

SEARCHING

Searching – It is the process of looking/searching for a specific element in the given list of elements.

Searching Techniques –

- Linear Search
- Binary Search

Linear Search:

Linear Search – It is the process of comparing the given **searching element s** with the each and every element in the given **list of elements A** .

- Compares searching element with each and every element of the array/List (data structure), if the searching element s found in the list A then it returns the location of searching element from the list of elements otherwise it returns the -1 – **sequential search**.
- Used mostly for unordered list of elements.
- For ex: `int A[] = {10, 12, 23, 75, 3, 4, 9, 1, 8, 5, 14, 34, 56, 76, 31, 26};`
searching element $s = 8$; s is found at position 9.

Algorithm – Linear Search

Linear Search(Element Type A[], Element Type s , int n) // n is the number of element in A

- **Step 1** – Let $i := 0$
- **Step 2:** Compare the searching element s with the i^{th} element in the list A .
- **Step 3** - If both are matched, then return $i+1$.

- **Step 4** - If both are not matched, then increment i . If i is less than n and then repeat the steps 2 and 3.
- **Step 5** - Return -1. // Element not found

Algorithm – Linear Search

Linear Search(Element Type A[], Element Type s, int n) // n is the number of element in A

- Step 1: [INITIALIZE] SET pos = -1, i = 0
- Step 2: Repeat Step 3 and Step 4 while $i < n$
- Step 3: IF $A[i] = s$
 - SET POS = $i + 1$
 - PRINT “ Element found in position : POS”
 - Go to Step
 - [END OF IF]
- Step 4: SET $i = i + 1$
- [END OF LOOP]
- Step 5: IF POS = -1
 - PRINT VALUE IS NOT PRESENT IN THE ARRAY
 - [END OF IF]
- Step 6: EXIT

Example:

{23, 45, 3, 5, 7, 4, 77, 46} searching 77 using linear search:

23	45	3	5	7	4	77	46
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

S=77

23	45	3	5	7	4	77	46
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

S=77

23	45	3	5	7	4	77	46
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

S=77

23	45	3	5	7	4	77	46
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

S=77

23	45	3	5	7	4	77	46
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

S=77

23	45	3	5	7	4	77	46
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

S=77

23	45	3	5	7	4	77	46
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Matched at index 6

7 comparisons are used.

Complexity Analysis:

- Best Case: $O(1)$ if searching element is found at first index.
- Worst Case: $O(N)$ searching element is found at the last position
- Average Case: $O(N)$. searching element is found at middle of the array
- Space complexity: $O(1)$

Limitations of Linear Search:

- Linear search cannot take the advantage in reducing the number of comparisons when the elements are in sorting order.

Binary search

- The input for the Binary search is the sorted elements (pre constraint).
- Idea - Compare searching element with the middle element in the list. There are three possibilities
 1. If the middle element is greater than searching element then the searching element may found only in the first half of the elements (Searching array size reduced to half).
 2. If the middle element is less than searching element then the searching element may found only in the second half of the elements (Searching array size reduced to half)
 3. If the middle element matches with the searching element, then return the position of the middle element.

The above three steps are repeated until it matches or elements exhausted.

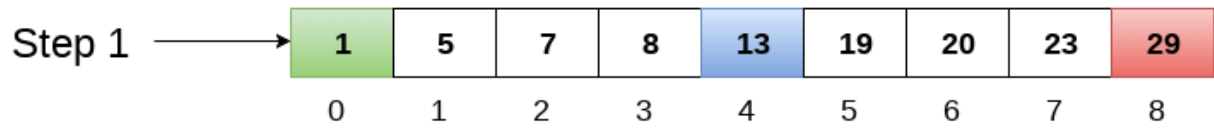
Algorithm:

- **Step 1:** [INITIALIZE] SET BEG = lower_bound
END = upper_bound, POS = - 1
- **Step 2:** Repeat Steps 3 and 4 while BEG <=END
- **Step 3:** SET MID = (BEG + END)/2
- **Step 4:** IF A[MID] = VAL
SET POS = MID
PRINT POS
Go to Step 6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]
- **Step 5:** IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
[END OF IF]
- **Step 6:** EXIT

