Design and Analysis of Algorithms

Chapter 2

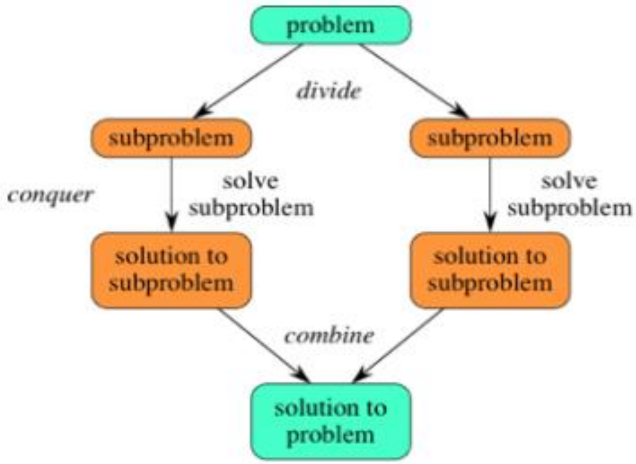
Divide & Conquer

Divide & Conquer

- Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity.
- **For example**, from O (n2) to O (n log n) to sort the elements.
- Divide and conquer strategy is as follows: **divide** the problem instance into two or more smaller instances of the same problem, **solve** the smaller instances recursively, and **assemble** the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide.
- Divide and conquer algorithm consists of 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 best cases.
  - **Combine**: - the solutions to the subproblems into the solution for the original problem.

Chapter 2 - Divide & Conquer

- Break a problem into smaller pieces and solve the smaller sub-problems
- **Searching a Dictionary**
- A detailed specification of this process:
    1. the goal is to search for a word w in region of the book
    2. the initial region is the entire book
    3. at each step pick a word x in the middle of the current region
    4. there are now two smaller regions: the part before x and the part after x
    5. if w comes before x, repeat the search on the region before x, otherwise search the region following x (go back to step 3)
- Note: at first a "region" is of a group of pages, but eventually a region is a set of words on a single page



Linear Search (Sequential Search)

Pseudocode

Loop through the array starting at the first element until the value of target matches one of the array elements.

If a match is not found, return –1.

Time is proportional to the size of input *(n)* and we call this time complexity $O(n)$.

Binary Search

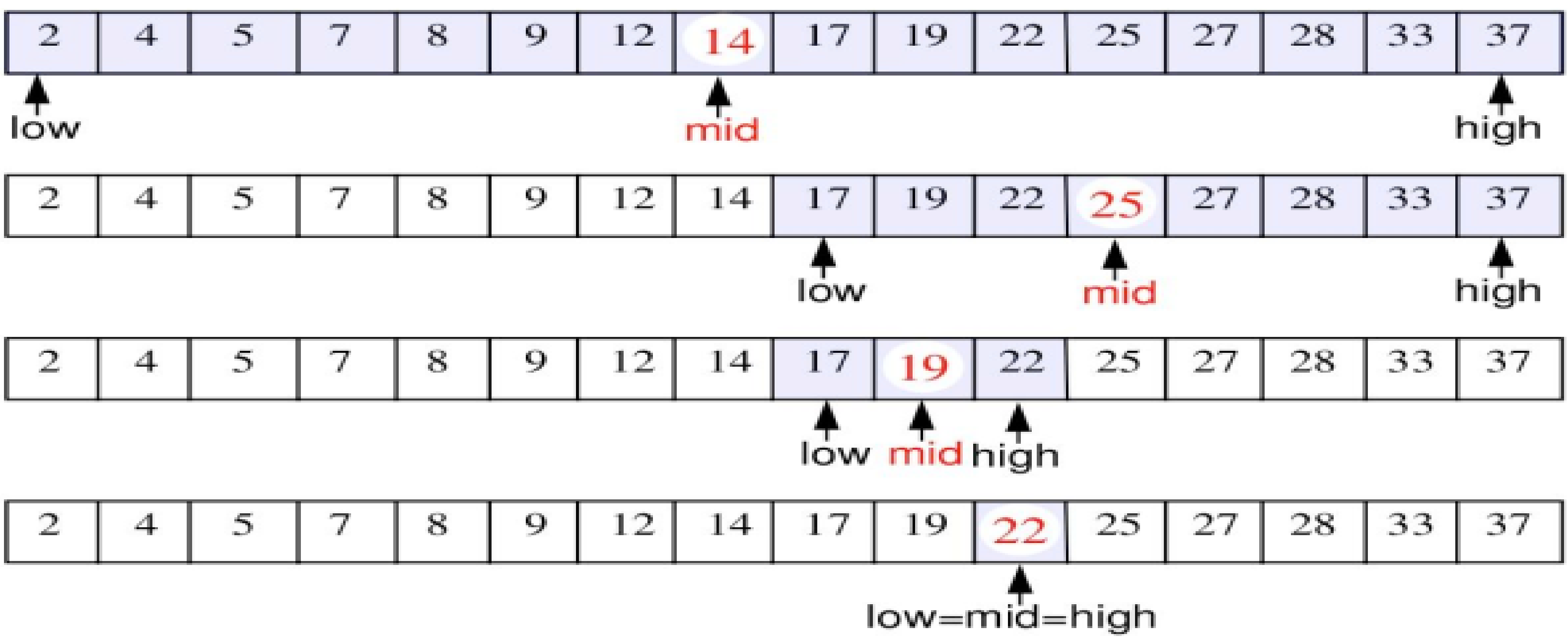
- The binary search algorithm uses the divide-and-conquer strategy to search through an array
- This searching algorithms works only on an ordered list.
- The basic idea is:
- Locate midpoint of array to search
- Determine if target is in lower half or upper half of an array.
    - If in lower half, make this half the array to search
    - If in the upper half, make this half the array to search

