# Sorting algorithms

- Selection sort
- Bubble sort
- Quick sort
- Insertion sort
- Merge sort

# Sorting algorithms

- **Sorting** refers to arranging data in a particular format.
- **Sorting algorithm** specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.
- Sorting is also used to represent data in more readable formats.
- Following are some of the examples of sorting in real-life scenarios :
  - **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
  - **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

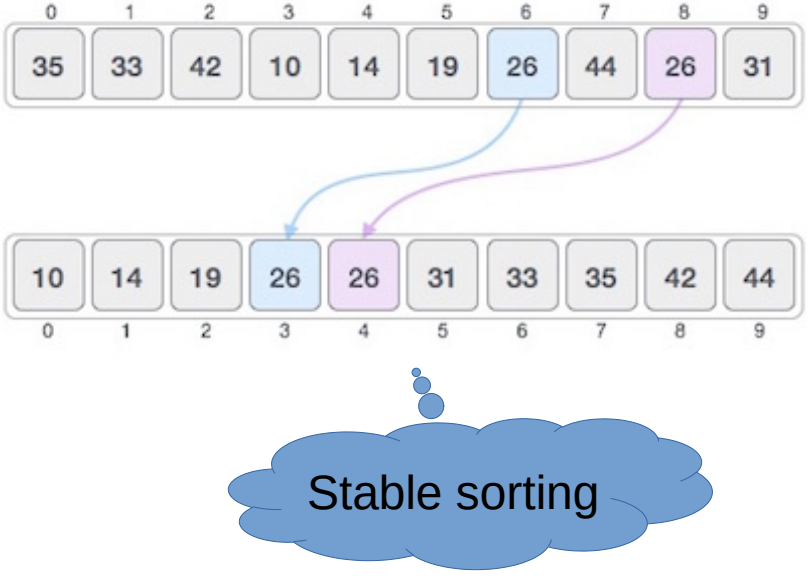
# Sorting algorithms

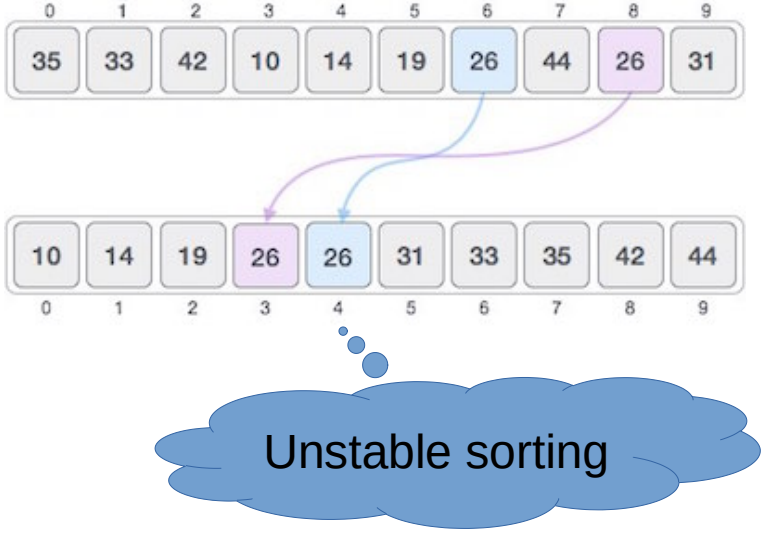
- **In-place Sorting and Not-in-place Sorting**
- Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.
- However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

# Sorting algorithms

- **Stable and Not Stable Sorting**

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.
- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.
- Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.



# Sorting algorithms

- Adaptive and Non-Adaptive Sorting Algorithm

- A sorting algorithm falls into the adaptive sort family if it takes advantage of existing order in its input.
- It benefits from the presortedness in the input sequence – or a limited amount of disorder for various definitions of measures of disorder – and sorts faster.
- Adaptive sort takes advantage of the existing order of the input to try to achieve better times, so that the time taken by the algorithm to sort is a smoothly growing function of the size of the sequence and the disorder in the sequence.
- The more presorted the input is, the faster it should be sorted
- A non-adaptive sorting algorithm does not consider the elements which are already sorted. They try to force every single element to be re-ordered to confirm their order.

# Sorting algorithms

- The complexity of sorting algorithm measures the running time of n items to be sorted.
- The operations in the sorting algorithm, where A1, A2 ….. An contains the items to be sorted and B is an auxiliary location, can be generalized as:
  - (a) Comparisons- which tests whether Ai < Aj or test whether Ai < B
  - (b) Interchange- which switches the contents of Ai and Aj or of Ai and B
  - (c) Assignments- which set B = A and then set Aj = B or Aj = Ai
- In general, the complexity functions measure only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

# Selection sort

- Selection sort is an in-place comparison sorting algorithm.
- Selection sort has an $O(n^2)$ time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort.
- Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited
- The algorithm divides the input list into two parts: a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.

