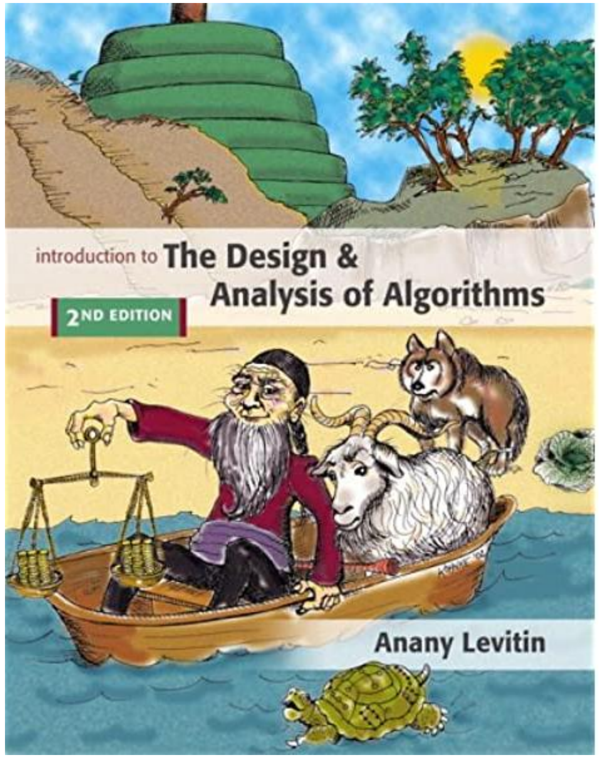
ESTD:2001
An Institute with a Difference

Design and Analysis of Algorithms

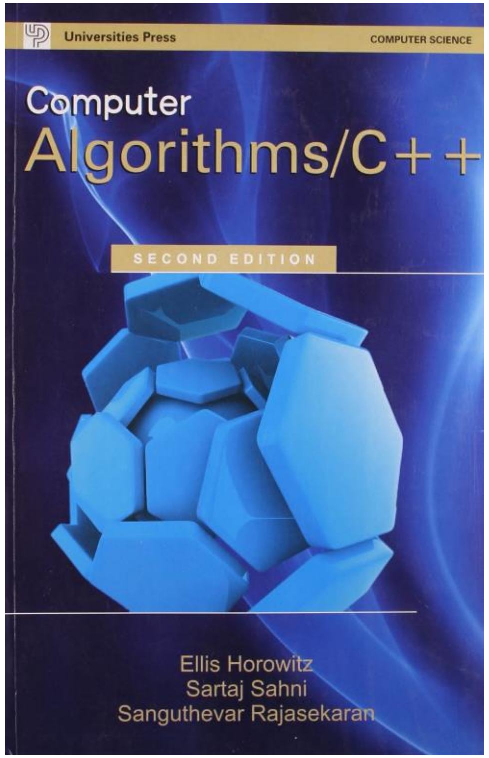
Divide and Conquer

Manjula L
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Text Books



introduction to The Design & Analysis of Algorithms
2ND EDITION
Anany Levitin



Universities Press
COMPUTER SCIENCE
Computer Algorithms/C++
SECOND EDITION
Ellis Horowitz
Sartaj Sahni
Sanguthevar Rajasekaran

Introduction

- Divide-and-Conquer (DaC) is probably the best-known general algorithm design technique.
- Given a function to compute on n input the DaC approach suggests splitting the inputs into k distinct subsets,1< k < n, yielding k subproblems
- These subproblem must be solved and then a method must be found to combine solutions into a solution of the whole.
- If the sub problems are relatively large then the divide and conquer approach can possibly be reapplied
- Often the sub problems resulting from our divide and conquer design are of the same type as the original problem.
- Fur those cases the re application of the divide and conquer principle is naturally expressed by a recursive algorithm
- The smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced

Introduction

Example : Detecting a counterfeit coin

- Given a bag of n coins and a machine that weighs 2 sets of coins, the task is to find
  - Whether the bag contains a counterfeit coin
  - If present then identify the Counterfeit coin

