

Let $A[1..n]$ be a sequence of n elements. Consider the problem of determining whether a given element x is in A . The problem can be rephrased as follows:

A straightforward approach is to scan the entries in A and compare each entry with x. If after j comparisons, $1 \leq j \leq n$, the search is successful, i.e. $x=A[j]$, j is returned, otherwise a value of 0 is returned indicating an *unsuccessful* search. This method is referred to as *sequential search*. It is also called linear search because the maximum number of element comparison grows linearly with the size of the sequence. The algorithm is shown as Algorithm LINEARSEARCH.

Output: j if $x=A[j]$, $1 \leq j \leq n$, and 0 otherwise.

```

1.j=1
2.while  (j<n)and  (x≠A[j])
3.      j=j+1
4.end while
5.if x==A[j] the return j else return 0

```

Suppose the following inputs are supply to Linear Search

$J=1$, while $j_1 < n^{\text{yes}}$ and $x \neq A[j]^{\text{yes}} \rightarrow^{\text{yes}}$

$j=j+1=2$ $A =$

1	2	3	4
10	26	50	100

 j

while $j_2 < n^{\text{yes}}$ and $x \neq A[j]^{\text{yes}} \rightarrow \text{yes}$

$j=j+1=3$ $A =$

1	2	3
10	26	50

 j

while $j_3 < n^{yes}$ and $x \neq A[j]^{yes} \rightarrow^{yes}$

		1	2	3	4
$j=j+1=4$	A =	10	26	50	100
					J

while $j_4 < n^{no}$ and $x \neq A[j]^{no} \rightarrow^{no}$, So goto line 5

Is $x == A[j]$? No \rightarrow return 0

BINARY SEARCH

For LINEARSEARCH, scanning all entries of A is inevitable if no more information about the ordering of the elements in A is given. If we are also given that the elements in A are sorted, say in nondecreasing order, then there is a much more efficient algorithm. The following example illustrates this efficient search method

Example 1.1: consider searching the array

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$A[1..14] =$	1	4	5	7	8	9	10	12	15	22	23	27	32	35
	↑													↑
	L													H

Let us suppose we want to search for $x=22$. First, we compare x with the middle element $A[(1+14)/2] = A[7] = 10$. Since $22 > A[7]$, it means x cannot be in $A[1..7]$, and therefore, this portion of the array can be discarded. So we are left with the sub array:

	8	9	10	11	12	13	14
$A[8..14] =$	12	15	22	23	27	32	35
	↑						↑
	L						H

Secondly, x is compared with the middle of the remaining elements $A[(8+14)/2] = A[11] = 23$. Since $22 < A[11]$, it means x cannot be in $A[11..14]$, and therefore this portion of the array can also be discarded. Thus, the remaining portion of the array to be searched is now reduced to

	8	9	10
$A[8..10] =$	12	15	22
	↑		↑
	L		H

Next, we compare x with the middle element $A[9]=15$. Since $22>15$, x cannot be in $A[8..9]$, therefore $A[8..9]$ is discarded. This leaves only one entry in array to be searched i.e,

$$A[10]= \begin{array}{c} 10 \\ \boxed{22} \\ \uparrow \\ L\ H \end{array}$$

Finally, we find that $x= A[10]$, and the search is successfully completed.

In general, let $A[low..high]$ be an array of elements sorted in nondecreasing order. Let $A[mid]$ be the middle element, and suppose that $x>A[mid]$. If x is in A , then it must be one of the elements $A[mid+1], A[mid+2], \dots, A[high]$. It follows that we only need to search for x in $A[mid+1..high]$. In other words, the entries in $A[low..high]$ are discarded in subsequent comparisons since, by assumption, A is sorted in nondecreasing order, which implies that x cannot be in this half of array. Similarly, if $x<A[mid]$, then we only need to search for x in $A[low..mid-1]$. This results in an efficient strategy which, because of its repetitive halving, is referred to as *binary search*. Algorithm BINARYSEARCH gives a more formal description of this method.

