{21} = a_{21}b_{11} + a_{22}b_{21}$

$c_{22} = a_{21}b_{12} + a_{22}b_{22}$

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity = o(n^3)

Consider two matrices A and B with 4x4 dimension each as shown below,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \qquad \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

The matrix multiplication of the above two matrices A and B is Matrix C,

where,
c11=a11∗b11+a12∗b21+a13∗b31+a14∗b41
c12=a11∗b12+a12∗b22+a13∗b32+a14∗b42
c21=a21∗b11+a22∗b21+a23∗b31+a24∗b41
c22=a21∗b12+a22∗b22+a23∗b32+a24∗b42
.
.
and so on.

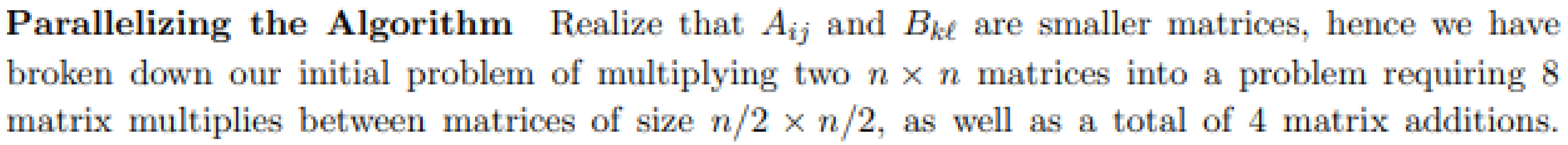
The divide and conquer paradigm is important general technique for designing algorithms.

In general, it follows the steps:

- divide the problem into sub problems

- recursively solve the sub problems

- combine solutions to sub problems to get solution to original problem



Parallelizing the Algorithm Realize that A_{ij} and $B_{k\ell}$ are smaller matrices, hence we have broken down our initial problem of multiplying two $n \times n$ matrices into a problem requiring 8 matrix multiplies between matrices of size $n/2 \times n/2$, as well as a total of 4 matrix additions.

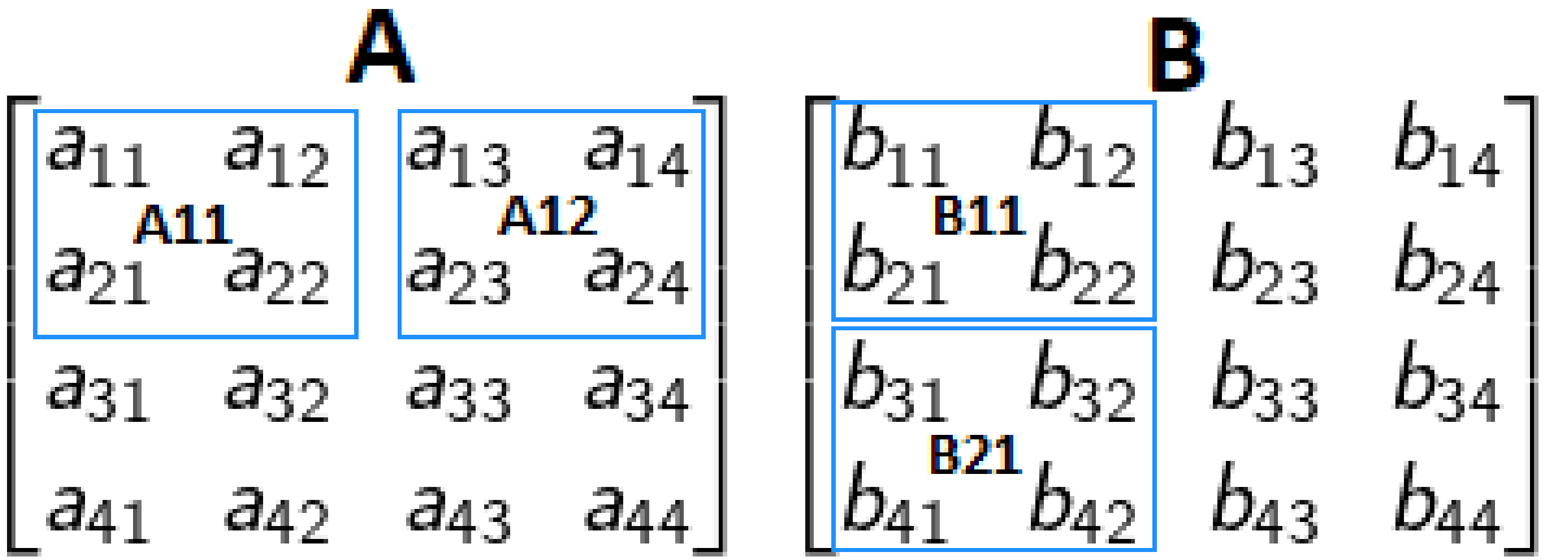
$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

