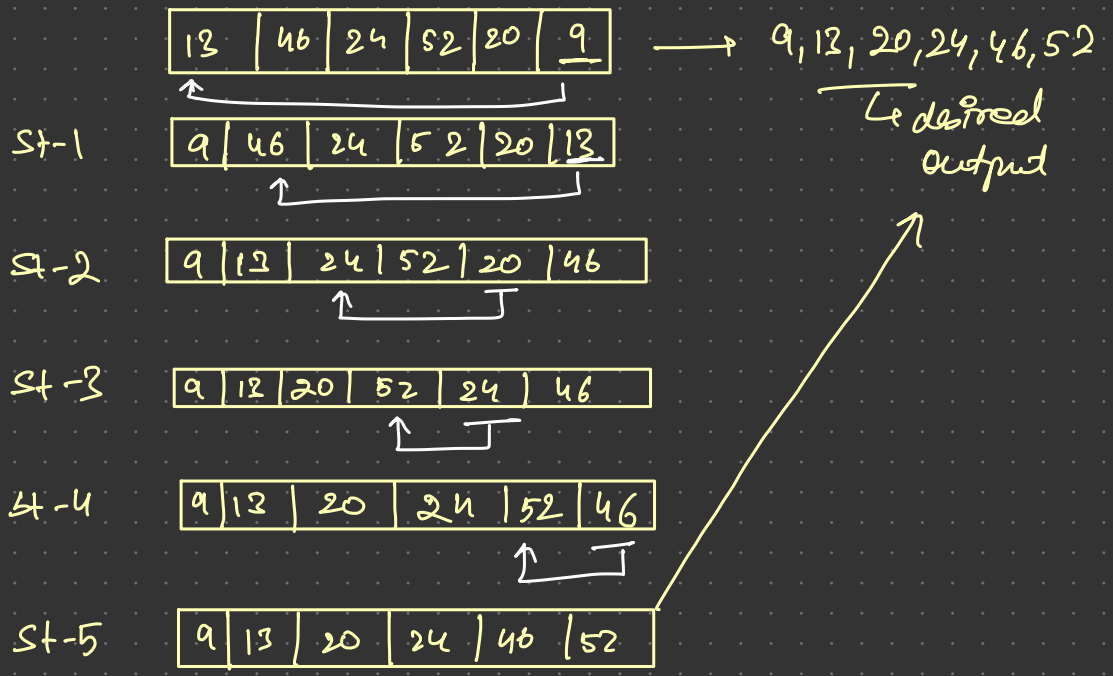
Sorting Algorithms

→ Selection Sort →

- As name suggest "selection"
- selection of minimum element
- swapping minimum with desired position in the array.



Time complexity → $O(N^2)$

Auxiliary space → $O(1)$

→ It is not stable as the relative order of same key is not maintained, it can be made stable.

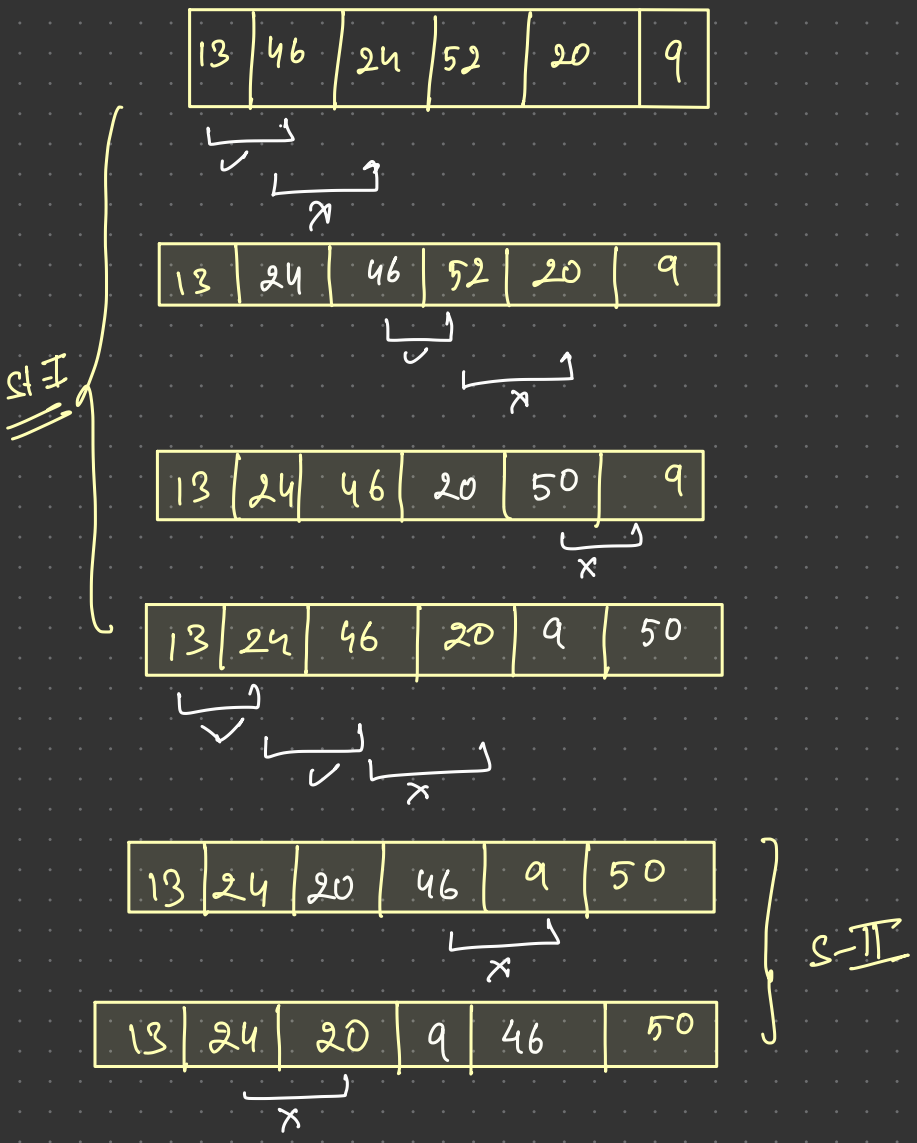
→ It is an in-place algorithm i.e. no extra space is used.