# Selection sort

| Sorted sublist | Unsorted sublist | Least element in unsorted list |
|---|---|---|
| () | (11, 25, 12, 22, 64) | 11 |
| (11) | (25, 12, 22, 64) | 12 |
| (11, 12) | (25, 22, 64) | 22 |
| (11, 12, 22) | (25, 64) | 25 |
| (11, 12, 22, 25) | (64) | 64 |
| (11, 12, 22, 25, 64) | () | |

**Selection sort**

| | |
|---|---|
| **Class** | Sorting algorithm |
| **Data structure** | Array |
| **Worst-case performance** | $O(n^2)$ comparisons, $O(n)$ swaps |
| **Best-case performance** | $O(n^2)$ comparisons, $O(1)$ swap |
| **Average performance** | $O(n^2)$ comparisons, $O(n)$ swaps |
| **Worst-case space complexity** | $O(1)$ auxiliary |

- The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.
- The time efficiency of selection sort is quadratic, so there are a number of sorting techniques which have better time complexity than selection sort. One thing which distinguishes selection sort from other sorting algorithms is that it makes the minimum possible number of swaps, $n - 1$ in the worst case.

# Selection sort

```
void selection_sort(int a[],int N){
int i,j;//loop variables  advance the position through the entire array
for (i = 0; i < N-1; i++) // it doesn't work for a single element
{
    int jMin = i;// assume the min is the first element
    for (j = i+1; j < N; j++) // test against elements after i to find the
smallest
    {
        if (a[j] < a[jMin])// if this element is less, then it is the new minimum
        {
            jMin = j;// found new minimum; remember its index
        }
    }
    if (jMin != i) // if a smaller element found, swap it
    {
        swap(a[i], a[jMin]);
    }
    cout<<"i="<<i<<": ";print(a, N);
}
}
```

```
Initial array:
     11 5 1 3 8 9 4 7 5 2
i=0: 1 5 11 3 8 9 4 7 5 2
i=1: 1 2 11 3 8 9 4 7 5 5
i=2: 1 2 3 11 8 9 4 7 5 5
i=3: 1 2 3 4 8 9 11 7 5 5
i=4: 1 2 3 4 5 9 11 7 8 5
i=5: 1 2 3 4 5 5 11 7 8 9
i=6: 1 2 3 4 5 5 7 11 8 9
i=7: 1 2 3 4 5 5 7 8 11 9
i=8: 1 2 3 4 5 5 7 8 9 11

Sorted array:
     1 2 3 4 5 5 7 8 9 11
```

# Bubble sort

- Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
- The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.
- This simple algorithm performs poorly in real world use and is used primarily as an educational tool.
- Bubble sort has a worst-case and average complexity of $O(n^2)$, where n is the number of items being sorted.
- Most practical sorting algorithms have substantially better worst-case or average complexity, often O(n log n).
- Even other $O(n^2)$ sorting algorithms, such as insertion sort, generally run faster than bubble sort, and are no more complex. Therefore, bubble sort is not a practical sorting algorithm.

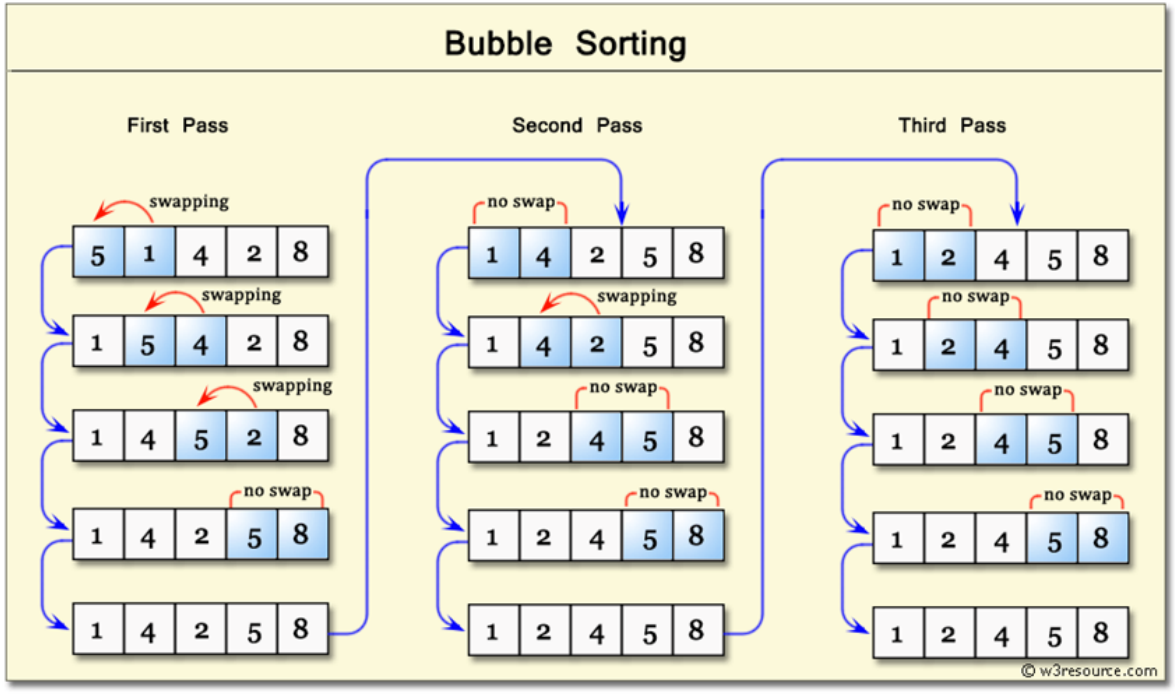
| | |
|---|---|
| **Class** | Sorting algorithm |
| **Data structure** | Array |
| **Worst-case performance** | $O(n^2)$ comparisons, $O(n^2)$ swaps |
| **Best-case performance** | $O(n)$ comparisons, $O(1)$ swaps |
| **Average performance** | $O(n^2)$ comparisons, $O(n^2)$ swaps |
| **Worst-case space complexity** | $O(n)$ total, $O(1)$ auxiliary |