A11 A12 A_{21} A_{22}

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

B11 B_{12} B21 B_{22}

Now, let's look at the Divide and Conquer approach to multiply two matrices.
Take two submatrices from the above two matrices A and B each as



$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

And the matrix multiplication of the two 2x2 matrices A11 and B11 is,

$$\underset{\textbf{A11}}{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}} * \underset{\textbf{B11}}{\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}} = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}$$

Also, the matrix multiplication of two 2x2 matrices A12 and B21 is as follows,

$$\underset{\textbf{A12}}{\begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix}} * \underset{\textbf{B21}}{\begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix}} = \begin{bmatrix} a_{13} * b_{31} + a_{14} * b_{41} & a_{13} * b_{32} + a_{14} * b_{42} \\ a_{23} * b_{32} + a_{24} * b_{42} & a_{23} * b_{32} + a_{24} * b_{42} \end{bmatrix}$$

We write A and B as block matrices,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

where block matrices Aij are of size n/2×n/2 (same with respect to block entries of B and C). Trivially, we may apply the definition of block-matrix multiplication to write down a formula for the block-entries of C, i.e.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

```
Algorithm MM(A,B,n)
{
if(n<=2)
{  C=4 formula;
($c_{11} = a_{11}b_{11} + a_{12}b_{21}$
$c_{11} = a_{11}b_{12} + a_{12}b_{22}$
$c_{21} = a_{21}b_{11} + a_{22}b_{21}$
$c_{22} = a_{21}b_{12} + a_{22}b_{22}$)  }
else
{
mid=n/2;
MM($A_{11}$,$B_{11}$,n/2) +MM($A_{12}$,$B_{21}$,n/2)
MM($A_{11}$,$B_{12}$,n/2) +MM($A_{12}$,$B_{22}$,n/2)
MM($A_{21}$,$B_{11}$,n/2) +MM($A_{22}$,$B_{21}$,n/2)
MM($A_{21}$,$B_{12}$,n/2) +MM($A_{22}$,$B_{22}$,n/2)
}
```

For n>2 the elements of C can be computed Using matrix multiplication and addition operations applied to matrices of Size n/2 x n/2.Sincen is a power of 2,these matrix products can be recursively computed by the same algorithm we are using for the nxn case. This Algorithm will continue applying itself to smaller-sized sub matrices until n Becomes suitably small(n = 2) so that the product is computed directly

To compute AB, We need to perform eight multiplications of n/2 x n/2matrices and four additions of n/2x n/2matrices.

Since two n/2 x n/2 matrices can be added in time cn^2 for some constant c, the overall Computing time T(n) of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

$a=8$, $b=2$, $f(n)=n^2$

$n^k = n^2$

$\text{Log}_2 8 = 3 \quad \Rightarrow \quad n^{\text{Log}_2 8} = n^3$

Time Complexity $= \theta(n^3)$

Theorem 1 *The recurrence*

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \text{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}$$

Strassen in 1969 which gives an overview that how we can find the multiplication of two **2*2 dimension matrix by the brute-force algorithm**.

But by using divide and conquer technique the overall complexity for multiplication two matrices is reduced. This happens by decreasing the total number if multiplication performed at the expenses of a slight increase in the number of addition.

For multiplying the two 2*2 dimension matrices **Strassen's** used some formulas in which there are seven multiplication and eighteen addition, subtraction, and in brute force algorithm, there is eight multiplication and four addition.