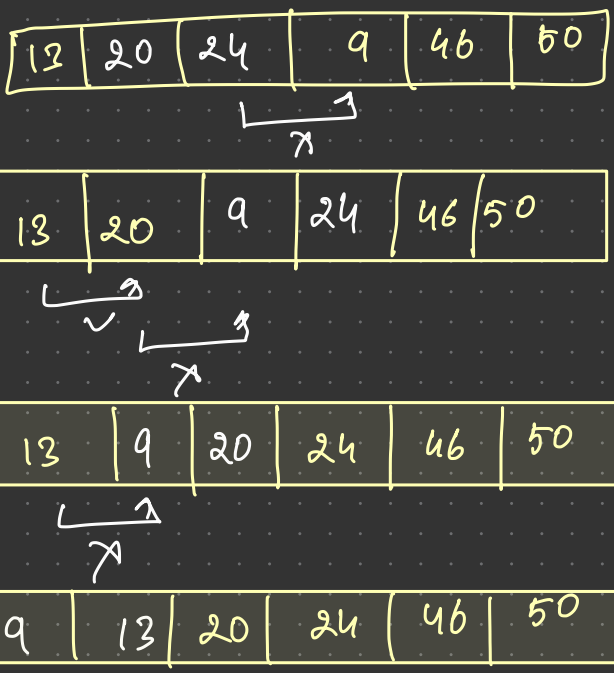
```cpp
void selectionSort(vector<int>& nums)
{
    //THIS ONE IS UNSTABLE
    int k=0;
    for(int i=0;i<nums.size();i++)
    {
        int mini=i;
        for(int j=i+1;j<nums.size();j++)
        {
            if(nums[j]<nums[mini])
            mini=j;
        }
        swap(nums[k++],nums[mini]);
    }
}
void selectionSortStable(vector<int>& nums)
{
    for(int i=0;i<nums.size();i++)
    {
        int min=i;
        for(int j=i+1;j<nums.size();j++)
        {
            if(nums[j]<nums[min])
            min=j;
        }
        int key=nums[min];
        for(int j=min;j>i;j--)
        {
            nums[j]=nums[j-1];
        }
        nums[i]=key;
    }
}
```

→ Bubble Sort

— Traverse from left and compare adjacent elements and higher one is placed at right.