Sorting In Place: Yes
Stable: Yes

Source :https://en.wikipedia.org/wiki/Bubble_sort

# Bubble sort

- Example: Take an array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared.




www.w3resource.com

# Bubble sort

```
#include <iostream>
#include <algorithm> // for swap functoin
using namespace std;

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
          // Last i elements are already in
      place
          for (j = 0; j < n-i-1; j++)
              if (arr[j] > arr[j+1])
                  swap(arr[j], arr[j+1]);
      }

      /* Function to print an array */
      void printArray(int arr[], int size)
      {
          int i;
          for (i = 0; i < size; i++)
              cout << arr[i] << " ";
          cout << endl;
      }

      // Driver code
      int main()
      {
          int arr[] = {64, 34, 25, 12, 22, 11,
      90};
          int n = sizeof(arr)/sizeof(arr[0]);
          bubbleSort(arr, n);
          cout<<"Sorted array: \n";
          printArray(arr, n);
          return 0;
```

The basic implementation function always runs $O(n^2)$ time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn't cause any swap. When the list is already sorted (best-case), the complexity is only O(n) for the best case.

```
// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
        if (arr[j] > arr[j+1])
        {
           swap(arr[j], arr[j+1]);
           swapped = true;
        }
     }

     //IF no two elements were swapped by inner loop, then
break
     if (swapped == false)
        break;
   }
}
```

Source: https://www.geeksforgeeks.org/bubble-sort/

# Quick sort

- Quicksort is an in-place sorting algorithm.
- When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort
- Quicksort is a divide-and-conquer algorithm.
- It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting. Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.
- Mathematical analysis of quicksort shows that, on average, the algorithm takes O(nlog n) comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons.

| Class | Sorting algorithm |
|---|---|
| Worst-case performance | $O(n^2)$ |
| Best-case performance | $O(n \log n)$ (simple partition) or $O(n)$ (three-way partition and equal keys) |
| Average performance | $O(n \log n)$ |
| Worst-case space complexity | $O(n)$ auxiliary (naive) $O(\log n)$ auxiliary |