• Loop back to step 1 until the size of the array to search is one, and this element does not match, in which case return –1.

The computational time for this algorithm is proportional to $\log_2 n$.

Therefore the time complexity is O(log n)

Find 22 in the following array.



Merge Sort

- Merge sort algorithm is a classic example of divide and conquer.
- To sort an array, recursively, sort its left and right halves separately and then merge them.
- The time complexity of merge mort in the *best case, worst case* and *average case* is O (n log n) and the number of comparisons used is nearly optimal.

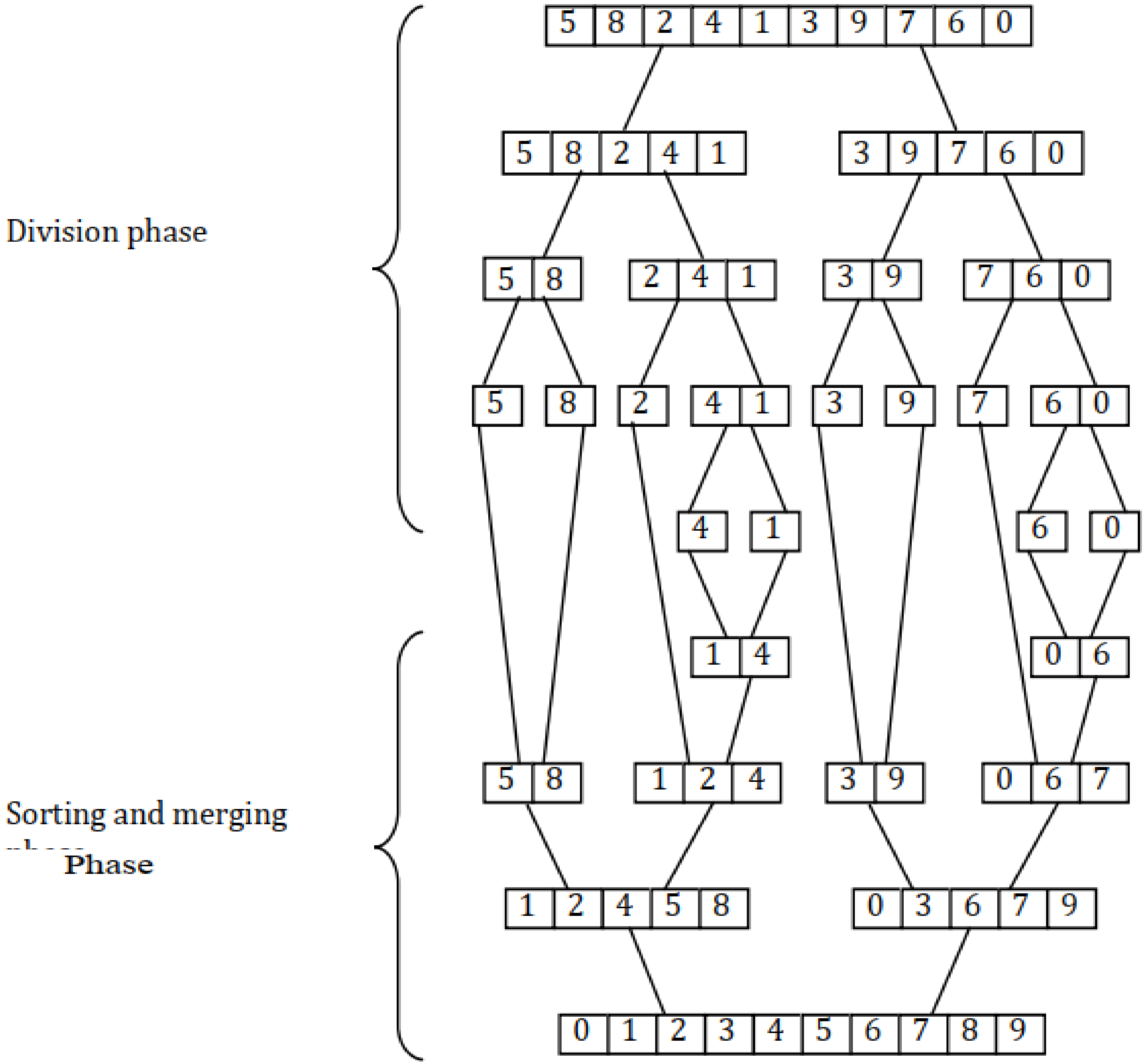
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).