— Two adjacent are sorted at one step





Time complexity → $O(N^2)$

Auxiliary space → $O(1)$

→ Bubble sort is stable

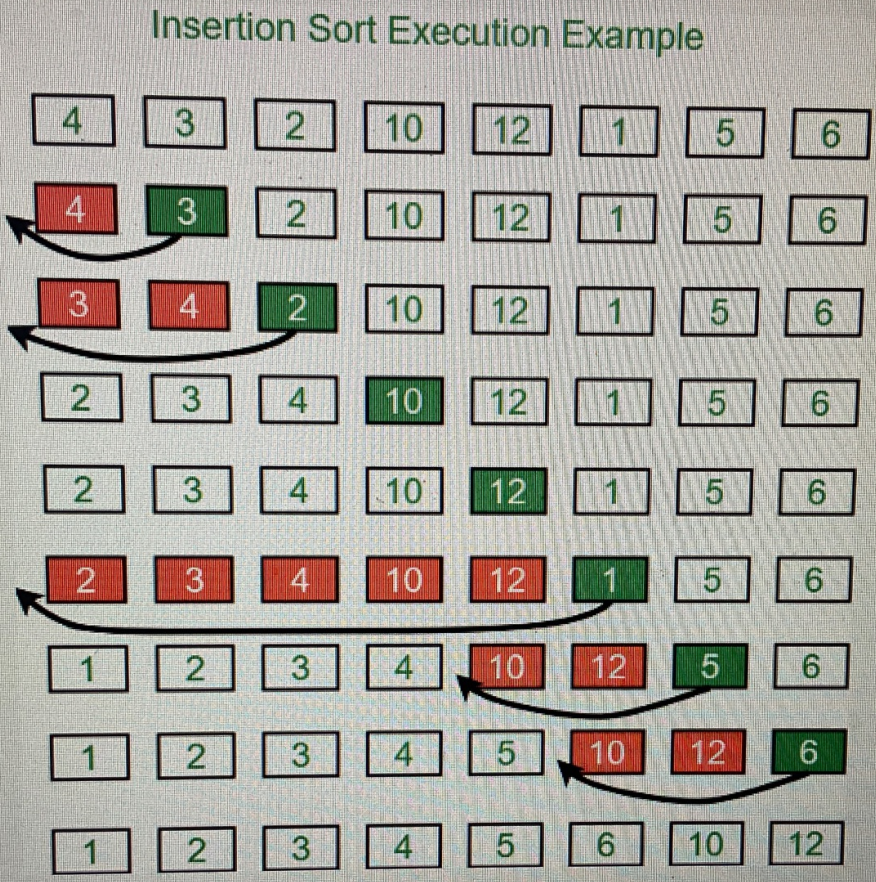
→ Buble sort is inplace

```
6
7   void bubbleSort(vector<int>& nums)
8   {
9       for(int i=0;i<nums.size();i++)
10      {
11          for(int j=0;j<nums.size()-i-1;j++)
12          {
13              if(nums[j]>nums[j+1])
14              swap(nums[j],nums[j+1]);
15          }
16      }
17  }
18
```

→ Insertion Sort

- Simple sorting algorithm that works similar to the way sorting playing cards in your hands.

- Array is splitted into sorted and unsorted part



Time Complexity ⟶

- Worst case ⟶ $O(N^2)$
- Average case ⟶ $O(N^2)$
- Best case ⟶ $O(N)$

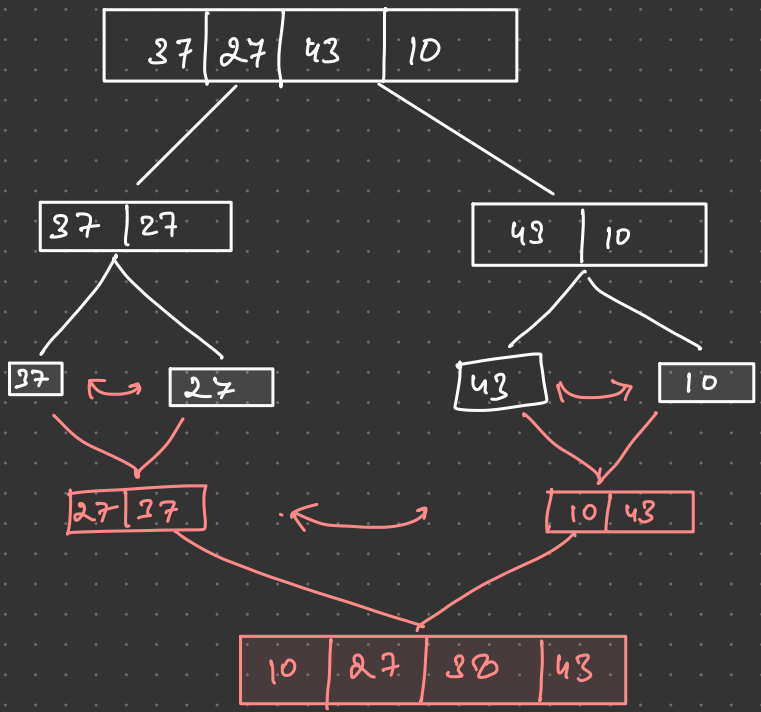
Auxiliary space ⟶ $O(1)$

- Follows incremental approach
- Insertion sort is in-place sorting algorithm
- Insertion sort is stable sorting algorithm

```
4  void insertionSort(vector<int>& nums)
5  {
6      for(int i=0;i<=nums.size()-1;i++)
7      {
8          int j=i;
9          while(j>0 && nums[j-1]>nums[j])
10         {
11             swap(nums[j-1],nums[j]);
12             j--;
13         }
14     }
15 }
```