Algorithm: BINARYSEARCH

Input: A sorted array $A[1..n]$ of n elements and an element x

Output: j , if $x= A[j]$, $1 \leq j \leq n$, and 0 otherwise.

```

1. low=1; high=n; j=0
2. while (low<=high) and (j==0)
3.     mid = ⌊(low+high)/2⌋
4.     If x == A[mid] then j=mid
5.     else if x < A[mid] then high=mid-1
6.     else low= mid+1
7. end while
8. return j

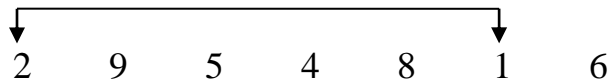
```

SORTING

Selection Sort

Let $A[1..n]$ be an array of n elements and suppose that we want to sort A in ascending order, first, the algorithm finds the smallest element in A and swap it with the first element $A[1]$. It then finds the smallest in the remaining $n-1$ elements and swaps it with the second element $A[2]$. This process of selecting the smallest element and swapping it with another element continue until a single number remain. For example, let us show how to sort the list: 2, 9, 5, 4, 8, 1, 6 using selection sort.

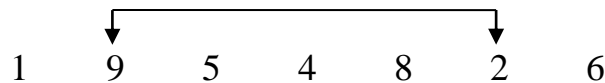
- * Select 1 (the smallest) and swap it with 2 (the first element in the list).



The number 1 is now in the correct position and thus, no longer need to be considered.

1 9 5 4 8 2 6

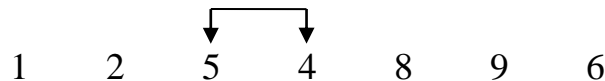
- * Select 2 (the smallest) and swap it with 9 (the first number) in the remaining list of numbers.



The number 2 is now in the right position and thus, no need to be considered.

1	2	5	4	8	9	6
---	---	---	---	---	---	---

- * Select 4 (the smallest) and swap it with 5 (first number) in the remaining list of numbers.



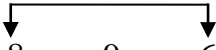
The number 4 is now in the right position and thus, no need to be considered.

1	2	4	5	8	9	6
---	---	---	---	---	---	---

Considering the above sorted list, element 5 is in its right position and need not to be considered.

- * Select 6 (the smallest) and swap it with 8 (first number) in the remaining list of numbers.

1 2 4 5 8 9 6

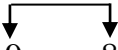


The number 6 is now in the right position and thus, no need to be considered.

1	2	4	5	6	9	8
---	---	---	---	---	---	---

- * Select 8 (the smallest) and swap it with 9 (first number) in the remaining list of numbers.

1 2 4 5 6 9 8



The number 8 is now in the right position and thus, no need to be considered.

1	2	4	5	6	8	9
---	---	---	---	---	---	---

Since, there is only one element remaining in the list, the sorting is completed.
This method is described in Algorithm SELECTIONSORT.

```

Algorithm: Selection sort
Input: An array A[1...n] of n elements
Output: A sorted in ascending order

1  for i = 1 to n-1
2    k=i
3    for j = i+1 to n
4      if A[j] < A[k] then k = j
5    end for
6    if K ≠ i then interchange A[i]and A[k]
7  end for

```

Example 1

Illustrate the operations of selection ALGORITHM using the following array.

	1	2	3	4	5	6	7
A	2	9	5	4	8	1	6

i=1, k=i=1, j=2, is $A[2] < A[1]$? No
j=3, is $A[3] < A[1]$? No
j=4, is $A[4] < A[1]$? No
j=5, is $A[5] < A[1]$? No
j=6, is $A[6] < A[1]$? yes $\Rightarrow k=j=6$
j=7, is $A[7] < A[6]$? NO
j=8,

is $k \neq i$? yes \Rightarrow swap $A[i_1]$ with $A[k_6]$, this gives

	1	2	3	4	5	6	7
A	1	9	5	4	8	2	6

i=2, k=i=2, j=3, is $A[3] < A[2]$? yes $\rightarrow k=j=3$
j=4, is $A[4] < A[3]$? yes $\rightarrow k=j=4$
j=5, is $A[5] < A[4]$? No
j=6, is $A[6] < A[4]$? yes $\Rightarrow k=j=6$
j=7, is $A[7] < A[6]$? No
j=8.