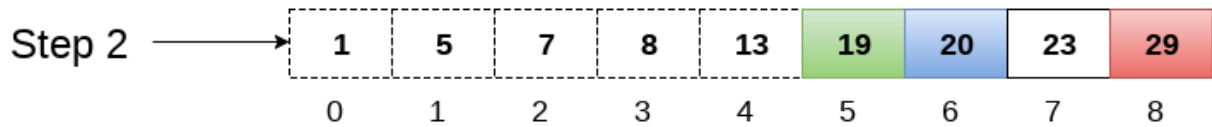
Complexity Analysis:

- Best Case: $O(1)$ if searching element is found at mid position.
- Worst Case: $O(\log n)$ searching element is not
- Average Case: $O(\log n)$. searching element is found at left or right part of the array
- Space complexity: $O(1)$

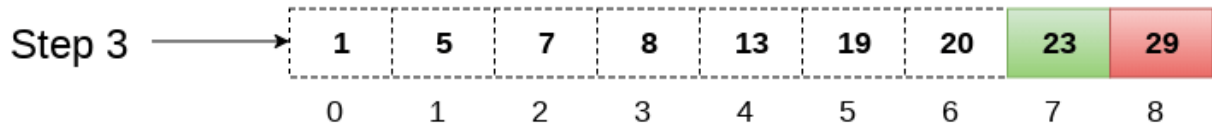
Item to be searched = 23



$a[mid] = 13$
 $13 < 23$
 $beg = mid + 1 = 5$
 $end = 8$
 $mid = (beg + end)/2 = 13 / 2 = 6$



$a[mid] = 20$
 $20 < 23$
 $beg = mid + 1 = 7$
 $end = 8$
 $mid = (beg + end)/2 = 15 / 2 = 7$



$a[mid] = 23$
 $23 = 23$
 $loc = mid$

Return location 7

Sorting

Definition: Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing with numerical data, or alphabetically with character data.

Sorting Methods:

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

Selection sort:

- The simplest technique to sort the given list of elements

Idea





1. Select the smallest element in the given list of n elements and place it in the first position.
2. Now Select smallest element in the remaining list of $n-1$ elements and place it in the second position.
3. Repeat this procedure until the entire array is sorted.

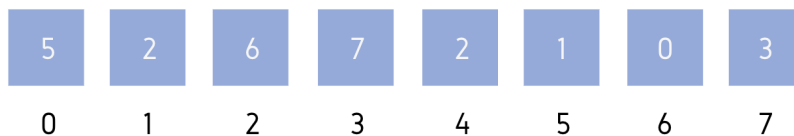
Detailed procedure

- The first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped, so that first position is filled with the smallest element in the sorted order.
- Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped.

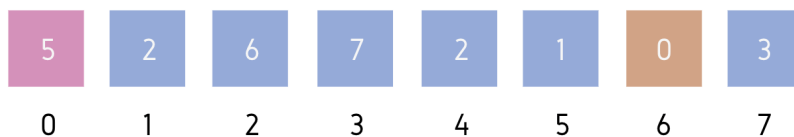
This procedure is repeated until the entire list is sorted.

A selection sort works as follows:

-  Part of unsorted array
-  Part of sorted array
-  Leftmost element in unsorted array
-  Minimum element in unsorted array



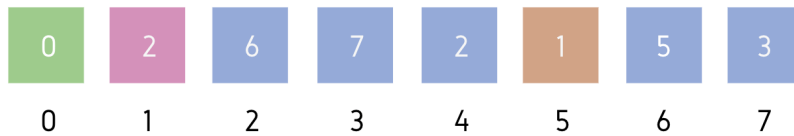
This is our initial array $A = [5, 2, 6, 7, 2, 1, 0, 3]$



Leftmost element of unsorted part = $A[0]$

Minimum element of unsorted part = $A[6]$

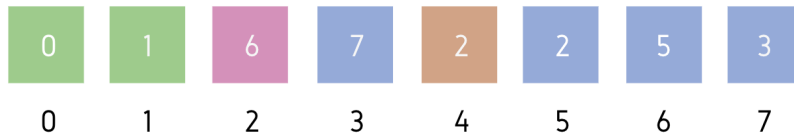
We will swap $A[0]$ and $A[6]$ then, make $A[0]$ part of sorted subarray.