Introduction

Control abstraction for Divide and Conquer approach

```
Algorithm DAndC(P)
    {
        if Small(P) then return S(P)
        else
        {
            divide P into Smaller instances P1, P2, P3 ... Pk, k≥1 ;
            Apply DAndC to each of these subproblems;
            return Combine(DAndC(P1), DAndC(P2),...DAndC(Pk));
        }
    }
```

Boolean valued function that determines whether the input is small enough or not

Solution to the problem

A function that determines the solution to **P** using the solutions to **k** subproblems

Introduction

- If the size of **P** is **n**, And the sizes of the **k** subproblems are **n1, n2, n3… nk**, respectively, then the computing time of DAndC is described by the recurrence relation
- $$T(n) = \begin{cases} g(n) & n \; small \\ T(n_1) + T(n_2) + \cdots . + T(nk) & otherwise \end{cases}$$

Where,

- **T(n)** is the time for DAndC on any input of size n
- **g(n)** is the time to compute the answer directly for small inputs
- **f(n)** is the time for dividing **P** and combining the solutions to subproblems.

Introduction

- The complexity of many divide and conquer is given by recurrences of the form
- $$T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

Where,

- **a** and **b** are constants
- **T(1)** is known
- **n** is a power of **b** (i.e., $n=b^k$)

Binary Search

- Binary search is a remarkably efficient algorithm for searching in a sorted array
- It works by comparing a search key **K** with the array's middle element **A[m]**.
- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if **K <A[m]**, and for the second half if **K >A[m]**

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0]\ldots A[m-1]}_{\substack{\text{search here if} \\ K<A[m]}} \quad A[m] \quad \underbrace{A[m+1]\ldots A[n-1]}_{\substack{\text{search here if} \\ K>A[m]}}. \end{array}$$

Binary Search

Example

- Apply binary search algorithm to the following set of numbers considering **70** as the key

3	14	27	31	39	42	55	70	74	81	85	93	98

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

Binary Search : Non-Recursive

```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// order, n ≥ 0, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid − 1;
        else if (x > a[mid]) then low := mid + 1;
             else return mid;
    }
    return 0;
}
```