Algorithm:

1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Example: Sort the following list using merge sort algorithm.

5	8	2	4	1	3	9	7	6	0


Division phase
Sorting and merging
Phase
5 8 2 4 1 3 9 7 6 0
5 8 2 4 1
3 9 7 6 0
0 1 2 3 4 5 6 7 8 9

Quick Sort

- **Quick sort is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).**
- **In essence, the quick sort algorithm partitions the original array by rearranging it into two groups.**
- **The first group contains those elements less than some arbitrary chosen. The chosen value is known as the *pivot element* .**

Algorithm:

1. Choose a **pivot value** (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
   - the left part has items less than or equal to the pivot value
   - the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

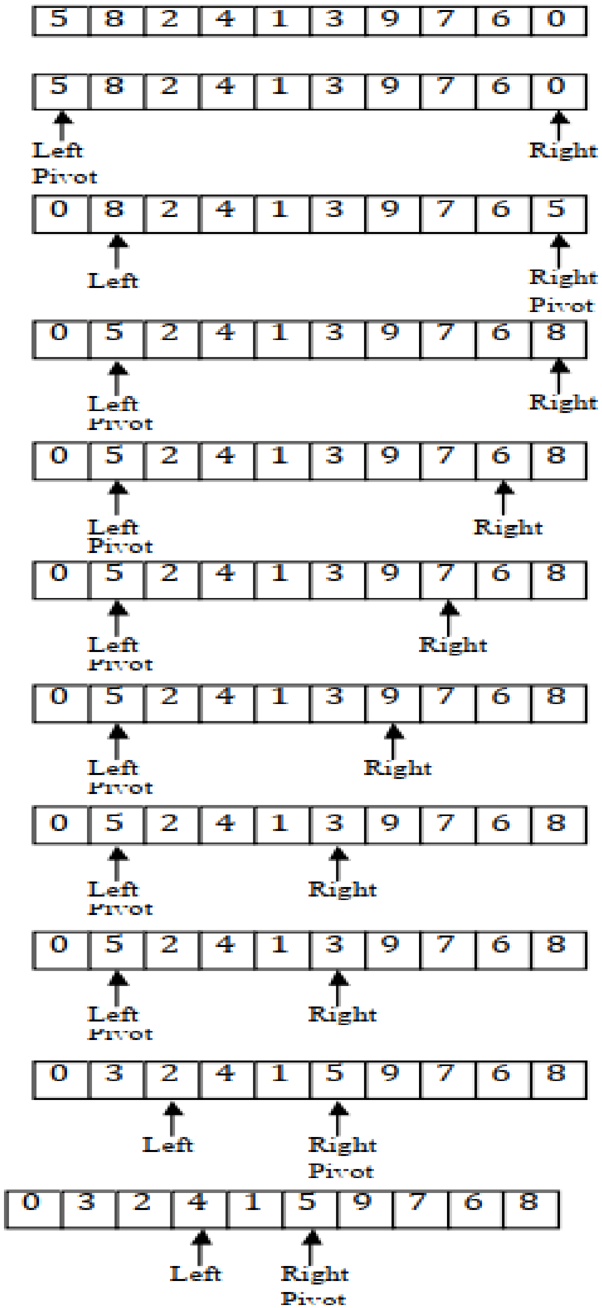
The following algorithm can be used to position a pivot value and create partition.