is $k \neq i$? yes \Rightarrow swap $A[i_2]$ with $A[k_6]$, this gives

	1	2	3	4	5	6	7
A	1	2	5	4	8	9	6

i=3, k=i=3, j=4, is $A[4] < A[3]$? yes $\Rightarrow k=j=4$
j=5, is $A[5] < A[4]$? No
j=6, is $A[6] < A[4]$? No
j=7, is $A[7] < A[4]$? No
j=8,

is $k \neq i$? yes \Rightarrow swap $A[i_3]$ with $A[k_4]$. This gives

	1	2	3	4	5	6	7
A	1	2	4	5	8	9	6

i=4, k=i=4, j=5, is $A[5] < A[4]$? No
j=6, is $A[6] < A[4]$? No
j=7, is $A[7] < A[4]$? No

is $k \neq i$? No \Rightarrow No swapping. This gives

	1	2	3	4	5	6	7
A	1	2	4	5	8	9	6

i = 5, k=i=5, j=6, is $A[6] < A[5]$? No
j=7, is $A[7] < A[5]$? yes $\Rightarrow k=j=7$

j=8.

is $k \neq i$? yes \Rightarrow swap $A[i_5]$ with $A[k_7]$. This gives

	1	2	3	4	5	6	7
A	1	2	4	5	6	9	8

i=6, $k = i = 6$, j=7, is $A[7] < A[6]$? yes $\Rightarrow k=j=7$

j=8

is $k \neq i$? yes \Rightarrow swap $A[i_6]$ with $A[k_7]$. This gives

	1	2	3	4	5	6	7
A	1	2	4	5	6	8	9

One of the disadvantages of this algorithm is that, it doesn't mind the arrangement of elements in the array.

INSERTION SORT

This is a sorting method in which the number of comparisons depends on the order of the input elements. The algorithm is shown below and it works at follows:

We begin with the sub array of size 1, $A[1]$, which is already sorted. Next, $A[2]$ is inserted before or after $A[1]$ depending on whether it is smaller than $A[1]$ or not. Continuing this way, in the i^{th} iteration, $A[i]$ is inserted in its proper position in the sorted sub array $A[1 \dots i-1]$. This is done by scanning the elements from index $i-1$ down to 1, each time comparing $A[i]$ with the element at the current position. In each iteration of the scan, an element is shifted one position up to a higher index. This process of scanning, performing the comparison and shifting continues until an element less than or equal to $A[i]$ is found, or when all the sorted sequence so far is exhausted. At this point, $A[i]$ is inserted in its proper position, and the process of inserting element $A[i]$ in its proper place is complete. The following example demonstrates how the algorithm works.

Step1: initially, the sorted sublist contains the first element in the list. Insert 9 into the list sublist


2 9 5 4 8 1 6

Step2: the sorted sublist is [2, 9]. Insert 5 into the sublist

2 9 → 5 4 8 1 6

Step3: the sorted sublist is [2, 5, 9]. Insert 4 into the sublist.

2 5 → 9 → 4 8 1 6

Step4: the sorted sublist is [2, 4, 5, 9]. Insert 8 into the sublist.

2 4 5 9 → 8 1 6

Step5: the sorted sublist is [2, 4, 5, 8, 9]. Insert 1 into the sublist.

2 → 4 → 5 → 8 → 9 → 1 6

Step6: The sorted sublist is [1, 2, 4, 5, 8, 9]. Insert 6 into the sublist

1 2 4 5 8 → 9 → 6

Step7: The entire list is now sorted.