Binary Search : Recursive

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else  if (x < a[mid]) then
                  return BinSrch(a, i, mid − 1, x);
              else return BinSrch(a, mid + 1, l, x);
    }
}
```

Binary Search : Analysis

Recurrence Relation

- $T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + 1 & n > 1 \end{cases}$

Solution

T(n)= T(n/2)+1

= [T(n/4)+1]+1 = T(n/4)+2

=[T(n/8)+1]+2 = T(n/8) +3

\- - - - - - - - - - - - - - -

\- $=T(n/2^k)+k$ $n = b^k$, $n=2^k$, $\log n = \log 2^k$, $k=\log n$

\- =T(n/n)+k

\- =1+k

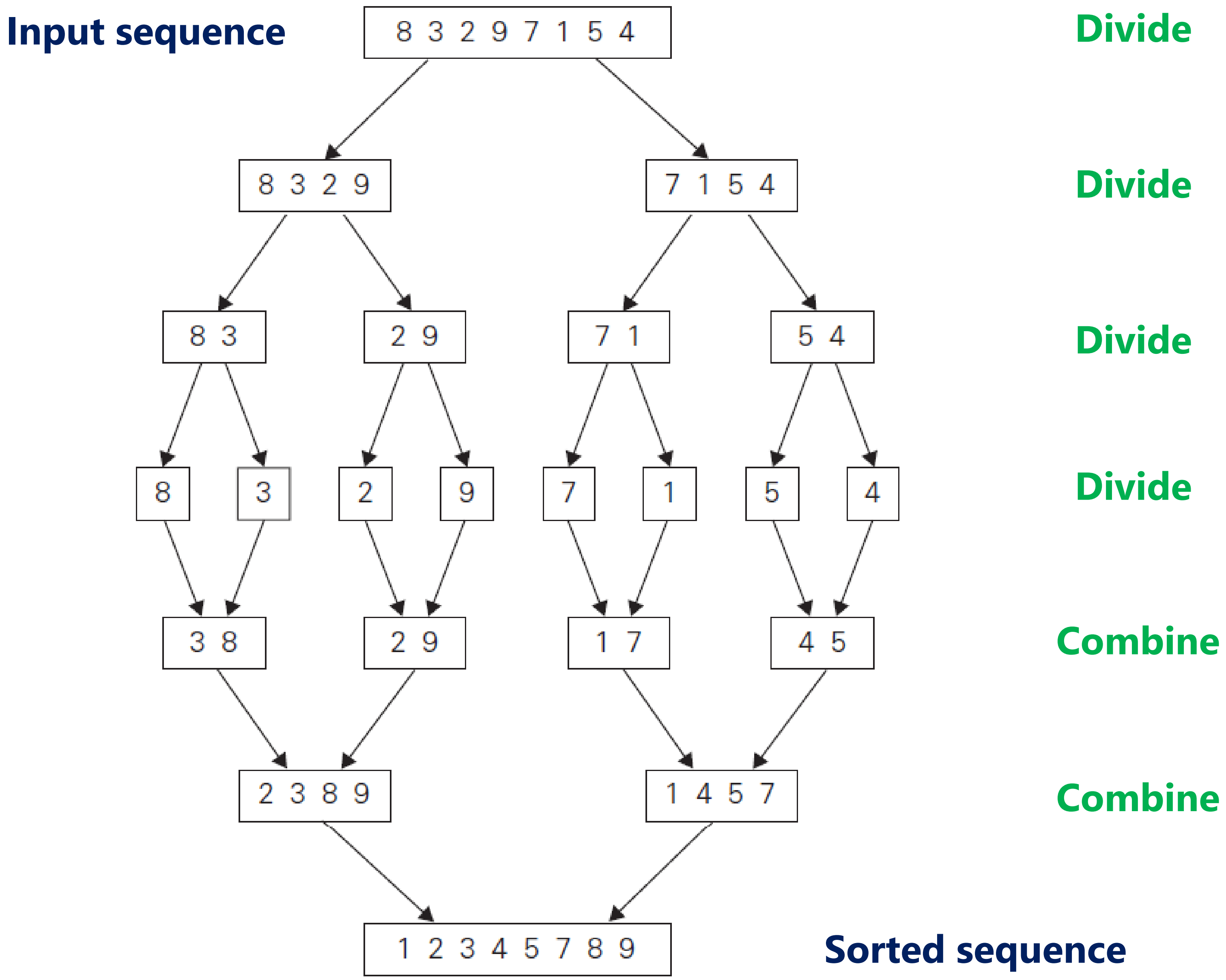
\- =1+logn

\- **T(n)=O(logn)**

Merge Sort

- Merge sort is a perfect example of a successful application of the divide-and conquer technique.
- It has the nice property that in the worst case its complexity is **O(nlogn).**
- Let us assume that the set of elements are to be sorted in **non-decreasing** order that is in **ascending** order.
- Given a sequence of n elements , a[1]……a[n], the idea is to imagine them split into two sets a[1]…… a[⌊n/2⌋] and a[⌊n/2⌋+1]…… a[n].
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of **n** elements
- This is an ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one

Merge Sort



Input sequence
8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9
Sorted sequence
Divide
Divide
Divide
Divide
Combine
Combine

Merge Sort

Do it yourself

- Consider the input sequence

11, 44, 22, 99, 66, 33, 88, 55, 77, 00

Obtain the merge sort tree representation showing the divide and combine phase

Merge Sort – algorithm

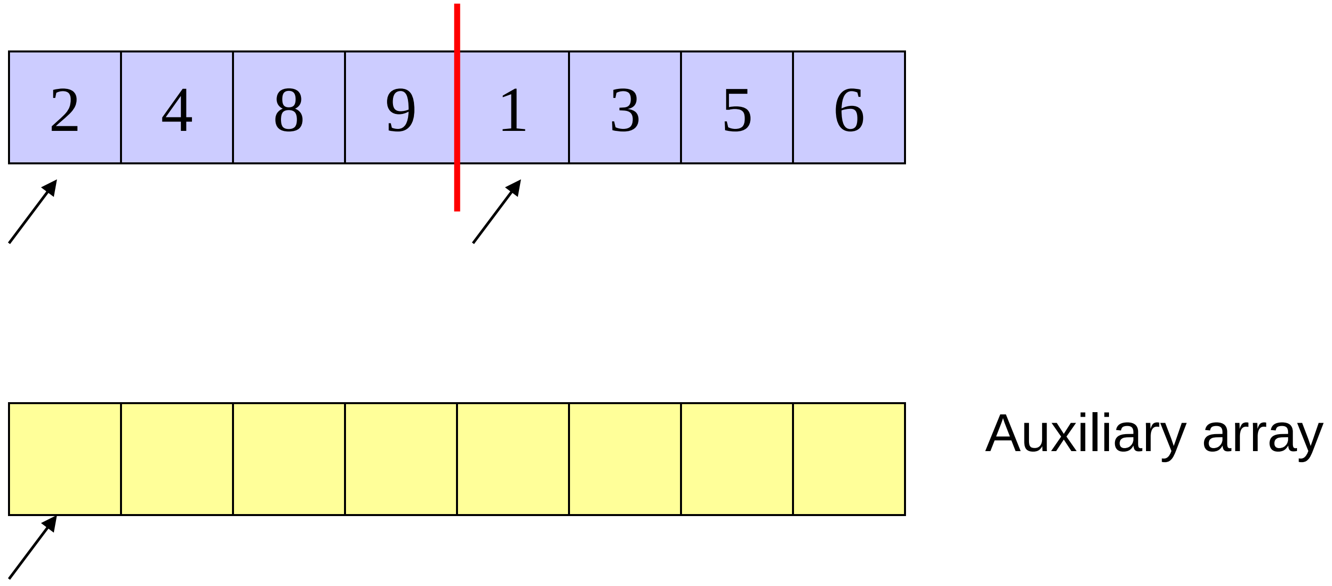
```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Merge Sort – algorithm

