Data Structures and Algorithms

Sorting Techniques

Engr. Bushra Tahir
Department of Electrical Engineering
Iqra National University

Introduction

- Sorting refers to arranging a set of data in some logical order
- For ex. A telephone directory can be considered as a list where each record has three fields - name, address and phone number.
- Being unique, phone number can work as a key to locate any record in the list.

Introduction

- Sorting is among the most basic problems in algorithm design.

- We are given a sequence of items, each associated with a given key value. And the problem is to rearrange the items so that they are in an increasing(or decreasing) order by key.

- The methods of sorting can be divided into two categories:
  - Internal Sorting
  - External Sorting

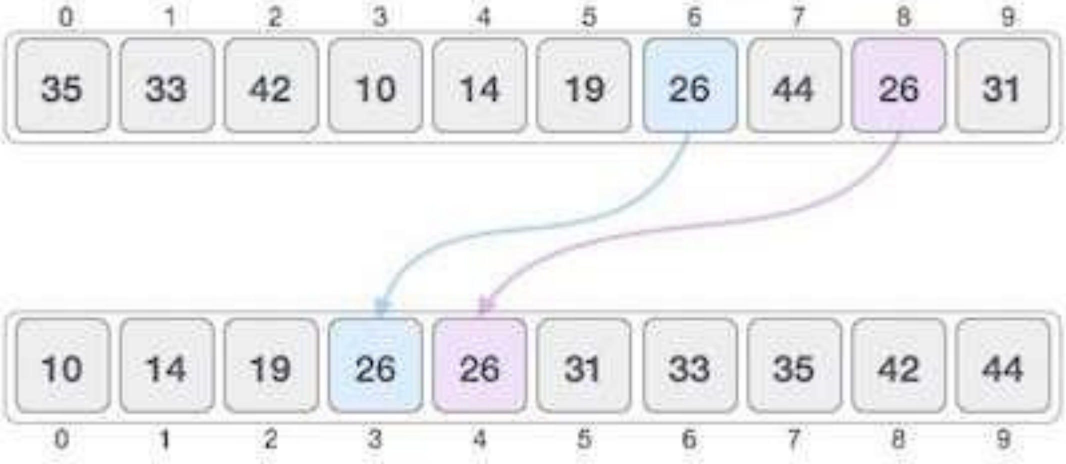
- Internal Sorting
  - ✓ If all the data that is to be sorted can be adjusted at a time in main memory, then internal sorting methods are used

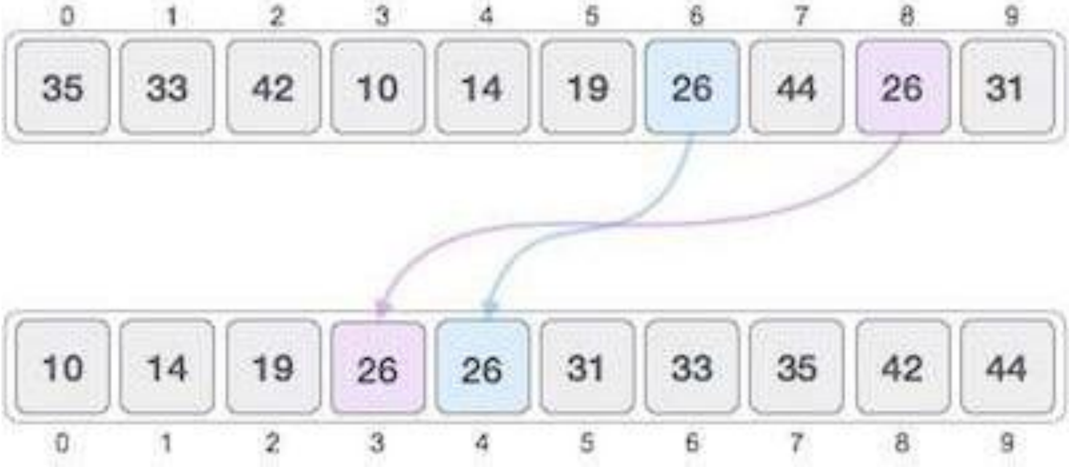
- External Sorting
  - ✓ When the data to be sorted can't be accommodated in the memory at the same time and some has to be kept in auxiliary memory, then external sorting methods are used.

❖ NOTE: We will only consider internal sorting

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.

Efficiency of Sorting Algorithm

- The complexity of a sorting algorithm measures the running time of a function in which n number of items are to be sorted.

- The choice of sorting method depends on efficiency considerations for different problems.

- Three most important of these considerations are:
  - The length of time spent by programmer in coding a particular sorting program
  - Amount of machine time necessary for running the program
  - The amount of memory necessary for running the program

Efficiency of Sorting Algorithm

- Various sorting methods are analyzed in the cases like – best case, worst case or average case.

- Most of the sort methods we consider have requirements that range from 0(nlogn) to 0(n^2).

- A sort should not be selected only because its sorting time is 0(nlogn); the relation of the file size n and the other factors affecting the actual sorting time must be considered

Efficiency of Sorting Algorithm

- Determining the time requirement of sorting technique is to actually run the program and measure its efficiency.

- Once a particular sorting technique is selected the need is to make the program as efficient as possible.

- Any improvement in sorting time significantly affect the overall efficiency and saves a great deal of computer time.

Efficiency of Sorting Algorithm

- Space constraints are usually less important than time considerations.

- The reason for this can be, as for most sorting programs, the amount of space needed is closer to $0(n)$ than to $0(n^2)$

- The second reason is that, if more space is required, it can almost always be found in auxiliary storage.

BUBBLE SORT

- In bubble sort, each element is compared with its adjacent element.
- We begin with the 0th element and compare it with the 1st element.
- If it is found to be greater than the 1st element, then they are interchanged.
- In this way all the elements are compared (excluding last) with their next element and are interchanged if required
- On completing the first iteration, largest element gets placed at the last position. Similarly in second iteration second largest element gets placed at the second last position and so on.

