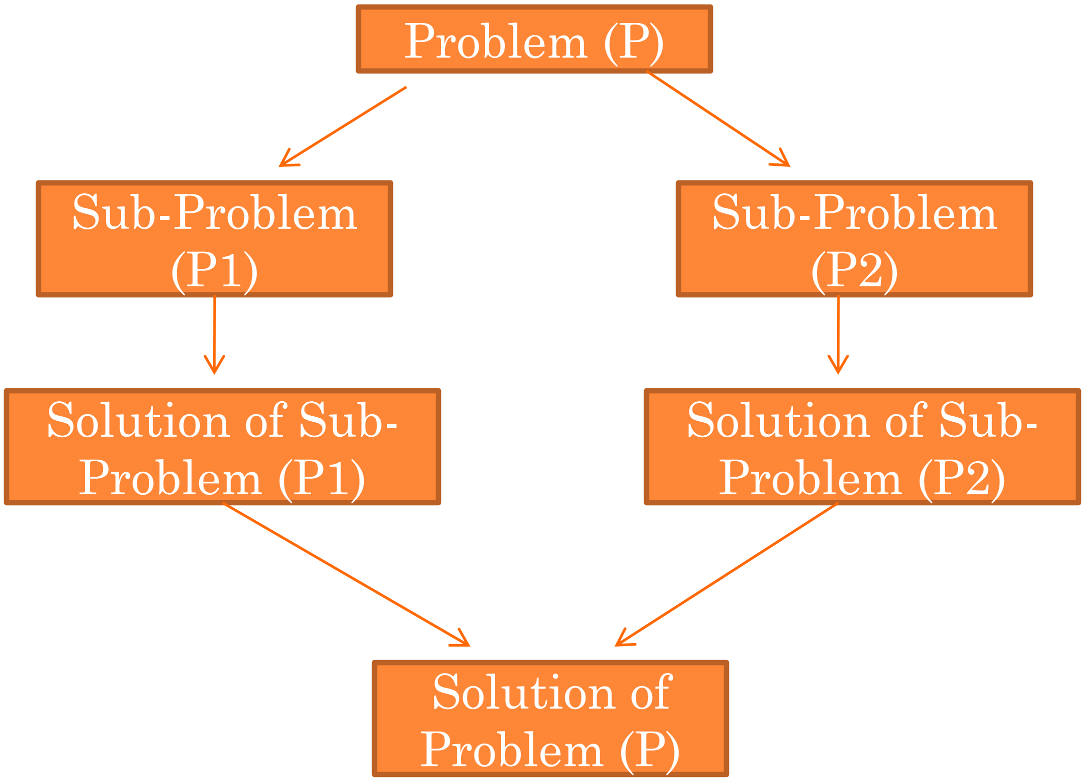
Divide and Conquer

Tanjina Helaly

Divide and Conquer

- Divide and Conquer is an algorithm design paradigm / technique.
- Divide and conquer **approach** has 3 steps
  - **Division -** the problem is divided into smaller sub-problems of similar type
  - **Conquer -**each sub-problem is solved independently.
  - **Merge/Combine -** The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.
- How far should we divide?
  - When we keep on dividing until you reach a stage where no more division is possible or solution is straight forward.

Divide and Conquer



RECURRENCE

- A ***recurrence is an equation or inequality that describes a function in terms*** of its value on smaller inputs.
  - E.g. fact(n) = n*fact(n-1)

- Recurrences go hand in hand with the divide-and-conquer paradigm,

- Because both of them are described/solved in terms of a smaller problem.

Example of Divide and Conquer

- Binary Search
  - Divide- divide in to 2 halves and select lower of upper half
  - Conquer – Search in selected half
  - Combine – None

- Merge Sort
  - Divide- divide in to 2 halves.
  - Conquer – Sort each half recursively.
  - Combine – Combine the 2 sorted list

Example of Divide and Conquer

- Quick Sort
  - Divide – partition the array using pivot (Divide is the most important segment of quick sort)
  - Conquer – move smaller element to the left and bigger to right
  - Combine - None

Example of Divide and Conquer

- Calculate power (x,n)
  - Divide – divide the power term n to half n/2
  - Conquer – Find the $x^{n/2}$
  - Combine – multiply the $x^{n/2}$ with $x^{n/2}$