https://en.wikipedia.org/wiki/Quicksort

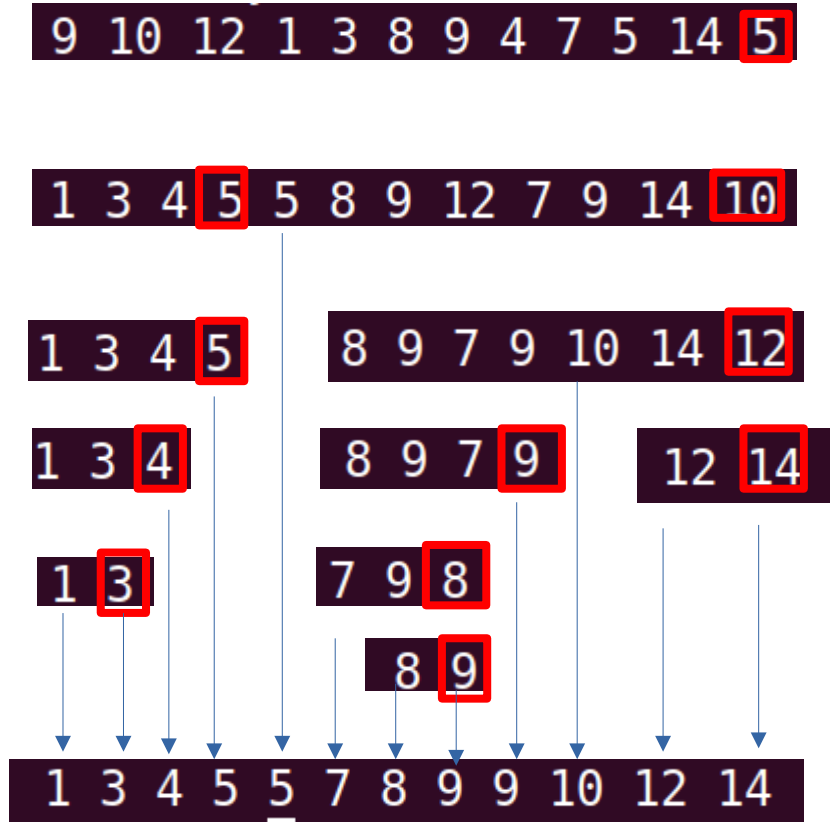
# Quick sort

```
int partition(int arr[], int low, int high) {
	int pivot = arr[high];    // pivot
	int i = (low - 1);  // Index of smaller element

	for (int j = low; j <= high - 1; j++) {
		// If current element is smaller than or equal to pivot
		if (arr[j] <= pivot) {
		i++;    // increment index of smaller element
		swap(arr[i], arr[j]);
		}
	}
	swap(arr[i + 1], arr[high]);
	return (i + 1);
	}

	void quickSort(int arr[], int low, int high) {
	if (low < high) {
		int p = partition(arr, low, high);/* p is partitioning index, arr[p] is now at right place */
		quickSort(arr, low, p - 1); // Separately sort elements before p and after p
		quickSort(arr, p + 1, high);
	}
	}

	int main() {
	int arr[] = {9, 11, 5, 1, 3, 8, 9, 4, 7, 5, 6,9 };
	int N = sizeof(arr) / sizeof(arr[0]); // the number of elements
	cout << "Initial array:" << endl;
	cout << "     ";print(arr, N);
	quickSort(arr, 0, N - 1);
	cout << "\nSorted array:" << endl;
	cout << "     ";print(arr, N);
	return 0;
	}
```

en yüksek veya en düşük eleman pivot olarak seçilebilir

# Quick sort example

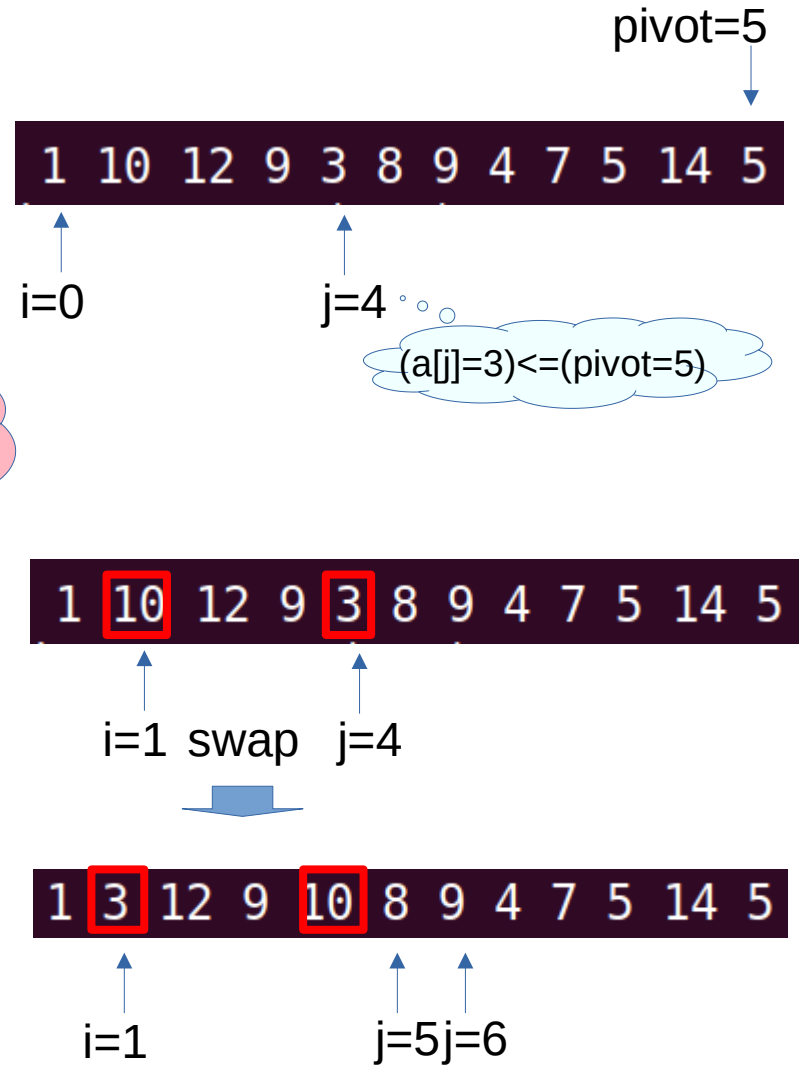
```
ubuntu@ubuntu: ~/Documen
ubuntu:code$ g++ quicksort.cpp -o prog
ubuntu:code$ ./prog
Initial array:
     9 10 12 1 3 8 9 4 7 5 14 5
pivot=5      1 3 4 5 5 8 9 12 7 9 14 10
pivot=5      1 3 4 5
pivot=4      1 3 4
pivot=3      1 3
pivot=10      8 9 7 9 10 14 12
pivot=9      8 9 7 9
pivot=7      7 9 8
pivot=8      8 9
pivot=12      12 14

Sorted array:
     1 3 4 5 5 7 8 9 9 10 12 14
ubuntu:code$
```