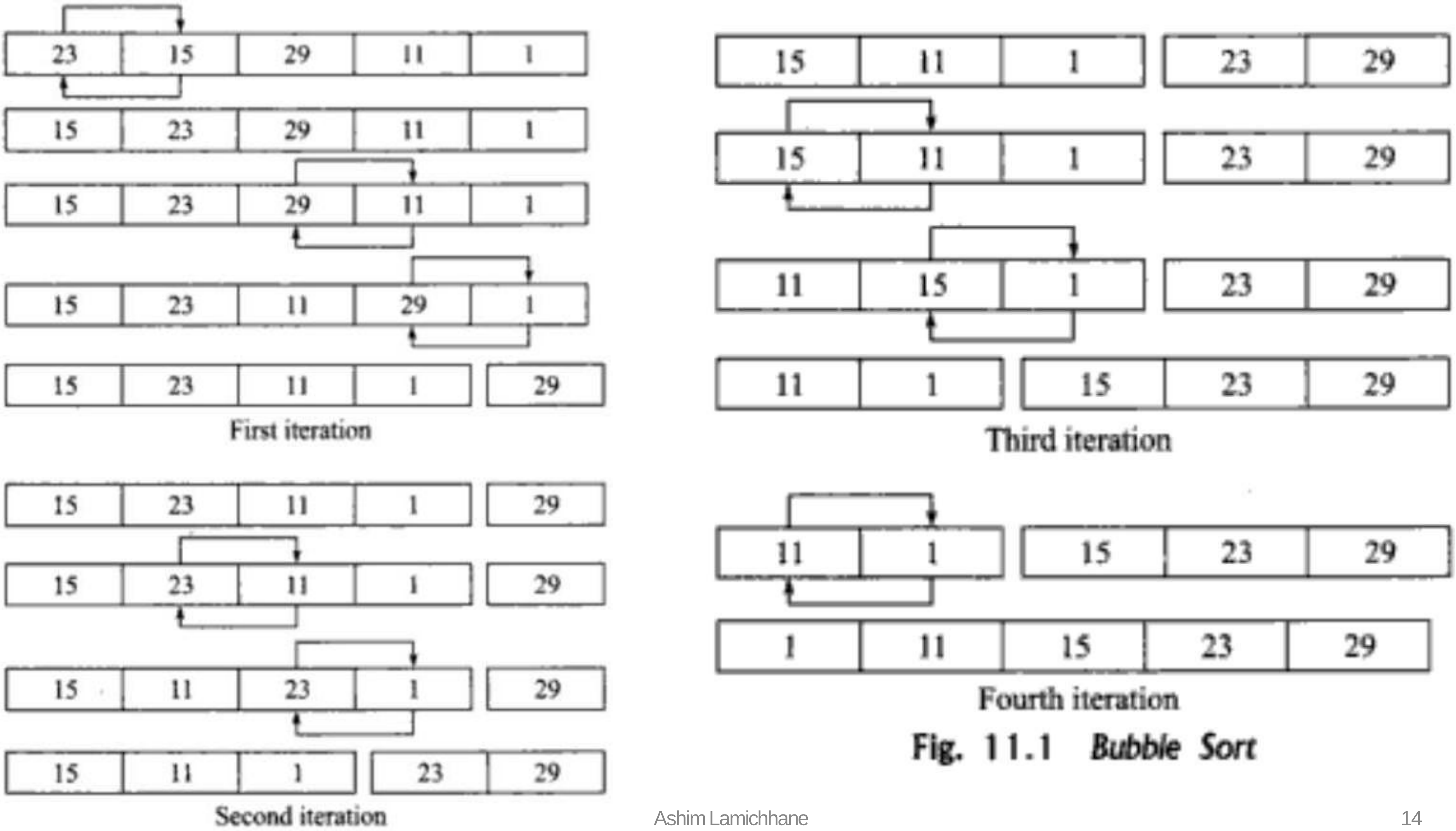


Fig. 11.1 *Bubble Sort*

Algorithm

```
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}
```

Algorithm

```
int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

TIME COMPLEXITY

- The time complexity for bubble sort is calculated in terms of the number of comparisons f(n) (or of number of loops)
- Here two loops(outer loop and inner loop) iterates(or repeated) the comparison.
- The inner loop is iterated one less than the number of elements in the list (i.e., n-1 times) and is reiterated upon every iteration of the outer loop

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

$$= n(n-1) = O(n2).$$

TIME COMPLEXITY

- Best Case
  - sorting a sorted array by bubble sort algorithm
  - In best case outer loop will terminate after one iteration, i.e it involves performing one pass which requires n-1 comparison

$$f\ (n) = O(n^2)$$

- Worst Case
  - Suppose an array [5,4,3,2,1], we need to move first element to end of an array
  - n-1 times the swapping procedure is to be called