1 2 4 5 6 8 9

This method is described in Algorithm INSERTIONSORT Below:

Algorithm: INSERTIONSORT

Input: An array A[1..n] of n elements.

Output: A[1..n] sorted in nondecreasing order.

```
1. for i = 2 to n
2.   x = A[i]
3.   j = i-1
4.   while (j>0) and (A[j]>x)
5.     A[j+1] = A[j]
6.     j = j-1
7.   end while
8.   A[j+1] = x
9. End For
```


EXAMPLE: Illustrating Insertion Sort using the following array

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

$i=2$, $x=A[i]=9$, $j=1$, while $j_1>0$ and $A[j_1]>x_9$? No \Rightarrow goto line8

$A[j+1]=A[2]=x=9$

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

$i=3$, $x=A[i]=5$, $j=2$, while $j_2>0$ and $A[j_2]>x_5$? yes

$A[j+1]=A[j]$

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

$j=j-1=1$

while $j_1>0$ and $A[j_1]>x_5$? No \Rightarrow Goto line8

$A[j+1]=A[2]=x=5$

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

$i=4$, $x=A[i]=4$, $j=j-1=3$, while $j_3>0$ and $A[j_3]>x_4$? Yes goto line 5

$A[j+1]=A[j]$

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

$j=j-1=2$

while $j_2>0$ and $A[j_2]>x_4$? yes goto 5, 6

$A[j+1]=A[j]$

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

$j=j-1=1$

while $j_1>0$ and $A[j_1]>x_4$? No \Rightarrow goto line8

$A[j+1]=A[2]=x=4$

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

TO BE COMPLETED

Recursion

Recursion is technique that leads to elegant solution to problems that are difficult to program using simple loop. It enables you to develop a natural, straightforward simple solution to an otherwise difficult problem. To use recursion is to program using method, this leads to the following definition

Recursive Method

A recursive method is one that calls itself. This causes automatic repetition, a virtual loop. In fact, most algorithm that use iteration (for loop, while loop, e.t.c) can be recast, replacing each loop with recursive call. So recursion can be viewed as an alternative to iteration.

Parts of a Recursive Method

All recursive methods must have the following:

1. Base Case (i.e., when to stop)
2. Work toward Base Case
3. Recursive Call (i.e., call ourselves)

The "work toward base case" is where we make the problem simpler. The recursive call is where we use the same method to solve a simpler version of the problem. The base case is the solution to the "simplest" possible problem. (For example, the base case to adding a list of numbers would be if the list had only one number... thus the answer is the number).

The following sections introduce the concepts and techniques of recursive programming and illustrate with examples of how to "think recursively".

Case study 1: Computing Factorials

Many mathematical functions are defined using recursion. Let's begin with a simple example. The factorial of a number n can be recursively define as follows:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

How do you find $n!$ For a given n ? To Find $1!$ is easy because you know that $0!$ is 1, and so $1! = 1 \times 0!$ Assuming that you know $(n-1)!$, you can obtain $n!$ immediately by using $n \times (n-1)!$ thus, the problem of computing $n!$ is reduced to computing $(n-1)!$. When computing $(n-1)!$, you can apply the same idea recursively until n is reduced to 0.

Let `factorial(n)` be the method for computing $n!$ if you call the method with $n=0$, it immediately returns the result. The method knows how to solve the simplest case, which is referred to as the base case or the stopping condition. if you call the method with $n > 0$, it reduces the problem into a sub problem for computing the factorial of $n-1$. The sub problem is essentially same as the original problem, but it is simpler or smaller. Because the sub problem

has the same property as the original problem, you can call the method with a different argument, which is referred to as a recursive call.