Leftmost element of unsorted part = $A[1]$

Minimum element of unsorted part = $A[5]$

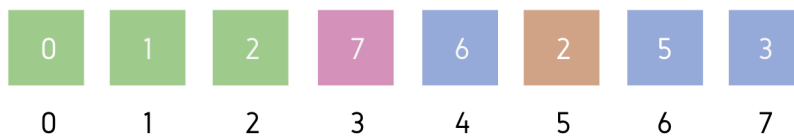
We will swap $A[1]$ and $A[5]$ then, make $A[1]$ part of sorted subarray.



Leftmost element of unsorted part = $A[2]$

Minimum element of unsorted part = $A[4]$

We will swap $A[2]$ and $A[4]$ then, make $A[2]$ part of sorted subarray.



Leftmost element of unsorted part = $A[3]$

Minimum element of unsorted part = $A[5]$

We will swap $A[3]$ and $A[5]$ then, make $A[3]$ part of sorted subarray.



Leftmost element of unsorted part = $A[4]$

Minimum element of unsorted part = $A[7]$

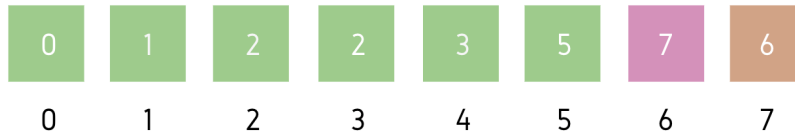
We will swap $A[4]$ and $A[7]$ then, make $A[4]$ part of sorted subarray.



Leftmost element of unsorted part = $A[5]$

Minimum element of unsorted part = A[6]

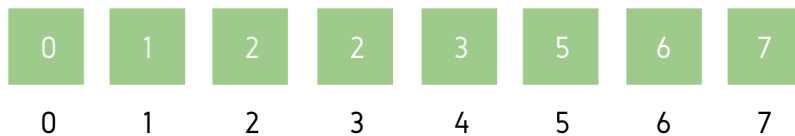
We will swap A[5] and A[6] then, make A[5] part of sorted subarray.



Leftmost element of unsorted part = A[6]

Minimum element of unsorted part = A[7]

We will swap A[6] and A[7] then, make A[6] part of sorted subarray.



This is the final sorted array.

Algorithm:

Selection_sort (Element Type A[],int n) // Assumption – index starts with 0

1. Set $i = 0$, $j = i + 1$
2. while($i < n-1$) // Number of passes
 - i. set $j = i + 1$ // j starts from the next element of i
 - ii. while($j < n$) // j go through the j^{th} element to n^{th} element
 - i. if ($A[j] < A[i]$) // if i pointed element is greater than j pointed element, then swap them
 - i. Swap ($A[i]$, $A[j]$)
 - ii. $j = j + 1$.
 - iii. $i = i + 1$.
3. Return.

Complexities:

Time complexity: $O(n^2)$ [In all cases]

Space complexity: $O(1)$

Bubble sort

Repeatedly move the largest element to the highest index position of the array.

Bubble sort –

The algorithm does two steps:

1. Starts at one end of the array and make repeated scans through the list comparing successive pairs of elements (Adjacent elements).
2. If the first element is larger than the second, called as an *inversion*, then the values are swapped.

Idea – The algorithm works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.

1. The pass through the list is repeated until no swaps are needed.
2. This technique is called *bubble sort* or *sinking sort* because the smaller values gradually
3. "*Bubble*" their way upward to the top of the array like air bubbles rising in water, while the

Example:

Larger values sink to the bottom of the array.