→ Merge Sort

- Divide and conquer algorithm
- Divide
  ↓
  sort
  ↓
  merge



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

Time complexity → $O(N \log(N))$

Auxiliary Space → $O(N)$

- stable sorting algorithm
- Guaranteed performance
- Parallelizable
- not recommended for small datasets.

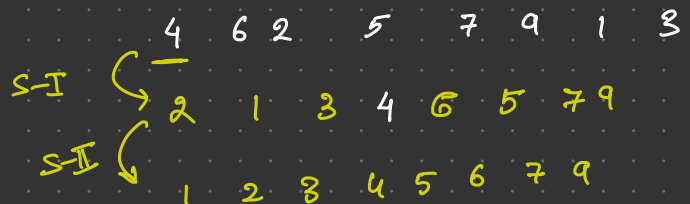
```cpp
void mergeSort(vector<int>& nums,int left,int right)
{
    if(left>=right)
    return;
    int mid=(left+right)/2;
    mergeSort(nums,left,mid);
    mergeSort(nums,mid+1,right);
    merge(nums,left,mid,right);
}
void merge(vector<int>& nums,int left,int mid,int right)
{
    int size=right-left+1;
    vector<int> temp(size,0);
    int i=left;
    int j=mid+1;
    int k=0;
    while(i<=mid && j<=right)
    {
        if(nums[i]<nums[j])
        temp[k++]=nums[i++];
        else
        temp[k++]=nums[j++];
    }
    while(i<=mid)
    temp[k++]=nums[i++];
    while(j<=right)
    temp[k++]=nums[j++];

    for(int i=left;i<=right;i++)
    {
        nums[i]=temp[i-left];
    }
}
```

– Quick Sort →

– Algorithm based on divide and conquer

– Picks a pivot and partition the given array around picked pivot by placing it in it's correct position.



Time Complexity →

Best case → $\Omega(N \cdot \log(N))$.

Average Case → $\Theta(N \log(N))$

Worst Case → $O(N^2)$.

Auxiliary space → $O(1)$

→ Not a stable algorithm

```
22
23  void quickSort(vector<int>& nums,int low,int high)
24  {
25      if(low<high)
26      {
27          int pivot_index=quick(nums,low,high);
28          quickSort(nums,low,pivot_index-1);
29          quickSort(nums,pivot_index+1,high);
30      }
31  }
```

```cpp
int quick(vector<int>& nums,int low,int high)
{
    int pivot=nums[low];//starting element as pivot
    int i=low;
    int j=high;
    while(i<j)
    {
        while(i<high && nums[i]<=pivot)
        i++;
        while(j>low && nums[j]>pivot)
        j--;
        if(i<j)
        swap(nums[i],nums[j]);
    }
    swap(nums[low],nums[j]);//swap to its correct position
    return j;
}
```

→ Heap sort →

- Sorting technique based on binary heap data structure.

Auxiliary space → $O(1) \cong O(\log N)$

Time complexity → $O(N \cdot \log(N))$

recursive stack space

- Typically not stable
- In place sorting algorithm.

```cpp
void heapSort(vector<int>& nums)
{
    int n=nums.size();
        for(int i=1;i<n;i++)
        {
            int ch=i;
            while(ch>0)
            {
                int par=(ch-1)/2;
                if(nums[par]<nums[ch])
                {
                    swap(nums[par],nums[ch]);
                }
                else
                break;

                ch=par;
            }
        }
        int size=n;
        while(size>1){
        swap(nums[0],nums[size-1]);
        size--;
        int par=0;
        int lef=par*2+1;
        int rig=par*2+2;
        while(lef<size)
        {
            int mini=par;
            if(nums[mini]<nums[lef])
            mini=lef;
            if(rig<size && nums[mini]<nums[rig])
            mini=rig;
            if(mini==par)
            break;

            swap(nums[mini],nums[par]);
            par=mini;
            lef=2*par+1;
            rig=2*par+2;
        }
        }
}
```