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

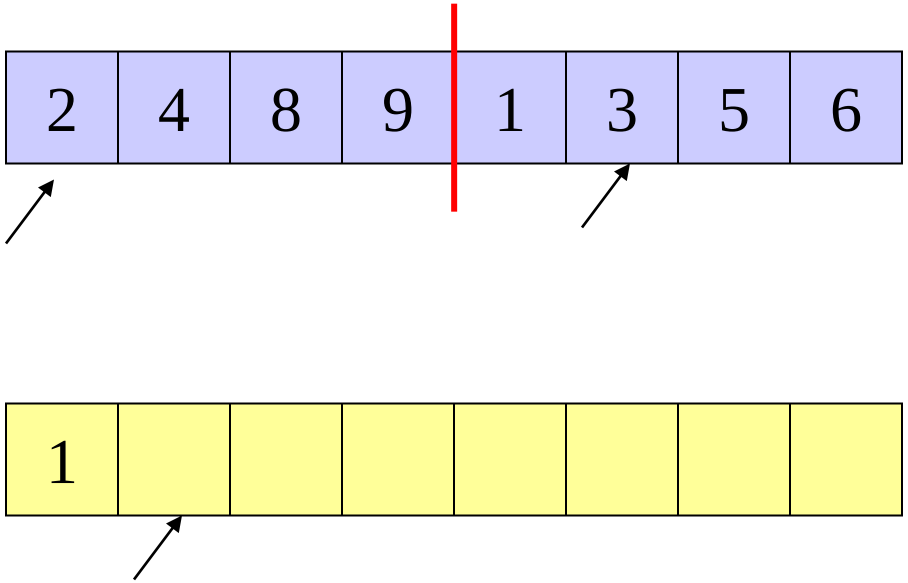
Working of Merge

- The merging requires an auxiliary array.