- Find Minimum of an array
  - Divide – Divide the array into 2 halves
  - Conquer – Find the minimum of the 2 subArray
  - Combine – take the minimum of the minimum of 2 subarray

Calculate Power

Time Complexity of optimized solution: O(logn)

```
#include<stdio.h>

int power(int x, int y) {
    int temp;
    if( y == 0) return 1;
    temp = power(x, y/2);
    if (y%2 == 0)  return temp*temp;
    else     return x*temp*temp;
}
```

Binary Search

Binary Search

```
Binary-Search(A, low, high, item):
   if (low>high) return false;
   else
      mid = (low+high)/2;
      if (item == A[mid])
          return true;
      if(item<A[mid])
          Binary-Search(A, low, mid-1, item)  ← Same problem of size n/2
      else
          Binary-Search(A, mid+1, high, item) ← Same problem of size n/2
```

Binary Search – Time Complexity

```
Binary-Search(A, low, high, item):
    if (low>high) return false;
    else
        mid = (low+high)/2;
        if (item == A[mid])
            return true;
        if(item<A[mid])
            Binary-Search(A, low, mid-1, item)
        else
            Binary-Search(A, mid+1, high, item)
```

For array of size n,
Time complexity is **T(n)**

constant time: c_1

constant time: c_2

constant time: c_3

Same problem of size n/2
So, Time Complexity= **T(n/2)**

Same problem of size n/2
So, Time Complexity= **T(n/2)**

So, Time Complexity T(n) = T(n/2) +C

Merge Sort

MERGE SORT

- Problem:
  - Given 2 sorted array, need to merge these 2 arrays in one sorted array.

- The key operation of the merge sort algorithm is the merging of two sorted sequences to one sorted sequence in the “combine” step

Merge Sort

- Algorithm of Merge
  - Keep track of the smallest element in each sorted half.
  - Choose smaller of two elements
  - Repeat until done

Merge Sort

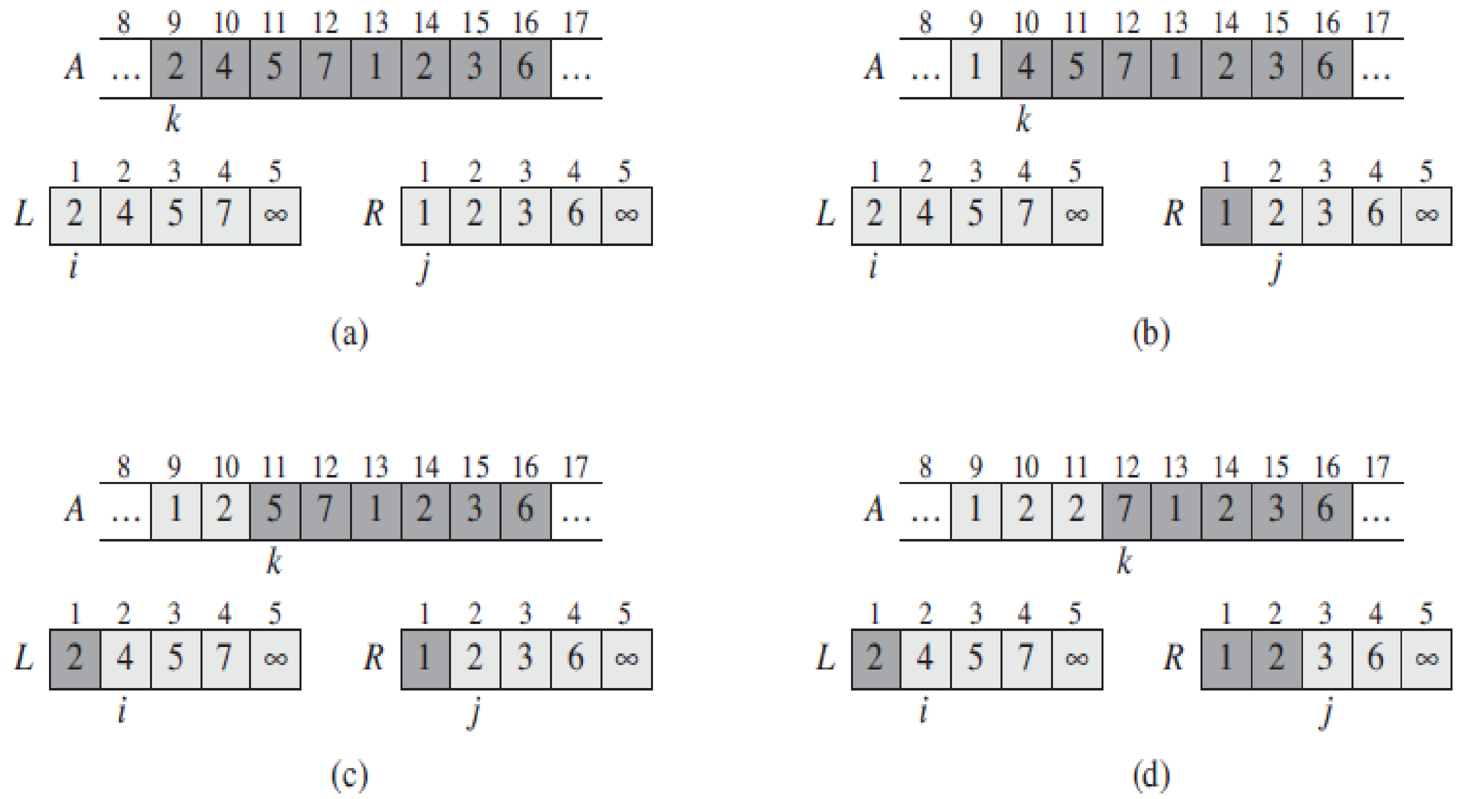
MERGE-SORT(A, l, h)

```
if l < h
    m = (l + h)/2
    MERGE-SORT(A, l, m)
    MERGE-SORT(A, m+1, h)
    MERGE(A, l, h, m)
```