$$f\ (n) = O(n^2)$$

- Average Case
  - Difficult to analyse than the other cases
  - Random inputs, so in general

$$f\ (n) = O(n^2)$$

- Space Complexity
  - O(n)

SELECTION SORT

- Find the least(or greatest) value in the array, swap it into the leftmost(or rightmost) component, and then forget the leftmost component, Do this repeatedly.

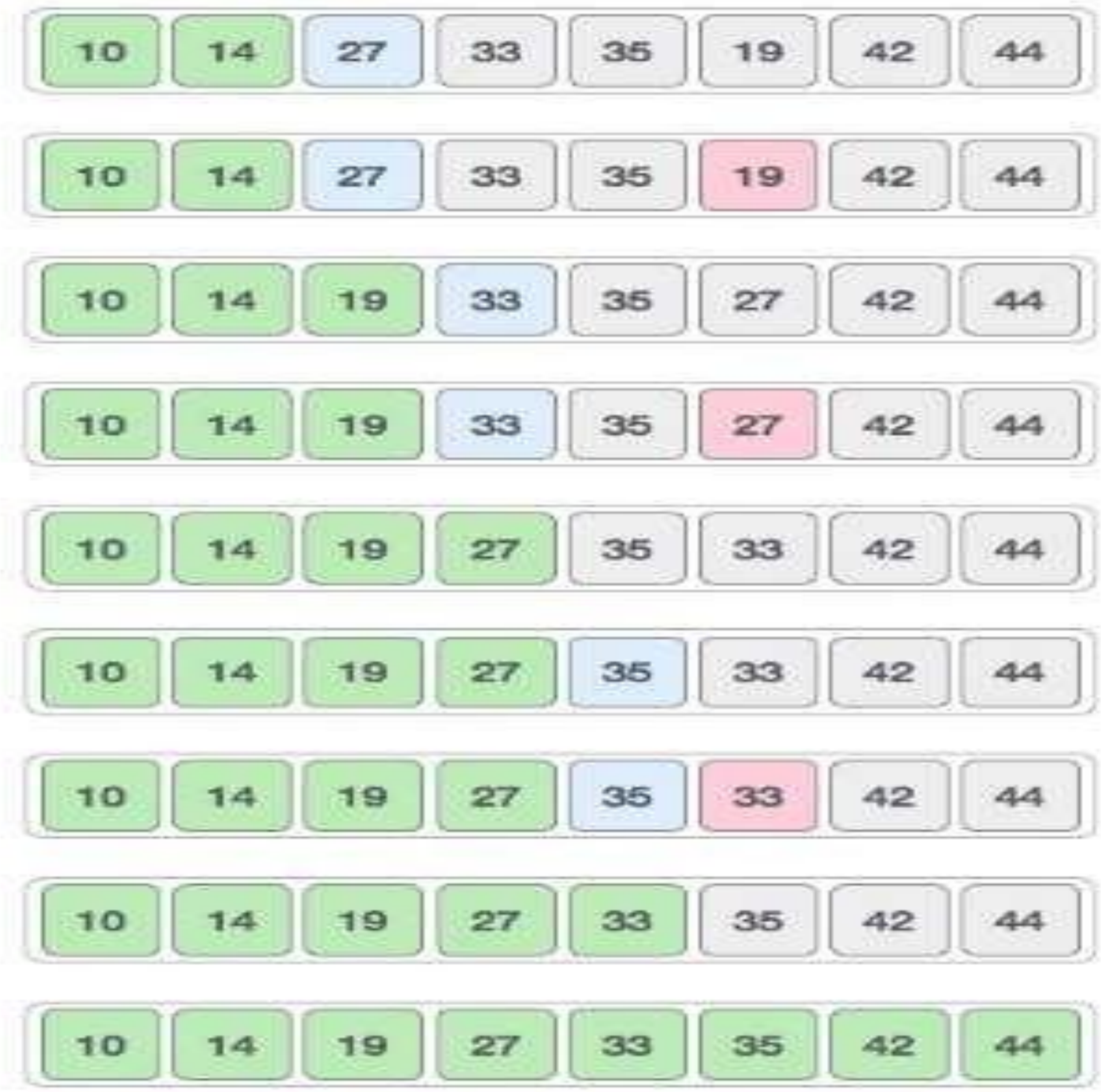
- Let a[n] be a linear array of n elements. The selection sort works asfollows:

- Pass 1: Find the location loc of the smallest element in the list of n elements a[0], a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[0].

- Pass 2: Find the location loc of the smallest element int the sub-list of n-1 elements a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted.

- Then we will get the sorted list

  a[0]<=a[2]<=a[3]…....<=a[n-1]


10 14 27 33 35 19 42 44
10 14 27 33 35 19 42 44
10 14 19 33 35 27 42 44
10 14 19 33 35 27 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 33 35 42 44
10 14 19 27 33 35 42 44

Algorithm:

```
SelectionSort(A)
{
        for( i = 0;i < n ;i++)
        {
                least=A[i];
                p=i;
                for ( j = i + 1;j < n ;j++)
                {
                        if (A[j] < A[i])
                        least= A[j]; p=j;
                }
        }
        swap(A[i],A[p]);
}
```

Time Complexity

- Inner loop executes (n-1) times when i=0, (n-2) times when i=1 and so on:
- Time complexity = (n-1) + (n-2) + (n-3) + …........ +2+1

  = $O(n^2)$

Space Complexity

- Since no extra space beside n variables is needed for sorting so
- O(n)

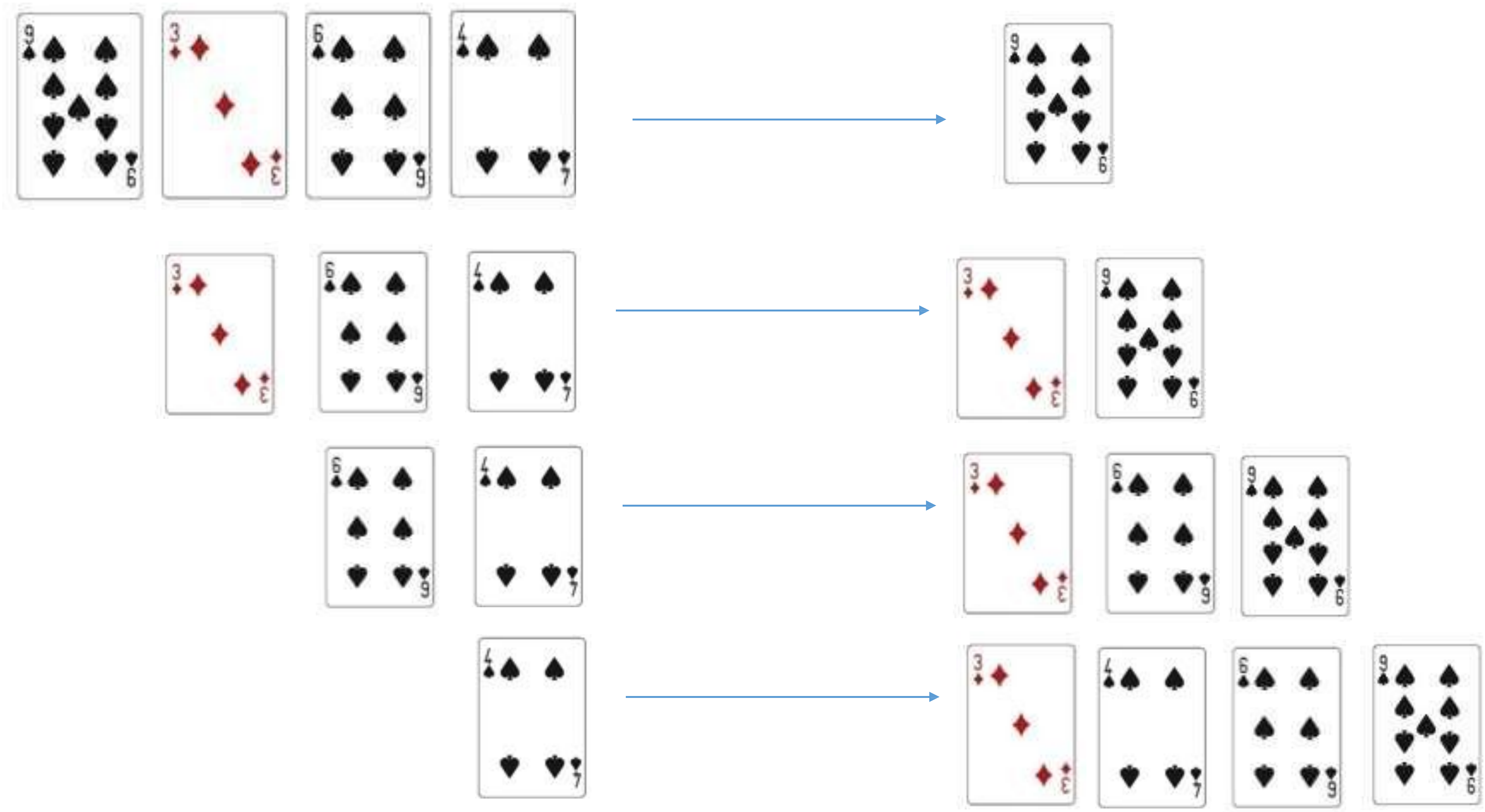
Insertion Sort

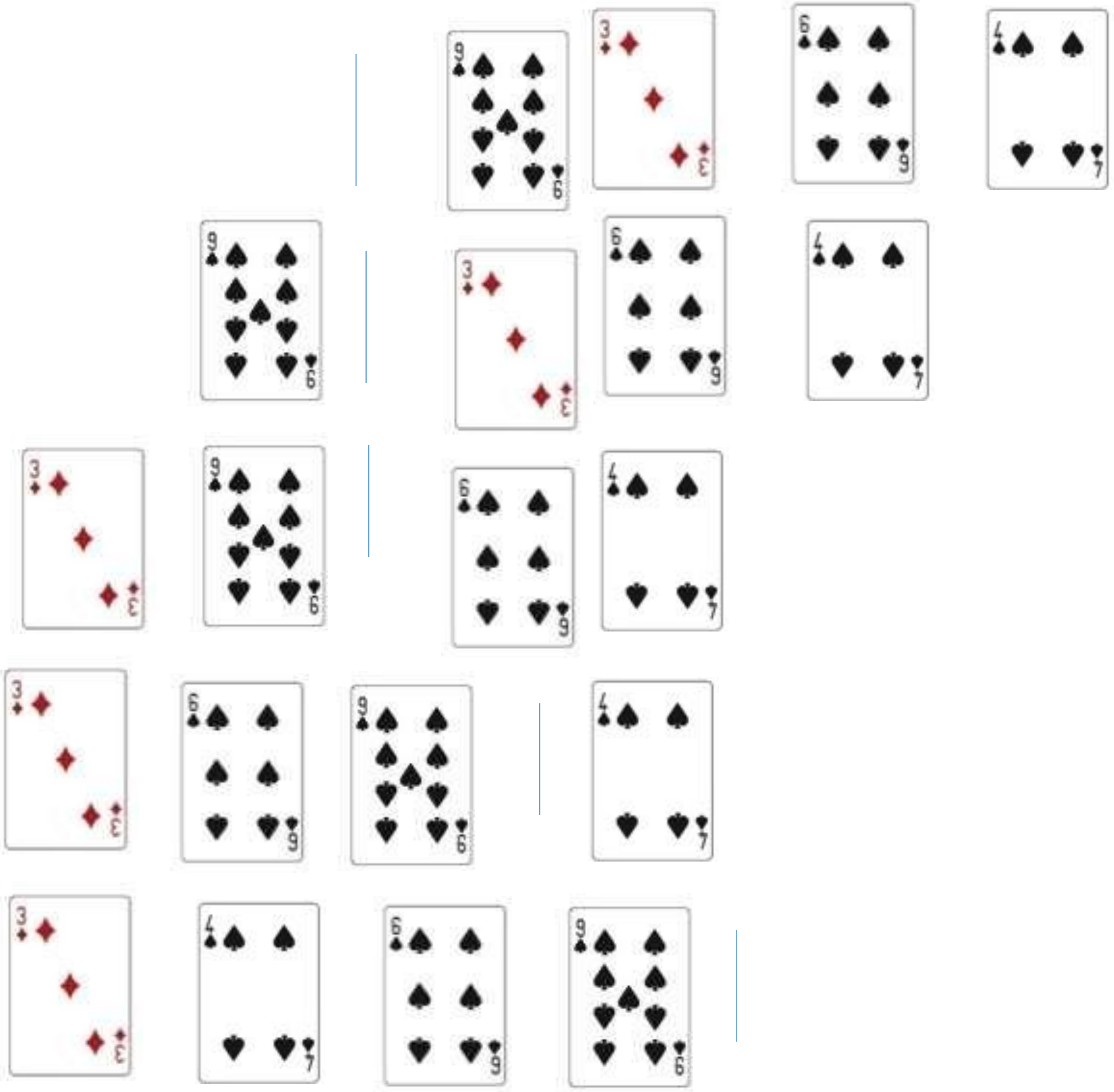
- Like sorting a hand of playing cards start with an empty hand and the cards facing down the table.

- Pick one card at a time from the table, and insert it into the correct position in the left hand.

- Compare it with each of the cards already in the hand, from right to left

- The cards held in the left hand are sorted.





Insertion Sort

- Suppose an array a[n] with n elements. The insertion sort works asfollows:

Pass 1: a[0] by itself is trivially sorted.

Pass 2: a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.

Pass 3: a[2] is inserted into its proper place in a[0],a[1] that is before a[0], between a[0] and a[1], or after a[1] so that a[0],a[1],a[2] is sorted.

pass N: a[n-1] is inserted into its proper place in a[0],a[1],a[2],........,a[n-2] so that a[0],a[1],a[2],............,a[n-1] is sorted with n elements.

7	2	4	5	1	3

2	7	4	5	1	3

2	4	7	5	1	3

2	4	5	7	1	3

1	2	4	5	7	3

1	2	3	4	5	7

Algorithm

7	2	4	1	5	3

1st Pass

7	7	4	1	5	3

2	7	4	1	5	3

i	value	hole
1	2	1
1	2	0