Pass 1: 3 5 2 6 8 1 9 (No change)

- 3 5 2 6 8 1 9 (Swap 2 and 5)
- 3 2 5 6 8 1 9 (No change)
- 3 2 5 6 8 1 9 (No change)
- 3 2 5 6 8 1 9 (Swap 8 and 1)
- 3 2 5 6 1 8 9 (No change)
- 3 2 5 6 1 8 9

(Observation – After pass-1 the largest element reached to its position and the smaller elements are moving slowly up)

(The number of comparisons is 6, i.e. (n-1))

Pass 2: 3 2 5 6 1 8 9 (Swap 3 and 2)

- 2 3 5 6 1 8 9 (No change)
- 2 3 5 6 1 8 9 (No change)
- 2 3 5 6 1 8 9 (Swap 6 and 1)
- 2 3 5 1 6 8 9 (No change)
- 2 3 5 1 6 8 9

(Observation – The second largest reached to its position and smaller elements slowly moving up)

Pass-3

- 2 3 5 1 6 8 9 (No change)
- 2 3 5 1 6 8 9 (No change)
- 2 3 5 1 6 8 9 (Swap 5 and 1)

- 2 3 1 5 6 8 9 (No change)

2 3 1 5 6 8 9

Pass-4

- 2 3 1 5 6 8 9 (No change)
- 2 3 1 5 6 8 9 (Swap 3 and 1)
- 2 1 3 5 6 8 9 (No change)
- 2 1 3 5 6 8 9

(Observation – The fourth largest element reached to its position and smaller elements slowly moving up)

(The number of comparisons are 3, i.e. (n-4))

Pass-5

- 2 1 3 5 6 8 9 (Swap 2 and 1)
- 1 2 3 5 6 8 9 (No change)

1 2 3 5 6 8 9

(Observation – The fifth largest element reached to its position and the smaller elements slowly moving up)

(The number of comparisons are 2, i.e. (n-5))

Pass-6

- 1 2 3 5 6 8 9 (No change)

1 2 3 5 6 8 9

(Observation – The sixth largest element reached to its position, so the last element (seventh one here) is also reached its position)

(The number of comparisons are 1, i.e. (n-6))

Complexities:

Time complexity: $O(n^2)$ [In average and worst case]
 = $O(n)$ [Best case](when the list is sorted)

Space complexity: $O(1)$

INSERTION SORT

The idea of insertion sort is similar to the idea of sorting playing cards

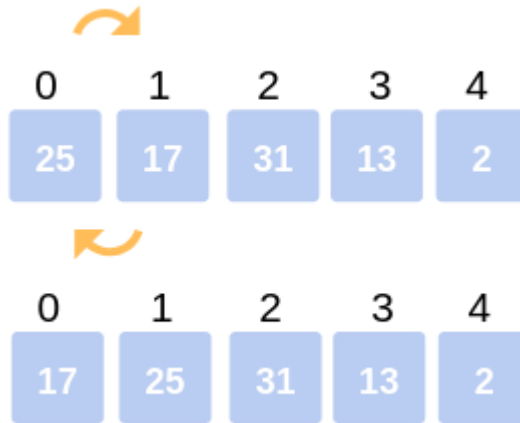
Pick a card and place it at the right place by moving cards to the right.

- Elements are considered in sorted and unsorted order.
- Every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.
- The insertion sort algorithm is performed using the following steps...
- *Step 1* - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- *Step 2*: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- *Step 3*: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Consider the following array: 25, 17, 31, 13, 2

First Iteration: Compare 25 with 17. The comparison shows $17 < 25$. Hence swap 17 and 25.

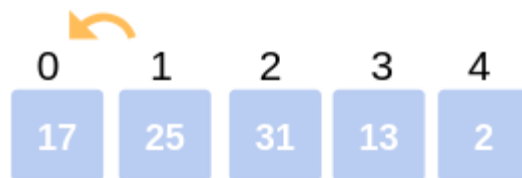
- The array now looks like:
- **17, 25, 31, 13, 2**



- First Iteration

Second Iteration: Begin with the second element (25), but it was already swapped on for the correct position, so we move ahead to the next element.