Working of Merge

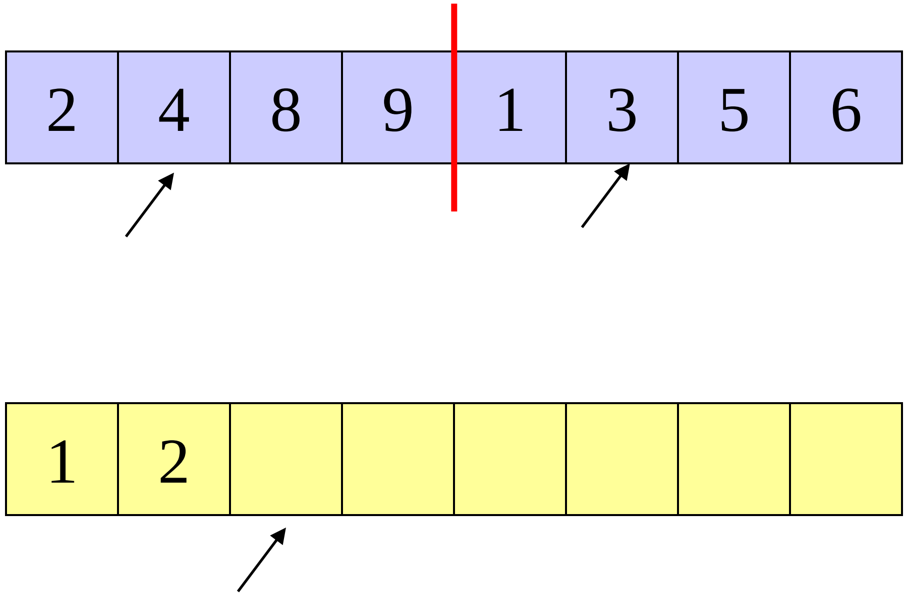
- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

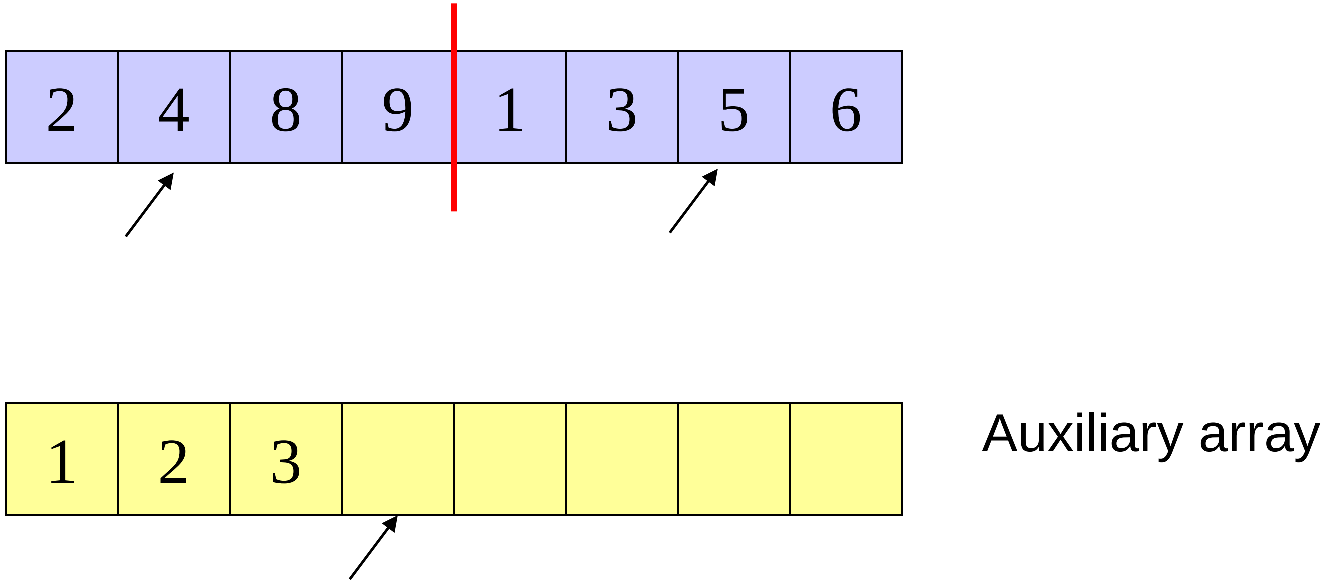
- The merging requires an auxiliary array.