The recursive method for computing $n!$ Can be simply described as follows:

```
long factorial (int n ) {  
    if (n == 0)  
        return 0;  
    else  
        return n × factorial (n-1);  
}
```

A recursive call can result in many more recursive calls, because the method keeps on dividing a sub problem into new sub problems. For a recursive method to terminate, the problem must eventually be reduced to a stopping case, at which point the method returns a result to its caller. The caller then performs a computation and returns the result is passed back to the original caller. The original problem can now be solved by multiplying n by the result of `factorial (n-1)`.

Identify the 3 parts of the recursive method (factorial):

All recursive methods must have the following three stages:

1. Base Case: `if (n == 0) return 0;`
2. "Work toward base case": $n-1$ becomes the new parameter to the same
This reduces the size of the parameter by 1, cause the method to approach the base case!
3. Recursive Call: `factorial (n-1);`

Illustrating the recursive `factorial` method

How does a recursive method works behind the scene? The figure below illustrate the execution the above `factorial` method, stating with $n=4$

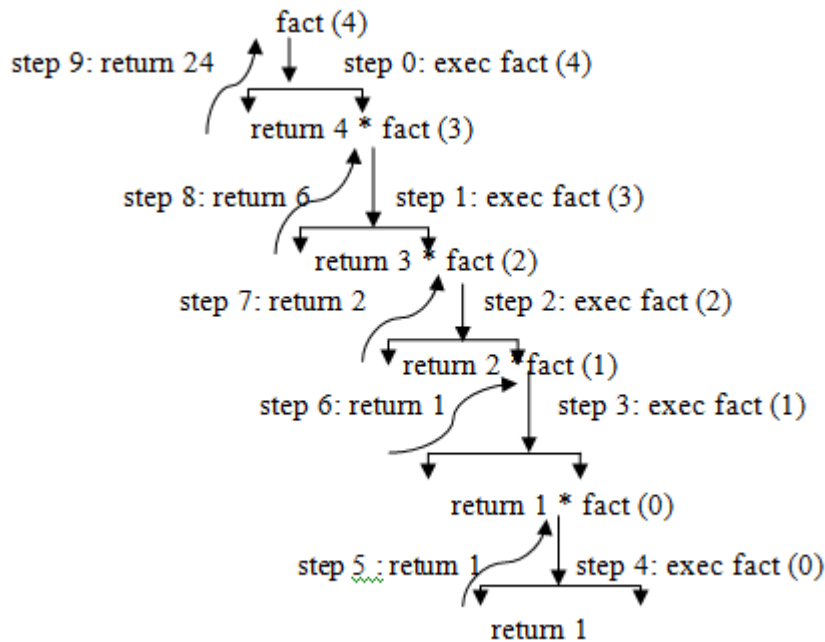


Figure 1: factorial of 4

Case study 2: Computing Case Fibonacci numbers

Consider the well-known Fibonacci-series problem

Series	0	1	1	2	3	5	8	13
Indexes	0	1	2	3	4	5	6	7

The Fibonacci-series begins with 0 and 1, and each subsequence number is the sum of the preceding two. The series can be recursively defined as

$$fib(index) = \begin{cases} 0 & \text{if } index = 0 \\ 1 & \text{if } index = 1 \\ fib(index - 2) + fib(index - 1), & \text{if } index \geq 2 \end{cases}$$