```
InsertionSort(){
        for (i=1;i<n;i++){
                value=C[ i ];
                hole= i ;
                while(hole>0 && C[hole-1]>value){
                        C[hole]=C[hole-1];
                        hole=hole-1;
                }
                C[hole]=value;
        }
}
```

Time Complexity

- Best Case:
  - If the array is all but sorted then
  - Inner Loop wont execute so only some constant time the statements will run
  - So Time complexity= O(n)

- Worst Case:
  - Array element in reverse sorted order
  - Time complexity=O(n^2)

- Space Complexity
  - Since no extra space beside n variables is needed for sorting so
  - Space Complexity = O(n)

Divide and conquer algorithms

- The sorting algorithms we’ve seen so far have worst-case running times of $O(n^2)$

- When the size of the input array is large, these algorithms can take a long time to run.

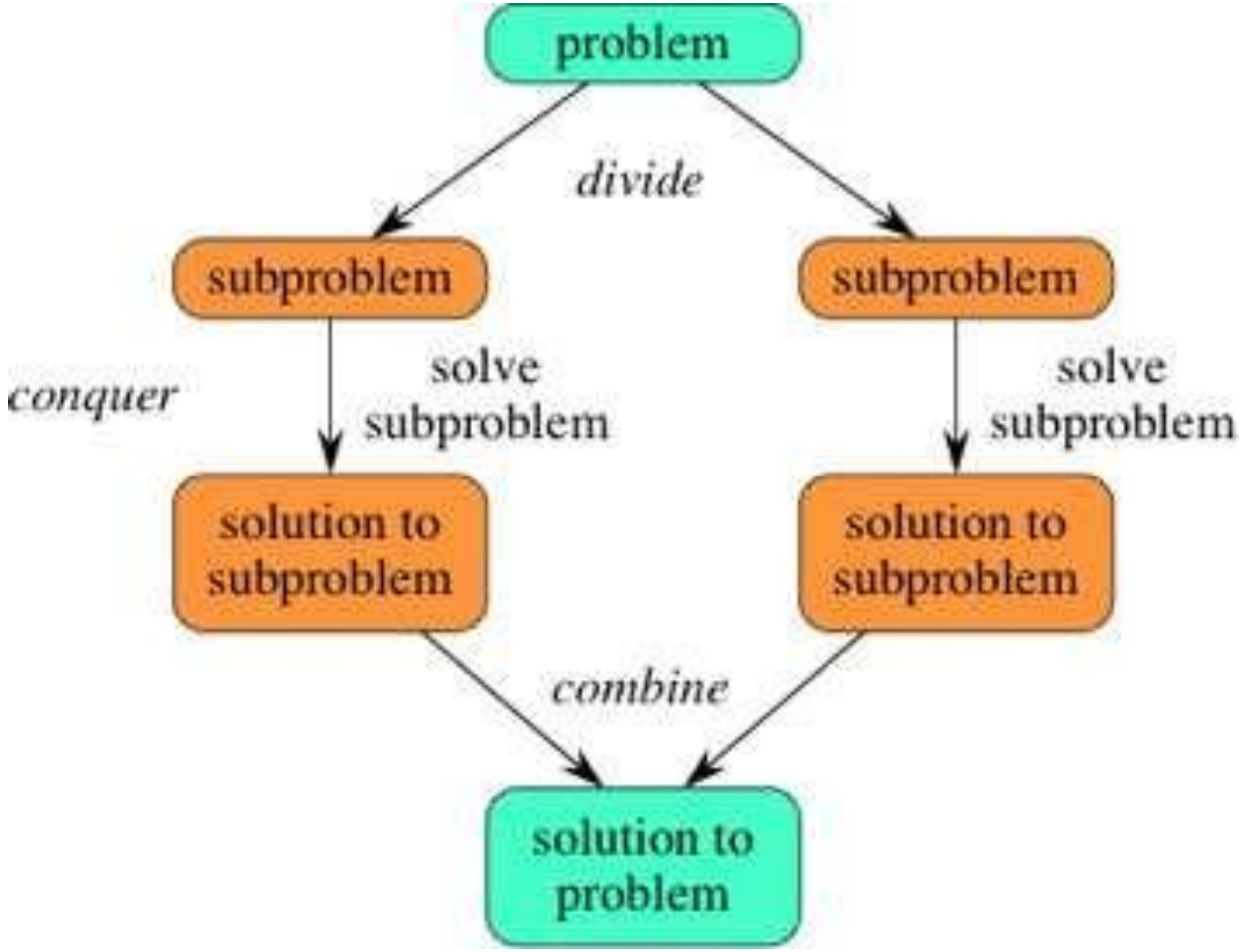
- Now we will discuss two sorting algorithms whose running times are better
  - Merge Sort
  - Quick Sort

Divide-and-conquer

- Divide-and-conquer, breaks a problem into sub problems that are similar to the original problem, recursively solves the sub problems, and finally combines the solutions to the sub problems to solve the original problem.

- Think of a divide-and-conquer algorithm as having three parts:
  - Divide the problem into a number of subproblems that are smaller instances of the same problem.
  - Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
  - Combine the solutions to the subproblems into the solution for the original problem.

Divide-and-conquer



Merge Sort

- Merge sort is a sorting technique based on divide and conquer technique.

- Merge sort first divides the array into equal halves and then combines them in a sorted manner.