Auxiliary array

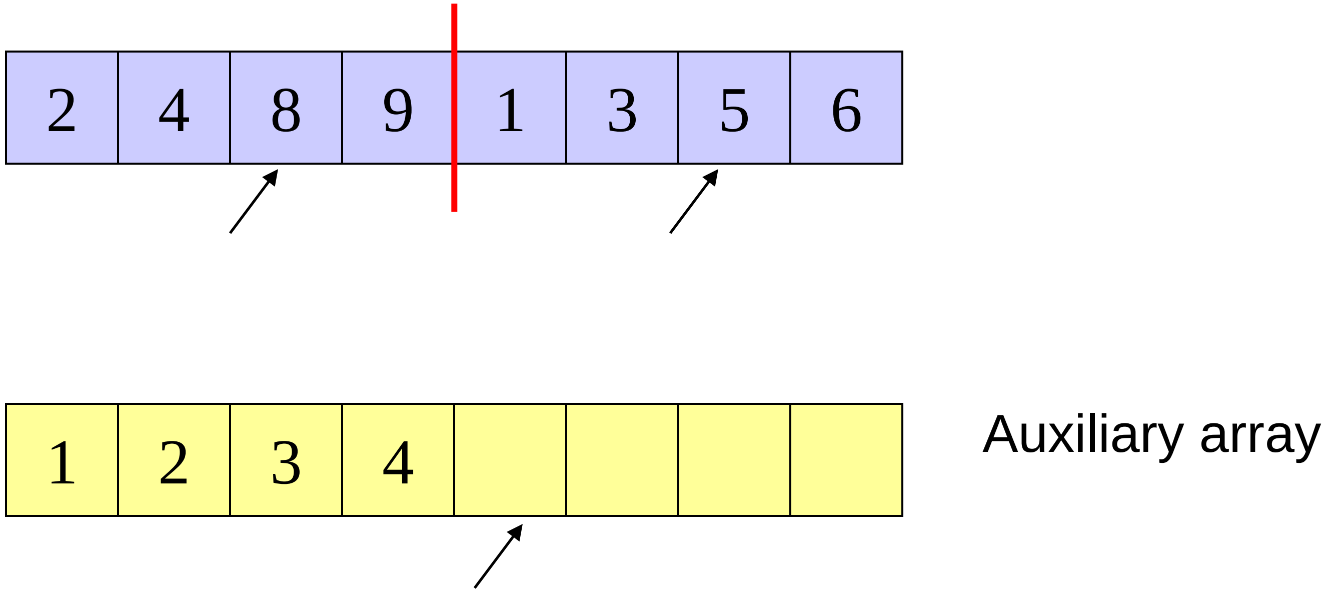
Working of Merge

- The merging requires an auxiliary array.