- Now hold on to the third element (31) and compare with the ones preceding it.
- Since $31 > 25$, no swapping takes place.
- Also, $31 > 17$, no swapping takes place and 31 remains at its position.
- The array after the Second iteration looks like:
- **17, 25, 31, 13, 2**

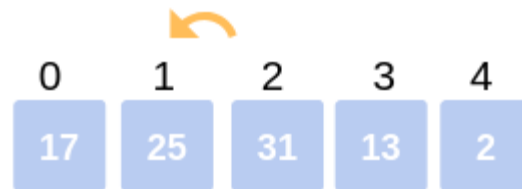


Second Iteration

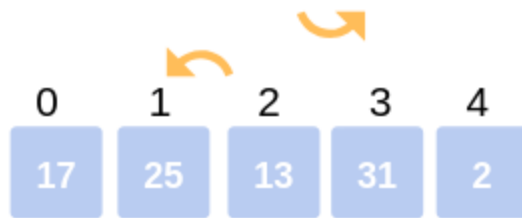
Third Iteration: Start the following Iteration with the fourth element (13), and compare it with its preceding elements.

- Since $13 < 31$, we swap the two.
- Array now becomes: 17, 25, 13, 31, 2.

- But there still exist elements that we haven't yet compared with 13. Now the comparison takes place between 25 and 13. Since, $13 < 25$, we swap the two.
- The array becomes **17, 13, 25, 31, 2**.
- The last comparison for the iteration is now between 17 and 13. Since $13 < 17$, we swap the two.
- The array now becomes **13, 17, 25, 31, 2**.