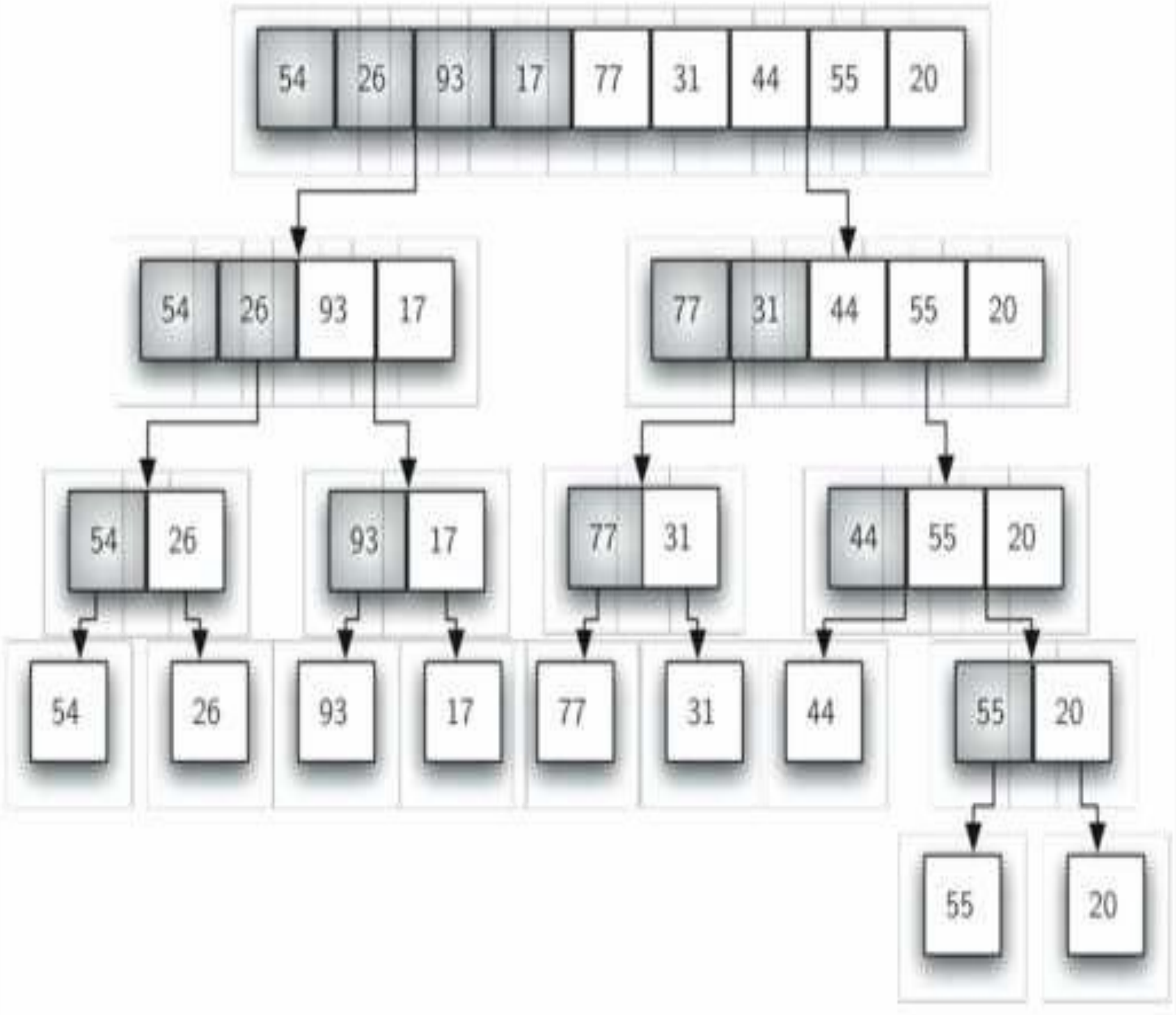
- With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

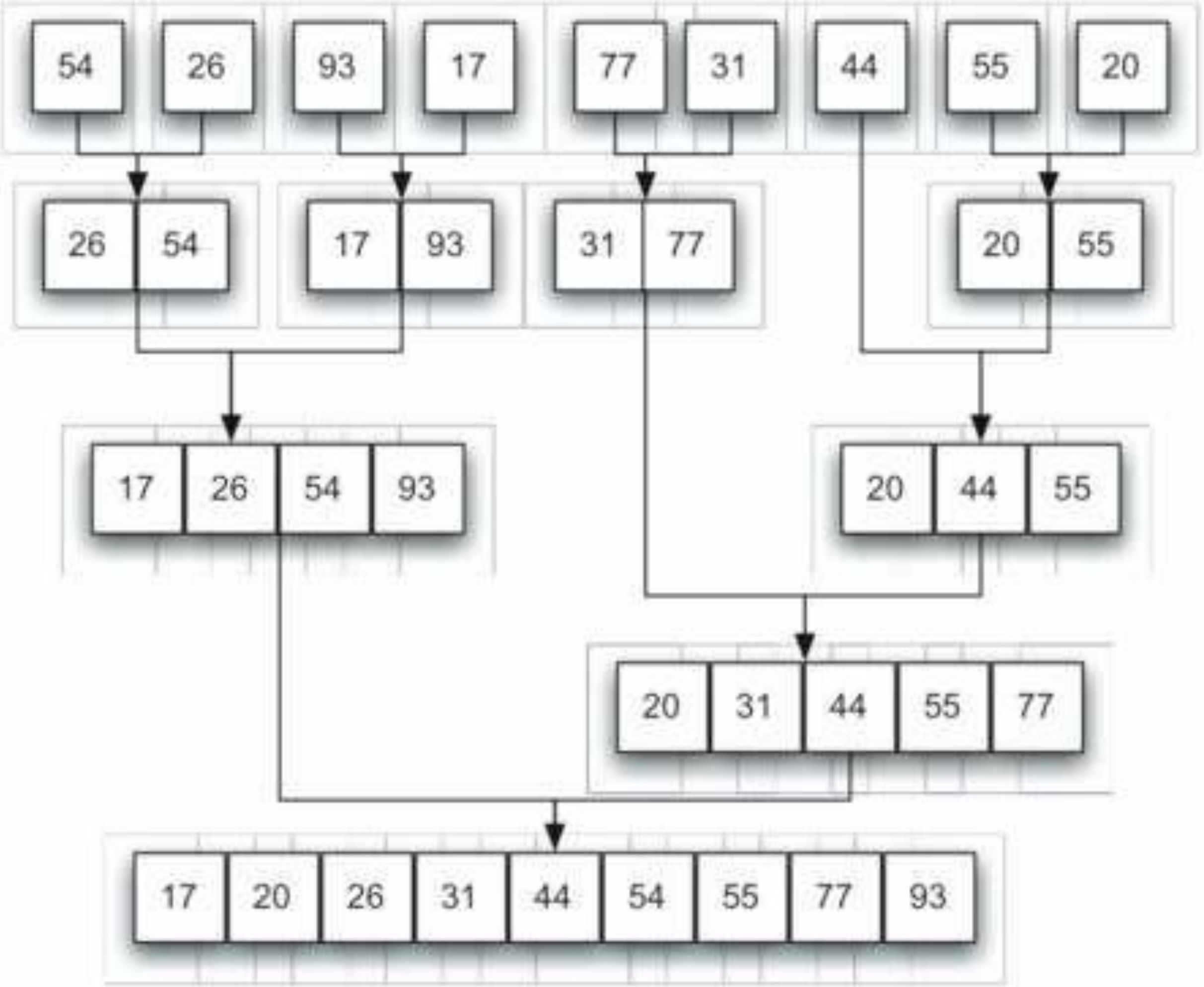
Merge Sort

- Because we're using divide-and-conquer to sort, we need to decide what our sub problems are going to be.

- Full Problem: Sort an entire Array
- Sub Problem: Sort a sub array

- Lets assume array[p..r] denotes this subarray of array.
- For an array of n elements, we say the original problem is to sort array[0..n-1]


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20


54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

Merge Sort

- Here's how merge sort uses divide and conquer
  1. *Divide* by finding the number q of the position midway between p and r. Do this step the same way we found the midpoint in binary search: add p and r, divide by 2, and round down.

  2. *Conquer* by recursively sorting the subarrays in each of the two sub problems created by the divide step. That is, recursively sort the subarray array[p..q] and recursively sort the subarray array[q+1..r].

  3. *Combine* by merging the two sorted subarrays back into the single sorted subarray array[p..r].

Merge Sort

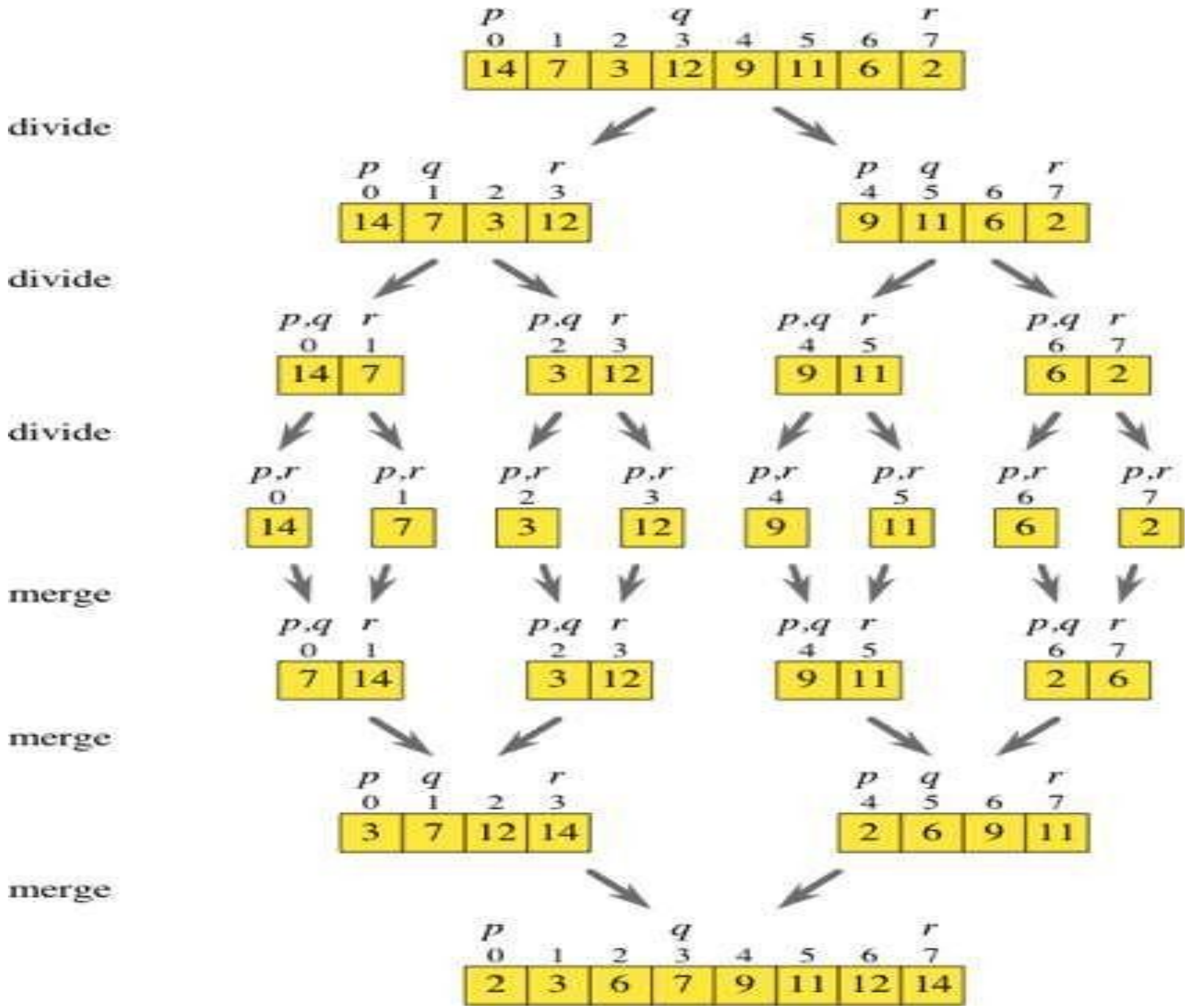
- Let's start with array holding [14,7,3,12,9,11,6,2]
- We can say that array[0..7] where p=0 and r=7
- In the divide step we compute q=3
- The conquer step has us sort the two subarrays
  - array[0..3] = [14,7,3,12]
  - array[4..7]= [9,11,6,2]
  - When we comeback from the conquer step, each of the two subarrays is sorted i.e.
  - array[0..3] = [3,7,12,14]
  - array[4..7]= [2,6,9,11]
- Finally, the combine step merges the two sorted subarrays in first half and the second half, producing the final sorted array [2,3, 6,7,9, 11, 12,14]