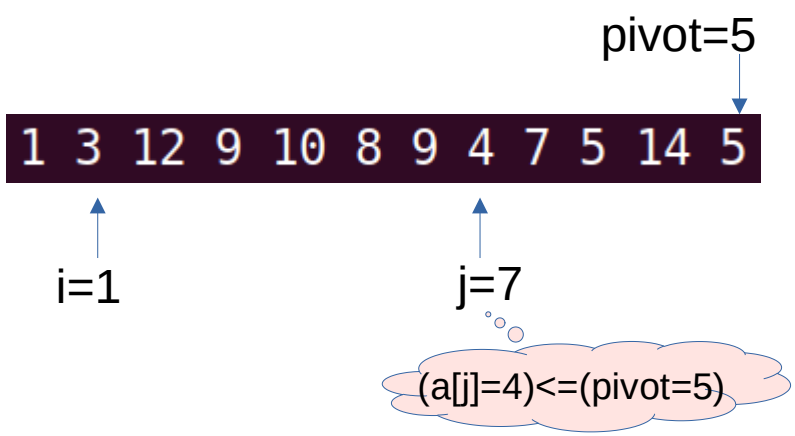
# Quick sort: an example partition

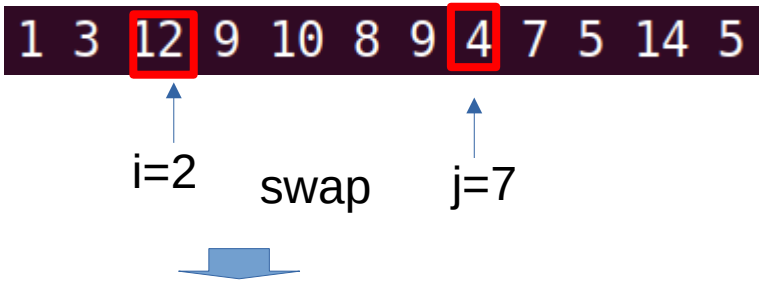
```
ubuntu:code$ g++ quicksort.cpp -o prog
ubuntu:code$ ./prog
Initial array:
     9 10 12 1 3 8 9 4 7 5 14 5
pivot=5
i=-1
i=-1  j=0
i=-1  j=1
i=-1  j=2
arr[j] <= pivot : swap(1,9)
i=0  j=3
arr[j] <= pivot : swap(3,10)
i=1  j=4
i=1  j=5
i=1  j=6
arr[j] <= pivot : swap(4,12)
i=2  j=7
i=2  j=8
arr[j] <= pivot : swap(5,9)
i=3  j=9
i=3  j=10
pivot=5     1 3 4 5 5 8 9 12 7 9 14 10
----------------------------------------
pivot=5
i=-1
```