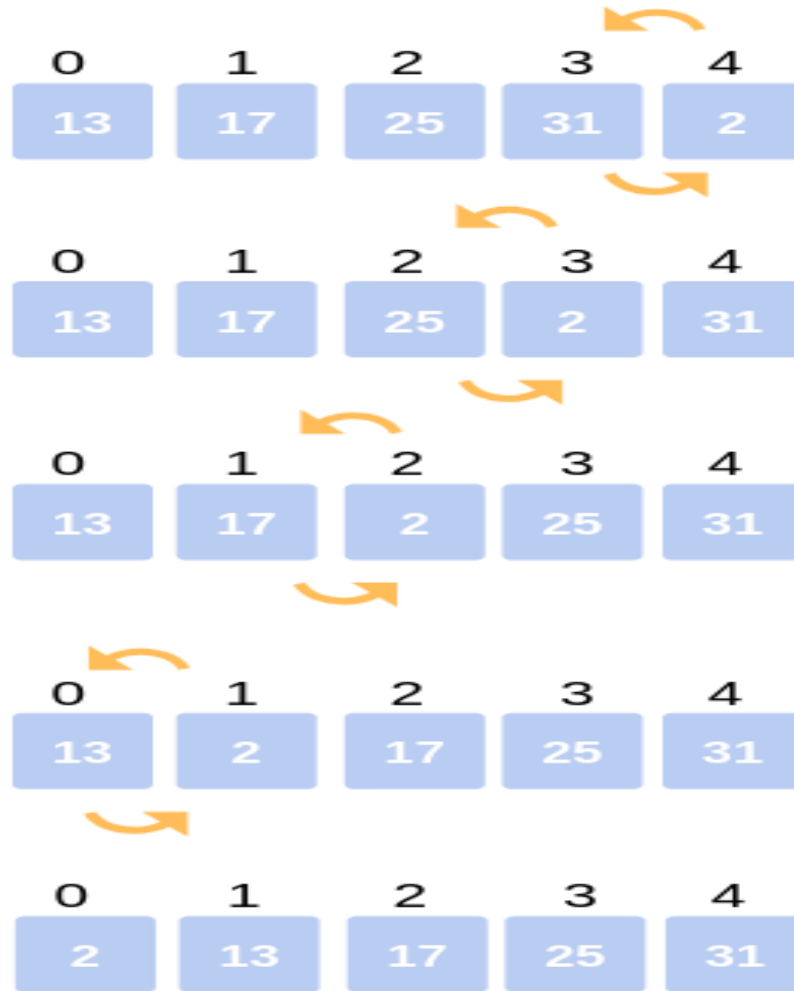


- Third Iteration



•

- **Fourth Iteration:** The last iteration calls for the comparison of the last element (2), with all the preceding elements and make the appropriate swapping between elements.
- Since, $2 < 31$. Swap 2 and 31.
- Array now becomes: 13, 17, 25, 2, 31.
- Compare 2 with 25, 17, 13.
- Since, $2 < 25$. Swap 25 and 2.
- **13, 17, 2, 25, 31.**
- Compare 2 with 17 and 13.
- Since, $2 < 17$. Swap 2 and 17.
- Array now becomes:
- **13, 2, 17, 25, 31.**
- The last comparison for the Iteration is to compare 2 with 13.
- Since $2 < 13$. Swap 2 and 13.
- The array now becomes:
- **2, 13, 17, 25, 31.**
- This is the final array after all the corresponding iterations and swapping of elements.



•

Algorithm

//Assumption is that index starts with 0

- `int a[100] ; //declare array a of size 100`
- `Input n // n is used to sort how many numbers`
- `for i = 0 to n-1 // read the array elements`
 - `Read a[i]; i++;`
- `for i = 1 to n-1`
 - `for (k = i; k > 0 and a[k] < a[k-1]; k--)`
 - `swap a[k] and a[k-1] // invariant: a[1..i] is sorted`
- `output a[1] to a[n]`

Complexities:

Time complexity: $O(n^2)$ [In average and worst case]
= $O(n)$ [Best case](when the list is sorted)

Space complexity: $O(1)$

Merge Sort

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

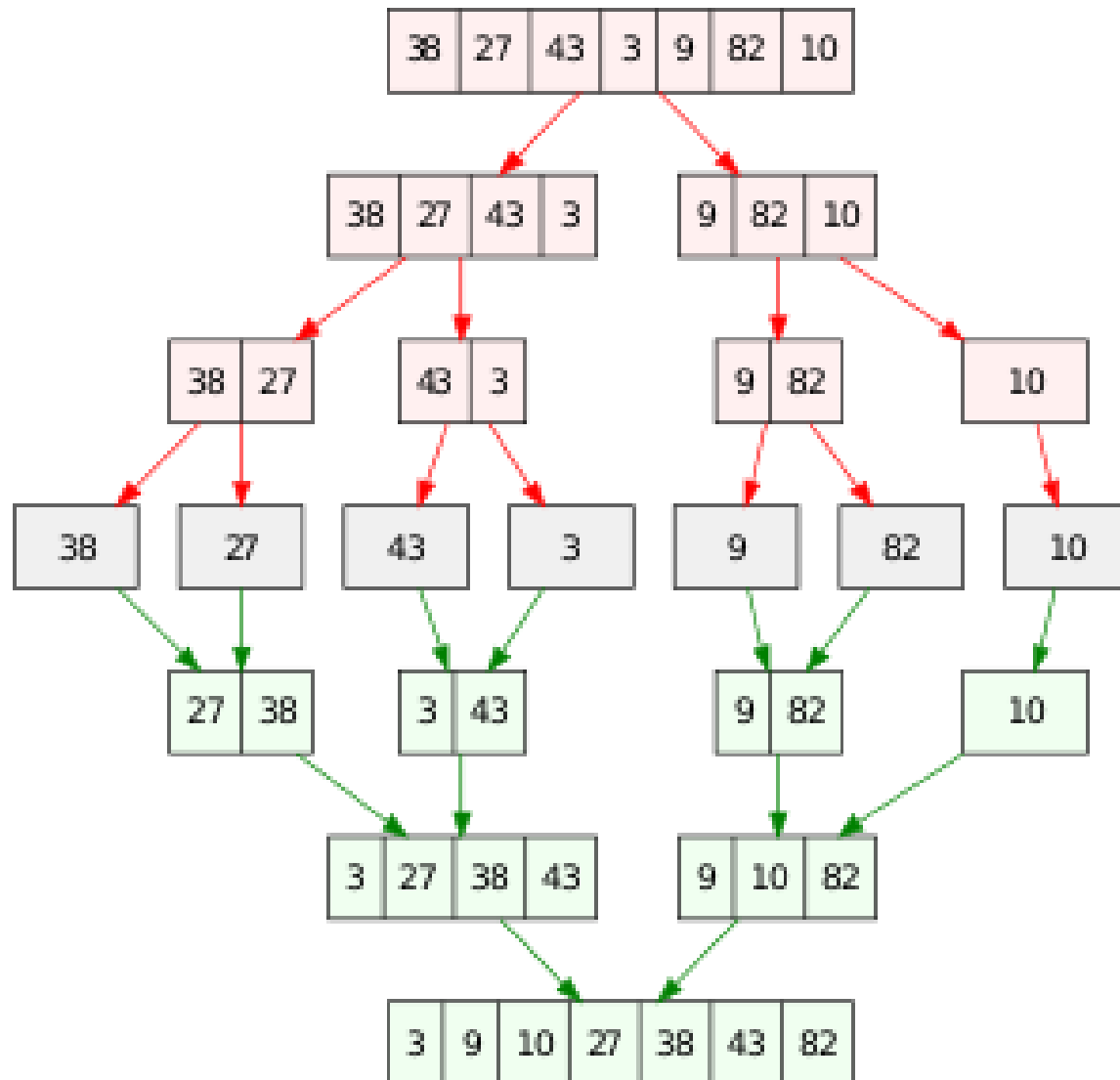
Merging is the process of combining two sorted lists to make one bigger sorted list