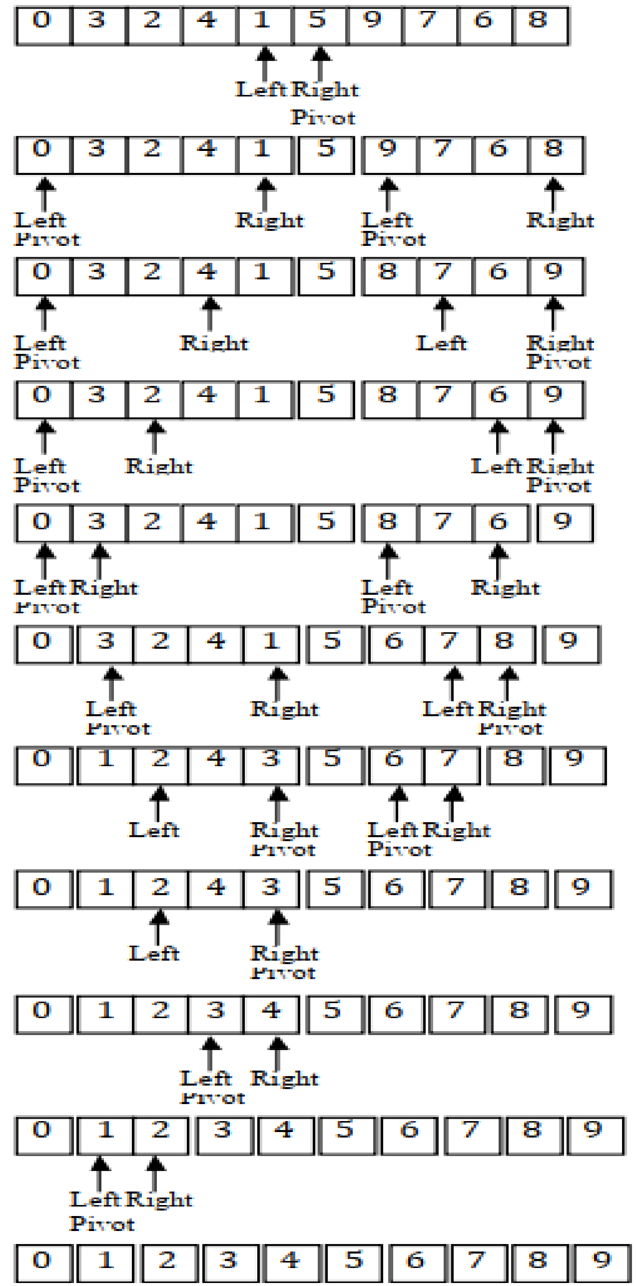
```
Left=0;
Right=n-1; // n is the total number of elements in the list
PivotPos=Left;
while(Left<Right) {
if(PivotPos==Left) {
if(Data[Left]>Data[Right]) {
swap(data[Left], Data[Right]);
PivotPos=Right;
Left++;
}
else Right--;
} else {
if(Data[Left]>Data[Right]) {
swap(data[Left], Data[Right]);
PivotPos=Left;
Right--; }
else Left++;
} }
```

Partition

```
while L <= R:
    while array[L] < pivot:
        L++
    while array[R] > pivot:
        R--
    if L <= R:
        array[L], array[R] = array[R], array[L]
        L++
        R--
return L
```

Example: Sort the following list using quick sort algorithm.





Selection Sort

Basic Idea:

- Loop through the array from i=0 to n-1.
- Select the smallest element in the array from i to n
- Swap this value with value at position i.

Given array abc = 34 8 64 51 32 21

No of passes	the array	positions moved
Original	34 8 64 51 32 21	-
Pass=1	8 34 64 51 32 21	1
Pass=2	8 21 64 51 32 34	1
Pass=3	8 21 32 51 64 34	1
Pass=4	8 21 32 34 64 51	1
Pass=5	8 21 32 34 51 64	1

Analysis

How many comparisons?

$(n-1)+(n-2)+\ldots+1= O(n^2)$

How many swaps?

$n=O(n)$

How much space?

In-place algorithm