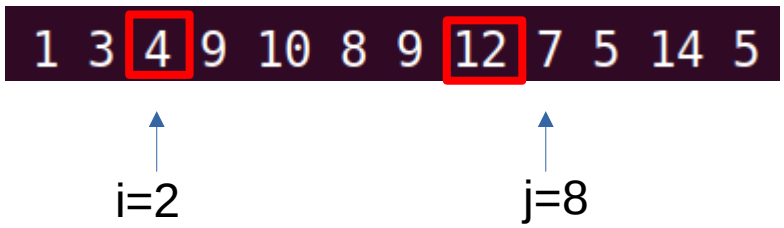
Example array paritition

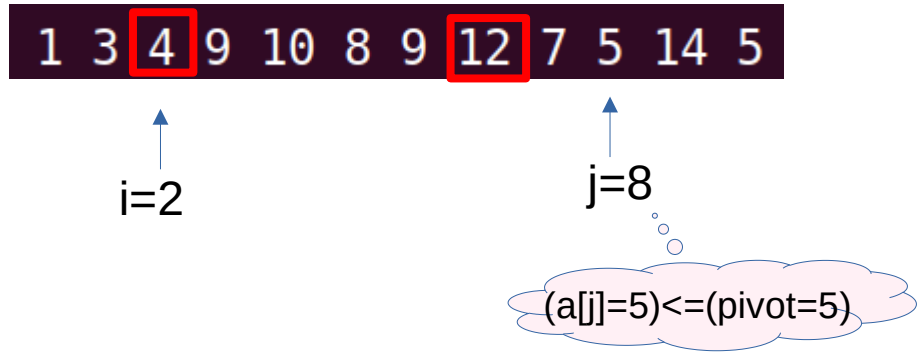


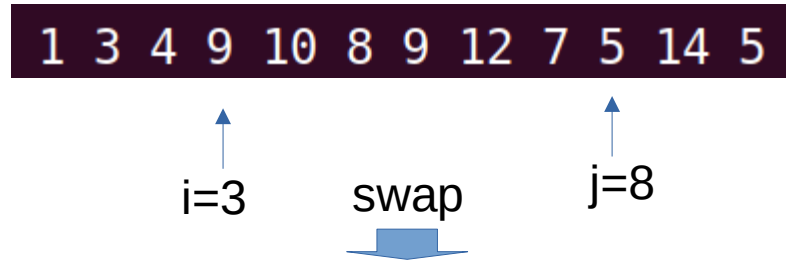
# Quick sort: an example partition

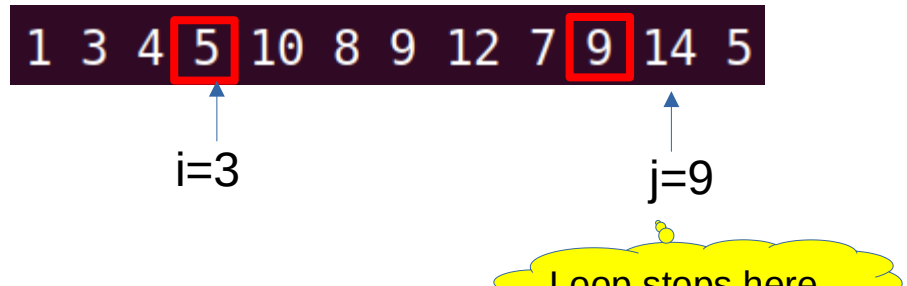


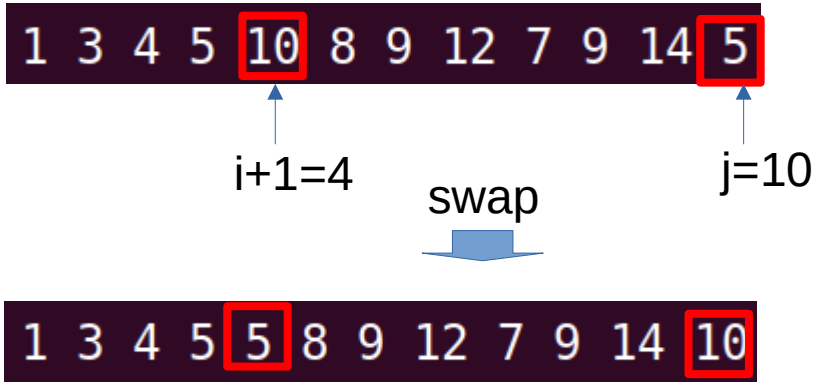



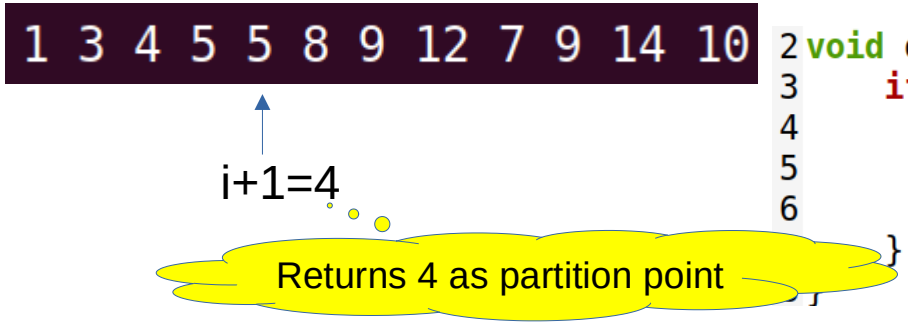



# Quick sort: an example partition




```
2 int partition(int arr[], int low, int high) {
3     int pivot = arr[high];
4     for (int j = low; j <= high - 1; j++) {
5         if (arr[j] <= pivot) {
6             i++;
7             swap(arr[i], arr[j]);
8         }
9     }
10    swap(arr[i + 1], arr[high])
11    return (i + 1);
12 }
13
```




```
2 void quickSort(int arr[], int low, int high) {
3     if (low < high) {
4         int p = partition(arr, low, high);
5         quickSort(arr, low, p - 1);
6         quickSort(arr, p + 1, high);
      }
```

```
quickSort(arr, 0, 3);
quickSort(arr, 5, 10);
```