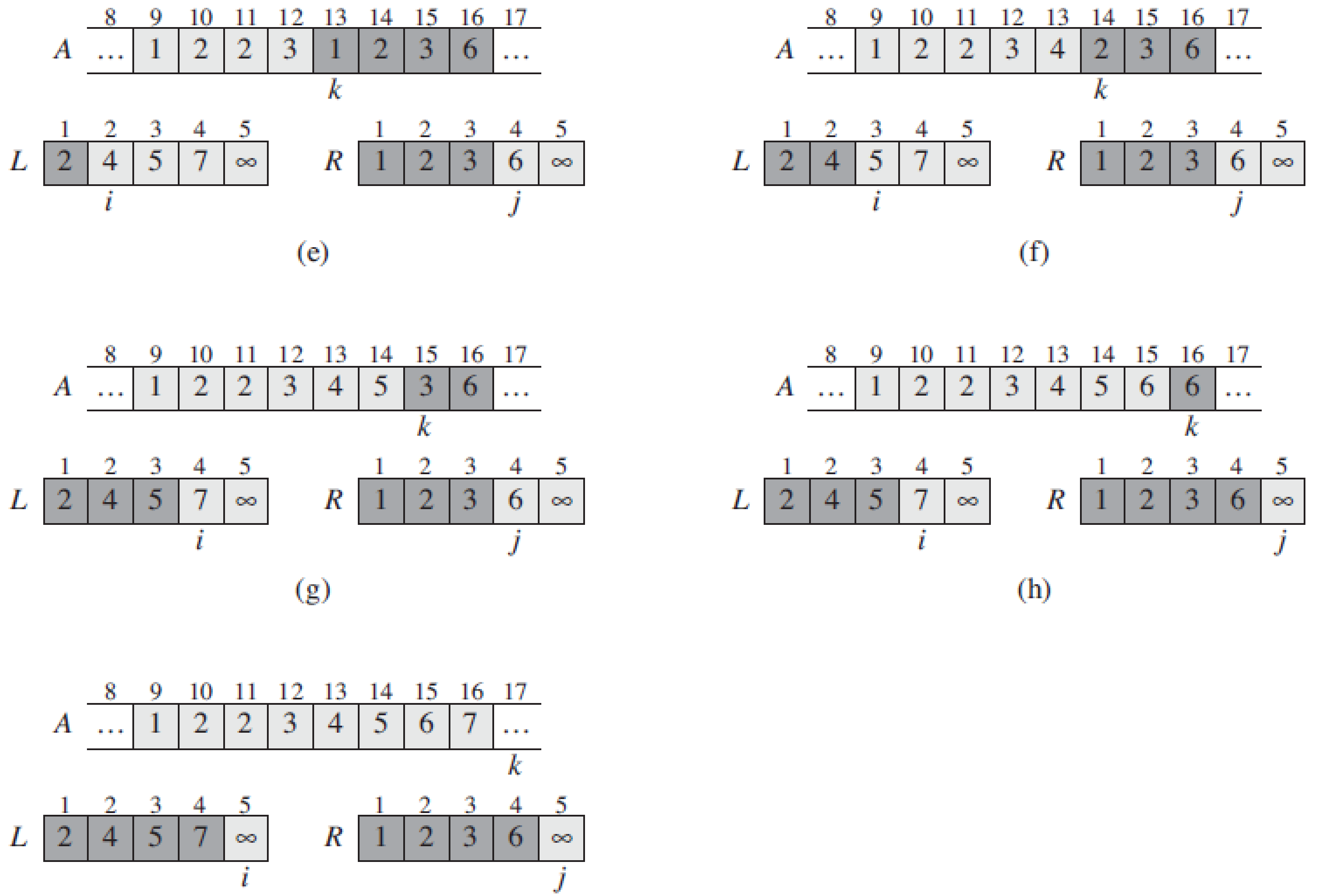
MERGE(A, low, mid, high)

```
n1 = mid – low + 1
n2 = high - mid
let L[0… n1] and R[0… n2] be new arrays
for i = 1 to n1
        L[i ]= A[low+ i  -1]
for j = 1 to n2
        R[j]= A[mid+ j]
L[n1 + 1] = ∞ , R[n2 + 1] = ∞
i = 1,  j = 1
for k = low to high
    if L[i] ≤  R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

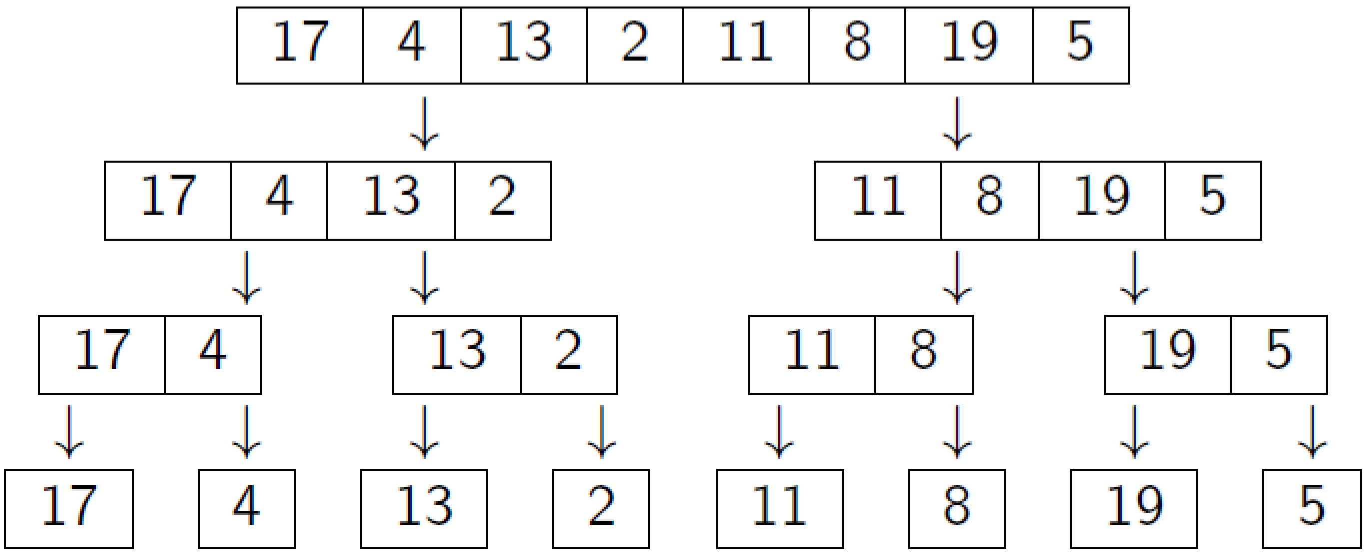