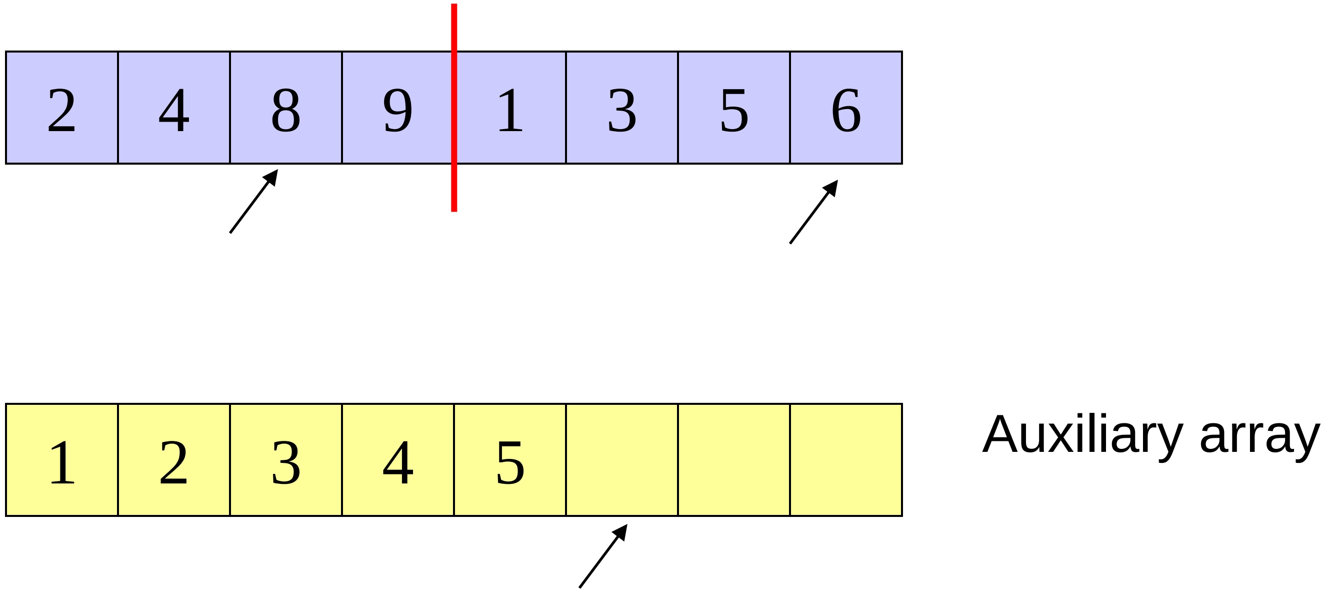
Working of Merge

- The merging requires an auxiliary array.



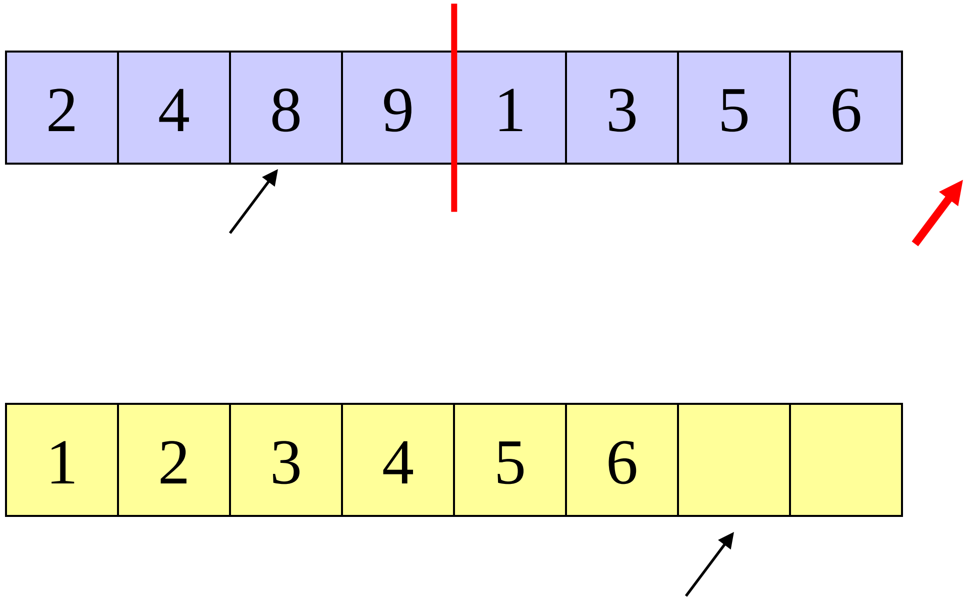
Working of Merge

- The merging requires an auxiliary array.