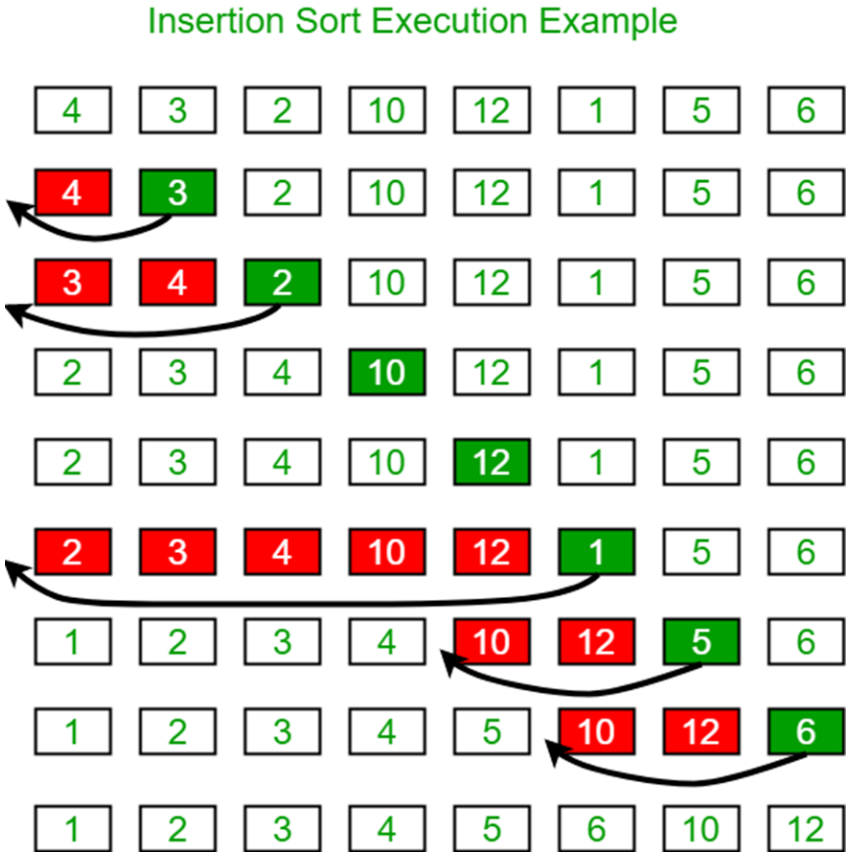
Repeat for the subarrays

# Insertion Sort

- Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Jon Bentley shows a three-line C version, and a five-line optimized version Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., O(n2)) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is O(kn) when each element in the input is no more than k places away from its sorted position
- Stable; it does not change the relative order of elements with equal keys
- In-place; it only requires a constant amount O(1) of additional memory space
- Online; it can sort a list as it receives it

# Insertion Sort




Pseudo code

```
i ← 1
while i < length(A)
    x ← A[i]
    j ← i - 1
    while j >= 0 and A[j] > x
        A[j+1] ← A[j]
        j ← j - 1
    end while
    A[j+1] ← x
    i ← i + 1
end while
```

# Insertion Sort

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
       //cout<<"i="<<i<<": ";print(arr,
n);
    }
}
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout<<"initial array:"<<endl;
    cout<<"i=0: ";print(arr, n);
    insertionSort(arr, n);
    cout<<"\nsorted array:"<<endl;
    print(arr, n);
    return 0;
}
```

```
<terminated> (exit value: 0) w14_ir
initial array:
i=0: 12 11 13 5 6
i=1: 11 12 13 5 6
i=2: 11 12 13 5 6
i=3: 5 11 12 13 6
i=4: 5 6 11 12 13

sorted array:
5 6 11 12 13
```

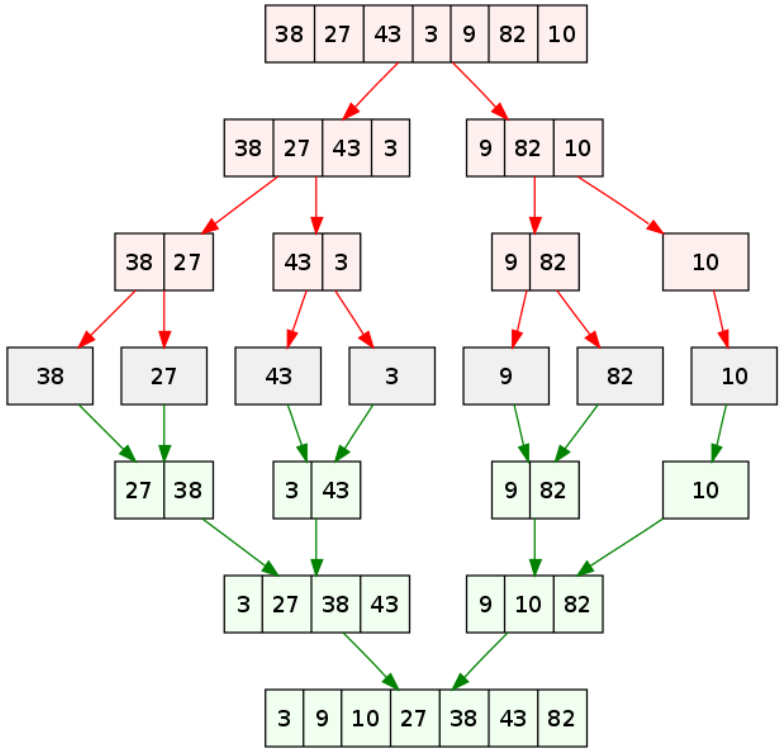
# Merge sort

- In computer science, merge sort is an efficient, general-purpose, and comparison-based sorting algorithm.
- Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output.
- Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.
- Conceptually, a merge sort works as follows:
  - Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
  - Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

# Merge sort

## Example code top down algorithm