Basic Idea of Merge sort

1. Divide the given input list into two halves

(Apply this step on the divided sub lists till the sub list size reached to one)

2. Recursively sort each sub list.
3. Merge the two sorted sub lists (From bottom to top).



Pseudo code:

```
void Merge_Sort(int A[], int low, int high) // A – input array, low – lower index, high – higher index
```

```
{ if (low < high) // If there are more than one element
```

```
    { int mid = (low + high) / 2;
```

Merge_Sort(A, low, mid); *// call the merge sort on first half of the elements of A*

Merge_Sort(A, mid+1, high); *// call the merge sort on second half of the elements of A*

Merge(A, low, mid, high); *// Merge the sorted elements of the first half and second half*

}

}

Complexities:

Time complexity: $O(n \cdot \log N)$ [In all cases]

Space complexity: $O(N)$

Quick sort:

Quick sort is the widely used sorting algorithm that makes $n \log n$ comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach.

- Idea
 - Select randomly one element (Call it as *pivot*) from the elements to sort (generally first element)
 - Partition the given array into two subsists such that one list contains all the elements less than or equal to chosen element (pivot element) and the another list contains the elements greater than chosen element (pivot element)
 - Now the pivot element is in its sorted position.

- Apply the same procedure on the sub lists till there are no sub lists.
- Partition procedure
 - Select the first element in the list as pivot element.
 - Initialize one pointer called it as left at the first element of the list, and initialize another pointer called it as right at the last element of the list.
 - Repeatedly move the left pointer to next position until the left pointer pointed element is less than or equal to pivot element or the pointer reached to last position.
 - Repeatedly move the right pointer to its previous position until the right pointer pointed element is greater than pivot element or the pointer reached to first position.
 - If $left < right$ // *if they do not cross each other*
 - Swap the elements pointed by left and right and continue the steps 3 and 4
 - If $left \geq right$ // *if they cross each other or they both points same position*
 - Swap the right pointed element and the pivot element.

Pseudo code:

void Quick_Sort(int A[], int low, int high) // A – input array, p is the lower limit of A and q is the higher limit of A