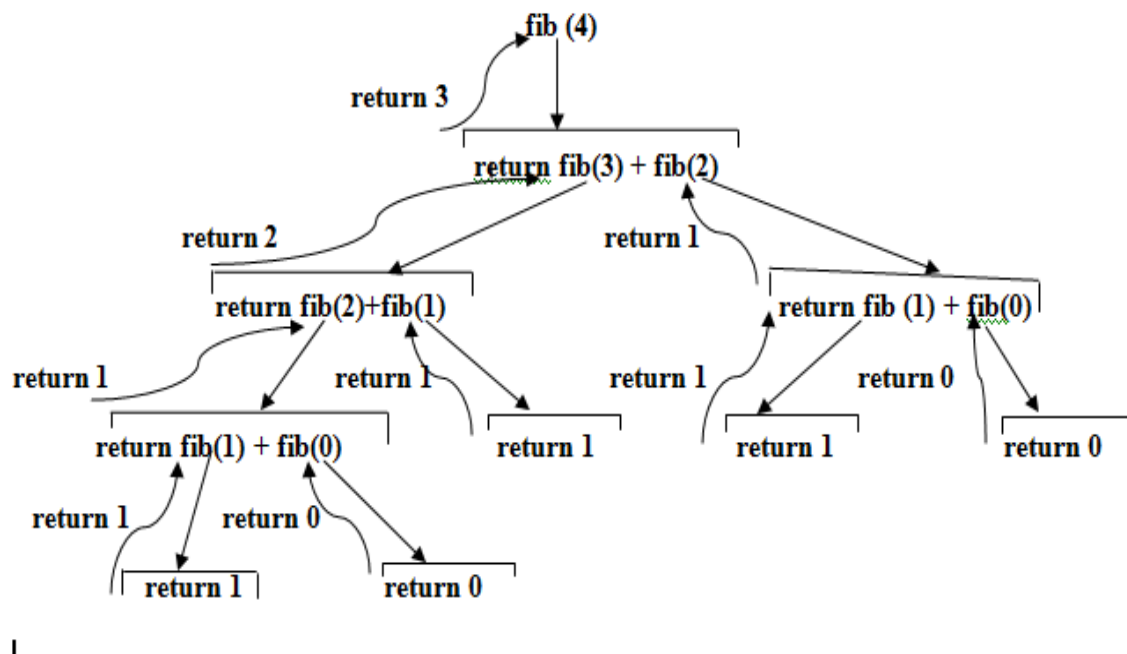
How do you find fib(index) for a given index? it is easy to find fib(2), because you know fib(0) and fib(1). Assuming that you know fib(index-2) and fib(index-1), you can obtain fib(index) immediately. Thus, the problem of computing fib(index) is reduced to computing fib(index-2) and fib(index-1). When doing so, you apply the idea recursively until index is reduced to 0 or 1.

The base case is index=0 or index=1. If you call the method with index=0 or index=1, it immediately returns the result. If you call the method with index>=2, it divides the problem

into two sub problems for computing $\text{fib}(\text{index}-1)$ and $\text{fib}(\text{index}-2)$ using recursive calls. The recursive method for computing the $\text{fib}(\text{index})$ can be written as follows:

```
long fib (long index)    {
    if (index == 0)      // base case
        return 0;
    else if (index == 1) // base case
        return 1;
    else
        return fib(index-1) + fib(index - 2);
} // fib ( )
```

Illustrating the $\text{fib}()$ Method



The figure above shows the successive recursive calls for evaluating $\text{fib}(4)$. The original method, $\text{fib}(4)$, makes two recursive calls, $\text{fib}(3)$ and $\text{fib}(2)$, and then returns $\text{fib}(3)+\text{fib}(2)$. But in what order are these methods called? In java, operands are evaluated from left to right, so $\text{fib}(2)$ is called after $\text{fib}(3)$ is completely evaluated.

MERGE SORT

The merge sort algorithm can be described recursively as follows: the algorithm divides the array into 2 halves and applies a merge sort on each half recursively. After the 2 halves are sorted, merge them. The algorithm for a merge sort is given below.

Algorithm: mergeSort

input : An array, lower index (p) , high index ®

output: A is sorted form

```
1.  mergeSort (A, p, r) {
2.    if P < R {
3.      q = (p + r)/2
4.      mergeSort (A, p, q)
5.      mergeSort (A, p, q+1, r)
6.      merge (A, p, q, r)
7.    }
8.  }
```

For example: using merge sort to sort the array (2 9 5 4 8 1 6 7)