Merge Sort – Merge/Combine step



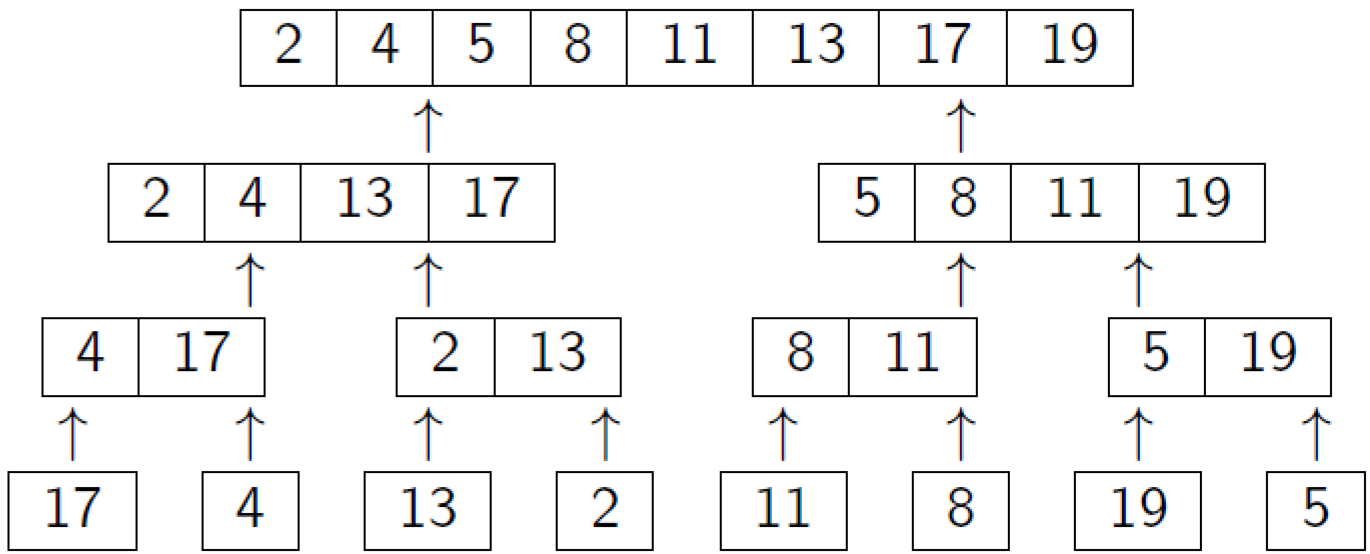
MERGE SORT – MERGE/COMBINE STEP



Merge Sort - Divide



Merge Sort – Merge/Combine



MERGE SORT

MERGE-SORT(A, l, h)	**T(n)**
if l < h	
m = (l + h)/2	**c**
MERGE-SORT(A, l, m)	**T(n/2)**
MERGE-SORT(A, m+1, h)	**T(n/2)**
MERGE(A, l, h, m)	**O(n) How??**

Time Complexity, T(n) = c + 2T(n/2) + O(n)

Merge Sort

MERGE(A, low, mid, high)