```
// Array A[] has the items to sort; array B[] is a work array.
void TopDownMergeSort(A[], B[], n)
{
    CopyArray(A, 0, n, B);           // one time copy of A[] to B[]
    TopDownSplitMerge(B, 0, n, A);   // sort data from B[] into A[]
}

// Split A[] into 2 runs, sort both runs into B[], merge both runs from B[] to A[]
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
void TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if (iEnd - iBegin <= 1)                     // if run size == 1
        return;                                 //   consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}

//  Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1   ].
// Result is            B[ iBegin:iEnd-1   ].
void TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i = iBegin, j = iMiddle;

    // While there are elements in the left or right runs...
    for (k = iBegin; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}
```



# Merge sort

- In sorting n objects, merge sort has an average and worst-case performance of O(nlogn).
- If the running time of merge sort for a list of length n is T(n), then the recurrence relation
- T(n) = 2T(n/2) + n follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists
- Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

| | |
|---|---|
| **Class** | Sorting algorithm |
| **Data structure** | Array |
| **Worst-case performance** | $O(n \log n)$ |
| **Best-case performance** | $\Omega(n \log n)$ typical, $\Omega(n)$ natural variant |
| **Average performance** | $\Theta(n \log n)$ |
| **Worst-case space complexity** | $O(n)$ total with $O(n)$ auxiliary, $O(1)$ auxiliary with linked lists[1] |

# Merge sort

```
// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {

// cout << "(" << l << "," << r << ")=";print(arr, l, r);
if (l < r) {
// m is thepoint where the array is divided into two subarrays
int m = l + (r - l) / 2;

cout << "1- mergeSort(arr," << l << "," << m << "):" << endl;
//cout<<"("<<l<<","<<m<<")";print(arr,l,m);
mergeSort(arr, l, m);
cout << "2- mergeSort(arr," << m + 1 << "," << r << "):" << endl;
//cout<<"("<<m+1<<","<<r<<")";print(arr,m + 1, r);
mergeSort(arr, m + 1, r);

// Merge the sorted subarrays
merge(arr, l, m, r);
}
}
```

# Merge sort

```
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {
// Create L ← A[p..q] and M ← A[q+1..r]
int n1 = q - p + 1;
int n2 = r - q;
int L[n1], M[n2];
for (int i = 0; i < n1; i++)
      L[i] = arr[p + i];
for (int j = 0; j < n2; j++)
      M[j] = arr[q + 1 + j];
// Maintain current index of sub-arrays and main array
int i, j, k;
i = 0;
j = 0;
k = p;
// Until we reach either end of either L or M, pick larger among
// elements L and M and place them in the correct position at
A[p..r]
while (i < n1 && j < n2) {
      if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
      } else {
            arr[k] = M[j];
            j++;
      }
      k++;
      }
      // When we run out of elements in either L or M,
      // pick up the remaining elements and put in A[p..r]
      while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
      }
      while (j < n2) {
            arr[k] = M[j];
            j++;
            k++;
      }
      }
```

```
// Driver program
int main() {
int arr[] = { 6, 5, 12, 10, 9, 1 };
int size = sizeof(arr) /
sizeof(arr[0]);
mergeSort(arr, 0, size - 1);
cout << "Sorted array: \n";
printArray(arr, size);
return 0;
}
```

```
<terminated> (exit value: 0) w14_mergesort.e
(0,5)=6 5 12 10 9 1
1- mergeSort(arr,0,2):
(0,2)=6 5 12
1- mergeSort(arr,0,1):
(0,1)=6 5
1- mergeSort(arr,0,0):
(0,0)=6
2- mergeSort(arr,1,1):
(1,1)=5
2- mergeSort(arr,2,2):
(2,2)=12
2- mergeSort(arr,3,5):
(3,5)=10 9 1
1- mergeSort(arr,3,4):
(3,4)=10 9
1- mergeSort(arr,3,3):
(3,3)=10
2- mergeSort(arr,4,4):
(4,4)=9
2- mergeSort(arr,5,5):
(5,5)=1
Sorted array:
1 5 6 9 10 12
```