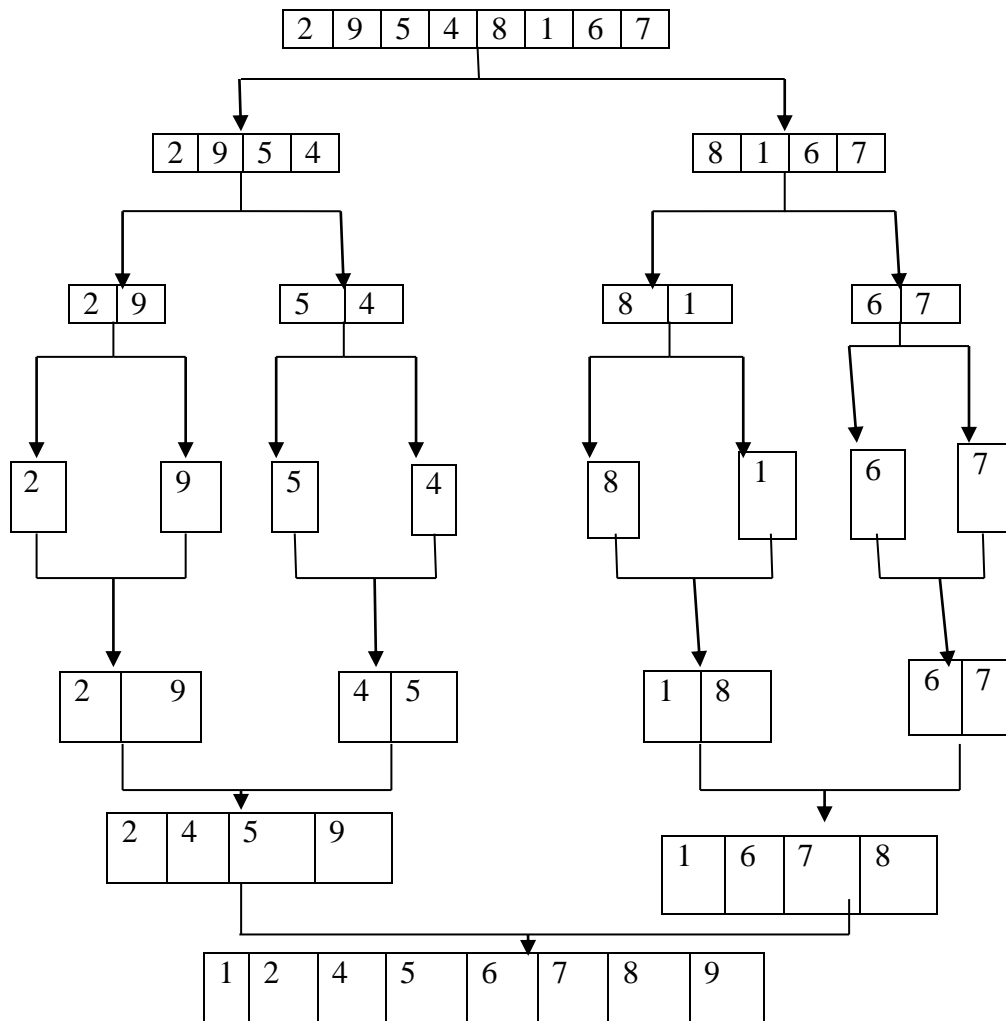


Figure ms: Merge sort employs a divide-and-conquer approach to sort the array.

The figure above illustrate the merge sort of an array of eight elements (2 9 5 4 8 1 6 7). The original array is split into (2 9 5 4) and (8 1 6 7). Apply a merge sort on these two sub arrays recursively to split (2 9 5 4) into (2 9) and (5 4) and (8 1 6 7) into (8 1) and (6 7). This process continues until the sub array contains only one element. For example, array (2 9) is split into sub arrays (2) and (9). Since array (2) contains a single element, it cannot be further split. Now merge (2) with (9) into a new sorted array (2 9); merge (5) with (4) into a new sorted array (4 5). Merge (2

9) with (4 5) into a new sorted array (2 4 5 9), and finally merge (2 4 5 9) with (1 6 7 8) into a new sorted array (1 2 4 5 6 7 8 9).