{ if (low < high) // If there are more than one element

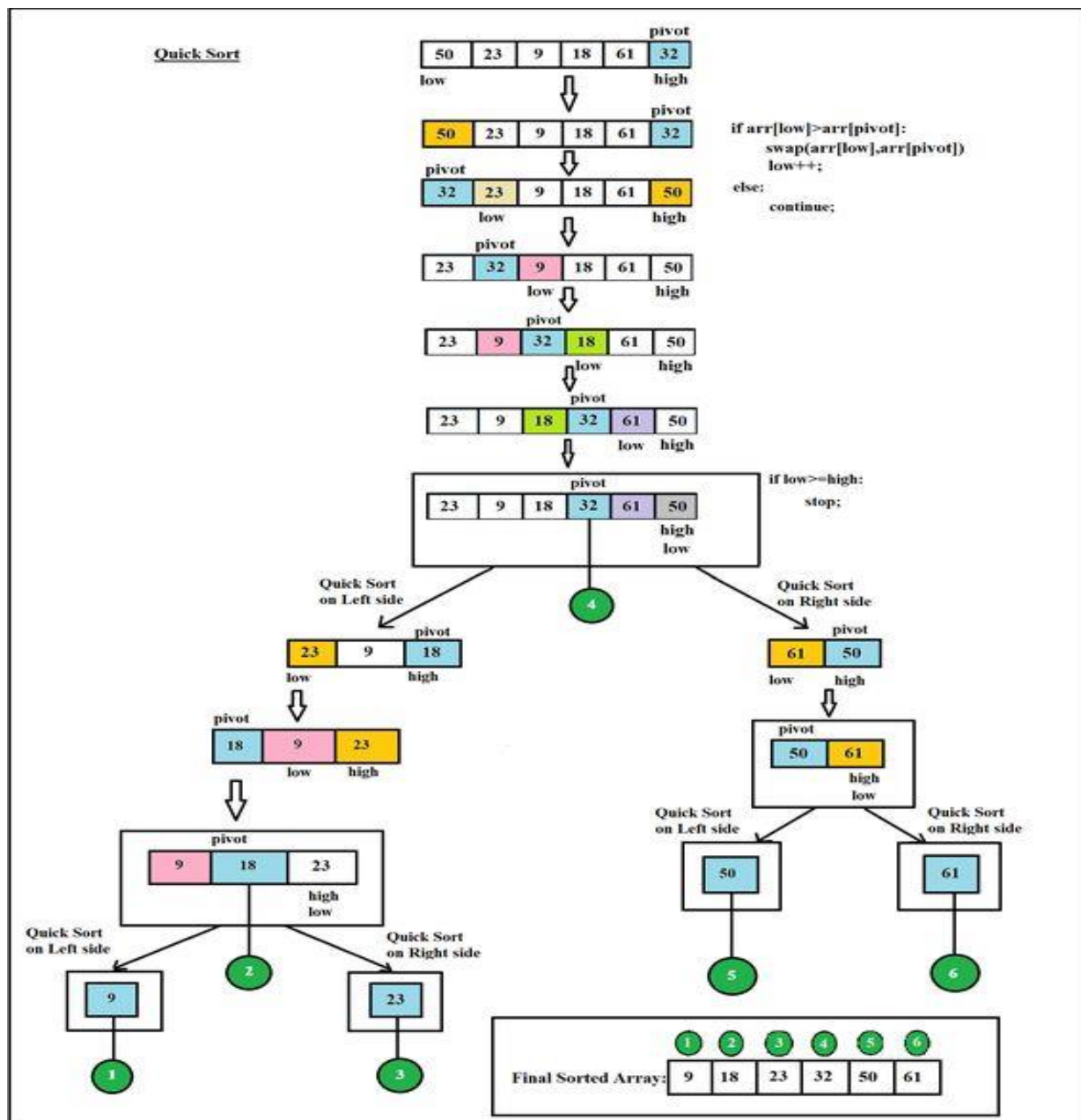
{ int p = Partition(A, low, high);

Quick_Sort(A, low, p-1); // call the merge sort on first half of the elements of A

Quick_Sort(A, p+1, high); // call the merge sort on second half of the elements of A

}

}



Complexities:

Time complexity: $O(N \cdot \log N)$ (in best and average cases)

$O(N^2)$ (in worst case)

Space complexity: $O(n)$

Heap sort:

Heap is a tree which must satisfy two properties

1. Structure Property- The tree must be completely filled, with the possible exception of the bottom level, which is filled from left to right (complete binary tree). All the leaves must be at level h or $h-1$.
2. Order Property – The key of parent node must be smaller (or larger) than its children.

Types of Heaps

1. Min heap – The value of a node must be less than or equal to its children.
2. Max heap - The value of a node must be greater than or equal to its children.

Time complexity: $O(N \cdot \log(n))$

Space complexity: $O(1)$

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(N)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$