How did the subarray array[0..3] become sorted?

- It has more than two element so it's not a base case.
- So with p=0 and r=3, compute q=1, recursively sort array[0..1] and array[2..3], resulting in array[0..3] containing [7,14,3,12] and merge the fist half with the second half, producing [3,7,12,14]

How did the subarray array[0..1] become sorted?

- With p=0 and r=1, compute q=0, recursively sort array[0..0] ([14]) and array[1..1] ([7]), resulting in array[0..1] still containing [14, 7], and merge the first half with the second half, producing [7, 14].



p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 2
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r 0 14
p,r 1 7
p,r 2 3
p,r 3 12
p,r 4 9
p,r 5 11
p,r 6 6
p,r 7 2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

Analysis of merge Sort

- We can view merge sort as creating a tree of calls, where each level of recursion is a level in the tree.

- Since number of elements is divided in half each time, the tree is balanced binary tree.

- The height of such a tree tend to be log n

Analysis of merge Sort

- Divide and conquer
- Recursive
- Stable
- 0(n) space complexity
- 0(nlogn) time complexity

Quick Sort

- Quick sort is one of the most popular sorting techniques.

- As the name suggests the quick sort is the fastest known sorting algorithm in practice.

- It has the best average time performance.

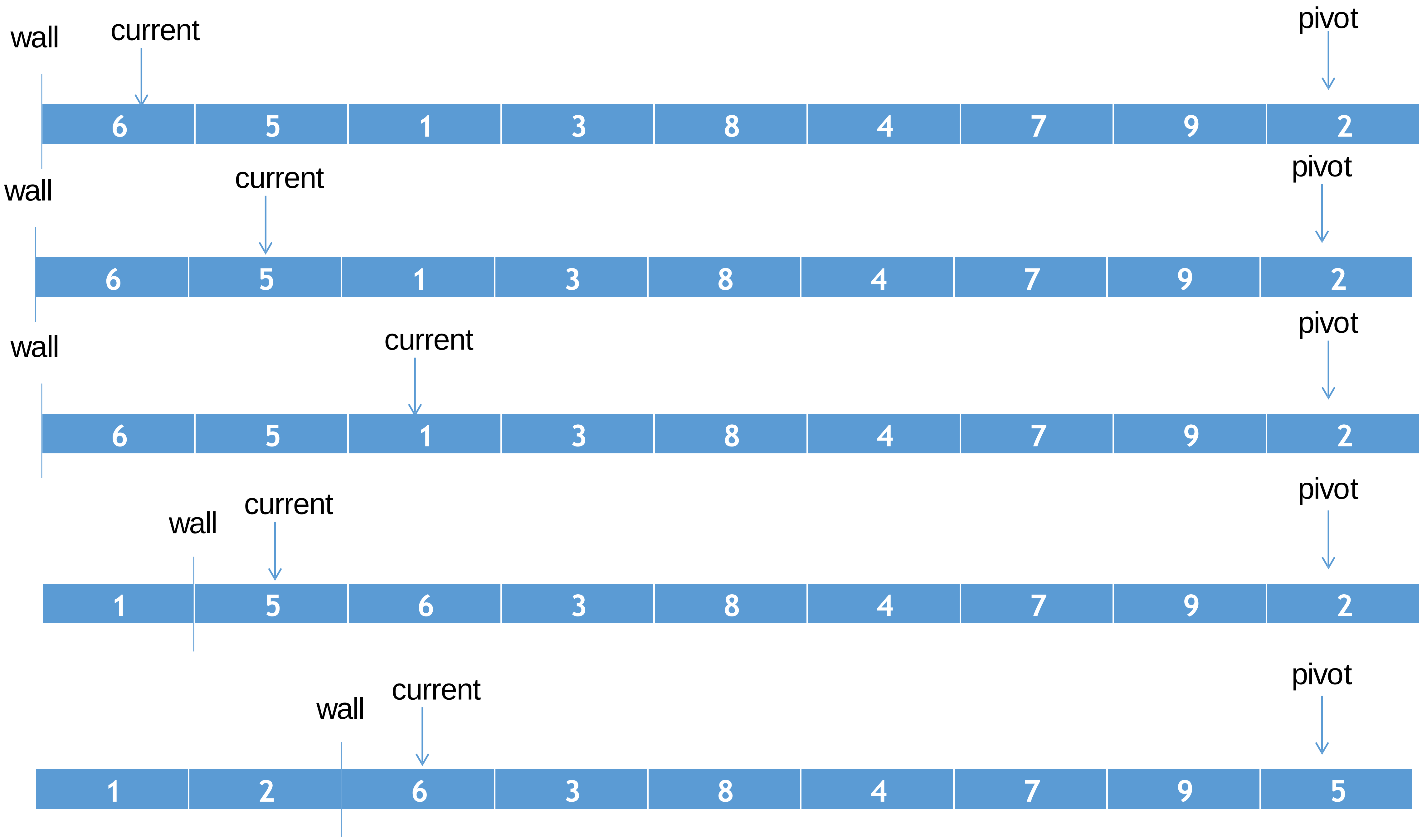
- It works by partitioning the array to be sorted and each partition in turn sorted recursively. Hence also called partition exchange sort.

Quick Sort

- In partition one of the array elements is choses as a pivot element

- Choose an element pivot=a[n-1]. Suppose that elements of an array a are partitioned so that pivot is placed into position I and the following condition hold:
  - Each elements in position 0 through i-1 is less than or equal to pivot
  - Each of the elements in position i+1 through n-1 is greater than or equal to key

- The pivot remains at the i^{th} position when the array is completely sorted. Continuously repeating this process will eventually sort an array.


wall
current
pivot
6 5 1 3 8 4 7 9 2
wall
current
pivot
6 5 1 3 8 4 7 9 2
wall
current
pivot
6 5 1 3 8 4 7 9 2
wall
current
pivot
1 5 6 3 8 4 7 9 2
wall
current
pivot
1 2 6 3 8 4 7 9 5

Algorithm

- Choosing a pivot
  - To partition the list we first choose a pivot element
- Partitioning
  - Then we partition the elements so that all those with values less than pivot are placed on the left side and the higher vale on the right
  - Check if the current element is less than the pivot.
    - If lesser replace it with the current element and move the wall up one position
    - else move the pivot element to current element and vice versa
- Recur
  - Repeat the same partitioning step unless all elements are sorted

Analysis of Quick Sort

- Best case
  - The best case analysis assumes that the pivot is always in the middle
  - To simplify the math, we assume that the two sublists are each exactly half the size of the original $T(N)=T(N/2)+T(N/2)\ldots.+1$ leads to $T(N)=O(nlogn)$

- Average case
  - $T(N)=O(nlogn)$

- Worst case
  - When we pick minimum or maximum as pivot then we have to go through each and every element so
  - $T(N) = O(n^2)$

Reference

- https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort
- http://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm
- http://bigocheatsheet.com/
- http://stackoverflow.com/questions/5222730/why-is-merge-sort-preferred-over-quick-sort-for-sorting-linked-lists