Strassen’s Algorithm is an algorithm for matrix multiplication. It is faster than the naive matrix multiplication algorithm. The idea behind Strassen’s algorithm is in the formulation of matrix multiplication as a recursive problem.

Pseudocode

1. Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2. Calculate the 7 matrix multiplications recursively.
3. Compute the sub matricies of C.
4. Combine these sub matricies into our new matrix C

A

$$\left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ & \textbf{A11} & & \textbf{A12} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ & A_{21} & & A_{22} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right]$$

B

$$\left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ & \textbf{B11} & & B_{12} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ & \textbf{B21} & & B_{22} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right]$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B - 11)$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

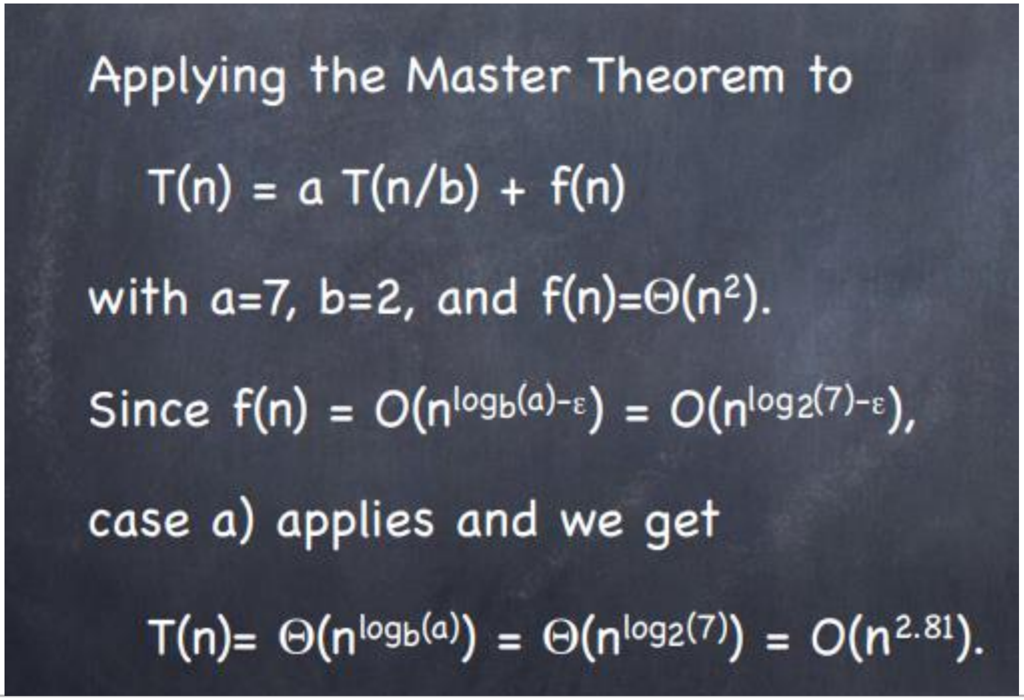
$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

From the above equations, the recurrence relation of the Strassen's approach is,

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 7T(\frac{n}{2}) + \Theta(n^2), & \text{if } n > 1 \end{cases}$$

So, from Case 1 of Master's Theorem, the time complexity of the above approach is $O(n^{\log_2 7})$ or $O(n^{2.81})$ which beats the divide and conquer approach asymptotically.


Applying the Master Theorem to

$T(n) = a\ T(n/b) + f(n)$

with $a=7$, $b=2$, and $f(n)=\Theta(n^2)$.

Since $f(n) = O(n^{\log_b(a)-\varepsilon}) = O(n^{\log_2(7)-\varepsilon})$,

case a) applies and we get

$T(n)= \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(7)}) = O(n^{2.81})$.

Advantages of Divide and Conquer

Divide and conquer is a powerful tool for solving conceptually difficult problems.

• Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle, divide & conquer .

•It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.

• It efficiently uses cache memory without occupying much space because it solves simple sub problems within the cache memory instead of accessing the slower main memory.

- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Results in efficient algorithms

- Divide & Conquer algorithms are adapted for execution in multi-processor machines
- Results in algorithms that use memory cache efficiently

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.

- An explicit stack may overuse the space.

It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU

.

- Recursion is slow

Another concern with it is the fact that sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n.
In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.

Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times

1. Which one of the following is the recurrence equation for the worst case time complexity of the Quicksort algorithm for sorting $n(\geq 2)$ numbers? In the recurrence equations given in the options below, c is a constant.

A. $T(n) = 2T(n/2) + cn$

B. $T(n) = T(n-1) + T(0) + cn$

C. $T(n) = 2T(n-2) + cn$

D. $T(n) = T(n/2) + cn$

Answer B

2. You have an array of n elements. Suppose you implement quicksort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is

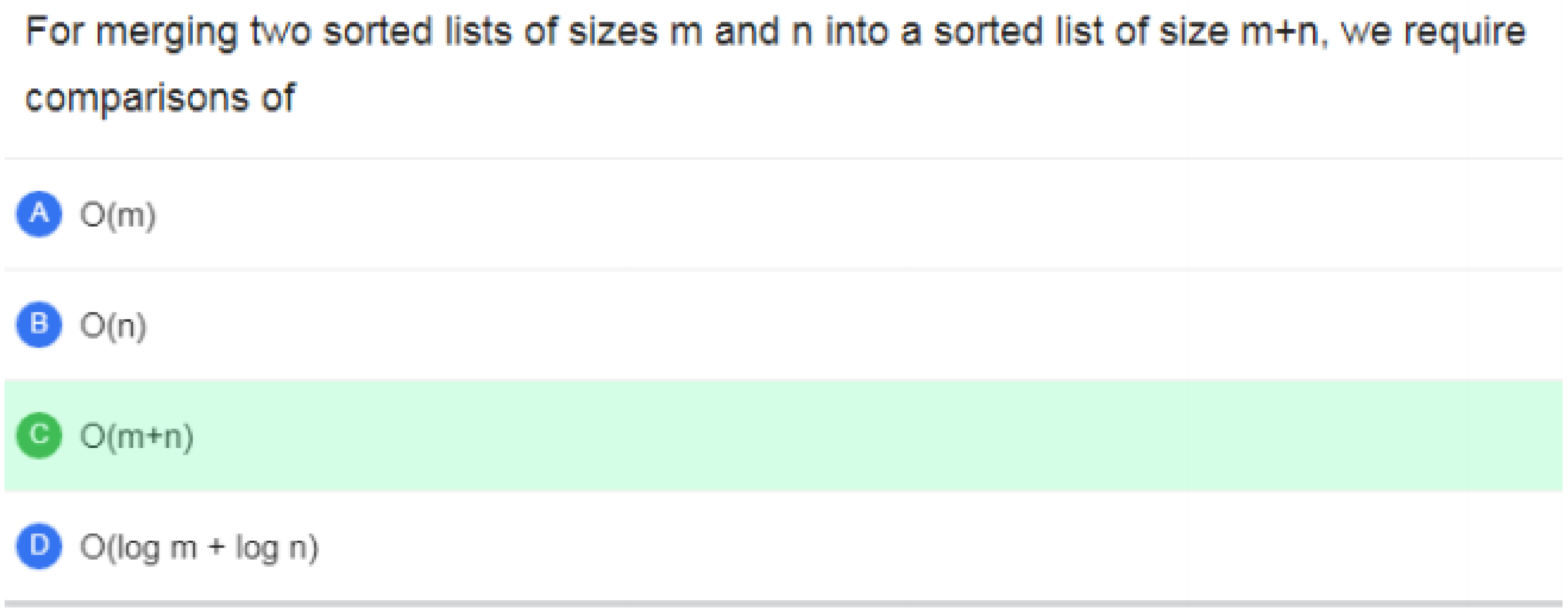
A. $O(n^2)$

B. O(nLogn)

C. Theta(nLogn)

D. $O(n^3)$

Answer A



For merging two sorted lists of sizes m and n into a sorted list of size m+n, we require comparisons of

- A O(m)
- B O(n)
- [✓ C O(m+n)]
- D O(log m + log n)