MERGING TWO SORTED LISTS

Suppose we have an array $A[1..m]$ and the three indices p , q and r , with $1 \leq p \leq q < r \leq m$, such that both the sub arrays $A[p, q]$ and $A[q+1, r]$ are individually sorted in non decreasing order. We want to rearrange the elements in A so that the elements in the sub array $A[p..r]$ are sorted in non-decreasing order. This process is referred to as merging $A[p..q]$ with $A[q+1..r]$. An algorithm to merge these two sub arrays works as follows:

We maintain two pointers s and t that initially point to $A[p]$ and $A[q+1]$, respectively. We prepare an empty array $B[p..r]$ which will be used as temporary storage. Each time, we compare the element $A[s]$ and $A[t]$ and append the smaller of the two to the auxiliary array B ; if they are equal we will choose to append $A[s]$. Next, we update the pointers: if $A[s] \leq A[t]$, then we increment s , otherwise we increment t . This process ends when $s=q+1$ or $t=r+1$. In the first case, we append the remaining elements $A[t..r]$ to B , and in the second case, we append $A[s..q]$ to B . Finally, the array $B[p..r]$ is copied back to $A[p..r]$. This algorithm is given in Algorithm MERGE

Algorithm: MERGE

Input: An array $A[1..m]$ of elements and three indices p , q , and r , with $1 \leq p \leq q < r \leq m$, such that both the sub arrays $A[p..q]$ and $A[q+1, r]$ are sorted individual in non decreasing order.

Output: $A[p..r]$ Contains the result of merging the sub arrays $A[p..q]$ and $A[q+1..r]$.


```

1. Comment: B[p..r] Is an auxiliary array.
2. s = p; t = q+1; k=p
3. While s ≤ q and t ≤ r
4.     if A[s] ≤ A[t] then
5.         B[k] = A[s]
6.         s= s+1
7.     else
8.         B[k] = A[t]
9.         t= t+1
10.    End If
11.    k = k + 1
12. End while
13. if s == q+1 then
14.     B[k..r] = A[t..r]
15. else
16.     B[k..r] = A[s..q]
17. End if
18. A[p..r]=B[p..r]

```

(ii) Illustrating the above Merge algorithm using the following array A, with

$p = 1$ and $r = 5$.

$s = p=1; t = q+1; k=p$

	1	2	3	4	5
A →	21	34	50	20	40
	↑		↑	↑	
	p, s		t	r, m	

<p>Iteration 1</p> <p>while $s^1 \leq q^3$ and $t^4 \leq r^5 \rightarrow$yes</p> <p>Is $A[s=1]^{21} \leq A[t=4]^{20} ? \rightarrow$No</p> <p>$B[k=1] = A[t=4] = 20$</p> <p>$s = 1$</p> <p>$t = t + 1 = 4 + 1 = 5$</p> <p>$k = k + 1 = 1 + 1 = 2$</p>	<div><div><div>12345</div><div>A =</div><div><div>2134502040</div><div>s </div></div></div></div>
---	--

<p>Is $s == q+1$? No</p> <p>$B[k^5..r^5] = A[s^3..q^3]$</p> <p>i.e $B[5] = A[3]$</p>	<div><div><div>12345</div><div>A =</div><table><tr><td>21</td><td>34</td><td>50</td><td>20</td><td>40</td></tr></table></div><div><div>12345</div><div>B =</div><table><tr><td>20</td><td>21</td><td>34</td><td>40</td><td>50</td></tr></table></div></div>	21	34	50	20	40	20	21	34	40	50
21	34	50	20	40							
20	21	34	40	50							
<p>$A[p..r] = B[p..r]$</p>	<div><div><div>12345</div><div>A =</div><table><tr><td>20</td><td>21</td><td>34</td><td>40</td><td>50</td></tr></table></div></div>	20	21	34	40	50					
20	21	34	40	50							