```
n1 = mid – low + 1
n2 = high - mid
let L[0… n1] and R[0… n2] be new arrays
for i = 1 to n1
        L[i ]= A[low+ i  -1]
for j = 1 to n2
        R[j]= A[mid+ j]
L[n1 + 1] = ∞ , R[n2 + 1] = ∞
i = 1,  j = 1
for k = low to high
   if L[i] ≤  R[j]
        A[k] = L[i]
        i = i + 1
   else A[k] = R[j]
        j = j + 1
```

O(n) (lines 4–7)

O(n) (lines 10–15)

Analysis – Divide and Conquer

- For any **Divide and Conquer** algorithm if the original problem of size **n** is divided into **a** number of sub-problems, each of size **n=b**, then the running time T(n) can be expressed as the following:

$$T(n) = \begin{cases} \Theta(1), & if\ n \le c \\ aT(n/b) + D(n) + C(n), & otherwise \end{cases}$$

- Here,
  - c is a small constant and
  - D(n) is the time needed to divide the problem and
  - C(n) is the time needed to combine them back.

Analysis – Merge Sort

- For Merge Sort
  - $D(n) = \Theta(1)$
  - $a = 2$
  - $b = 2$
  - $c = 1$

- So, Time Complexity of Merge Sort

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(1), & \text{if } n > 1 \end{cases}$$

Analysis – Merge Sort

- Or we can replace $\Theta(1)$ with constant, c and $\Theta(n)$ with ***cn***. For small constant c, we can rewrite the whole equation as,

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(n/2) + cn, & \text{if } n > 1 \end{cases}$$

Reference

- Introduction to Algorithms – Chapter 2.3