Working of Merge

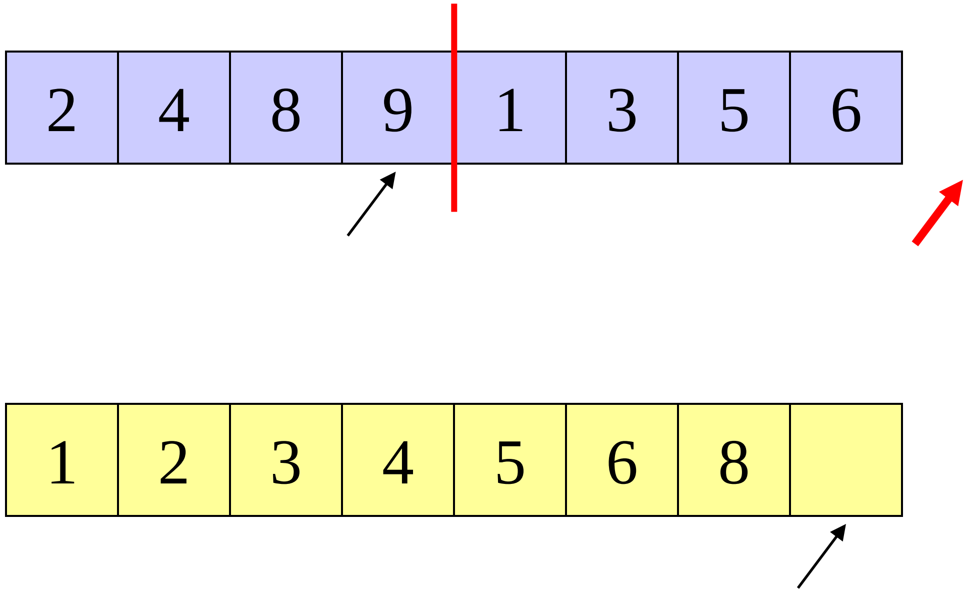
- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

- The merging requires an auxiliary array.

2	4	8	9	1	3	5	6

1	2	3	4	5	6	8	9

Auxiliary array

Merge Sort

- Lets Trace the algorithm

```
Algorithm MergeSort(low, high)
{
        if (low<high) then
        {
          mid= ⌊(low + high)/2⌋
        MergeSort(low, mid)
        MergeSort(mid+1, high)
          Merge(low, mid, high)
        }
    }
```

Find the tracing in the video

Merge Sort -Analysis

- The recurrence relation for Merge sort is given by
- $$T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + cn & n > 1, c \text{ is a constant} \end{cases}$$
- **Solution**

In the given relation a=2, b=2, f(n)=cn , n is power of b so $n=b^k$, $n=2^k$

T(n)=2T(n/2) +cn substitute T(n/2)=2T(n/4)+c(n/2)

= 2[2[T(n/4)+cn/2)] +cn

=4T(n/4)+2cn substitute T(n/4)=2T(n/8)+c(n/4)

=4[2T(n/8)+cn/4)+2cn

=8T(n/8)+3cn

The general pattern ?

= $2^kT(n/2^k)$+kcn $n=2^k$, k=logn

=nT(1)+logncn

= n+cnlogn considering only leading term and ignoring constants we get

T(n)= Θ(nlogn)

Merge Sort -Summary

Properties summarized

- Merge Sort is useful for sorting linked lists.
- Merge Sort is a stable sort which means that the same element in an array maintain their original positions with respect to each other.
- Overall time complexity of Merge sort is $\Theta(n \log n)$.
  - i.e. its best, worst and average case time complexity is $\Theta(n \log n)$.
- It is more efficient as it is in worst case also the runtime is $\Theta(n \log n)$
- The space complexity of Merge sort is $O(n)$. This means that this algorithm takes a lot of space and may slower down operations for the large data sets.
- Merge sort is not in-place sorting

Quick Sort

- Quicksort is the other important sorting algorithm that is based on the divide-and conquer approach.
- Unlike merge sort, which divides its input elements according to their position in the array, quicksort divides them according to their value.
- The idea of array partition is used in this sorting.
- A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\overbrace{A[1] \ldots A[s-1]}^{\text{All are } \le A[s]} \quad A[s] \quad \overbrace{A[s+1] \ldots A[n]}^{\text{All are } \ge A[s]}$$

- Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently

Quick Sort

- Now note the difference between the working of Merge sort and Quick sort
- In Merge sort the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions;
- In Quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Quick Sort

Pivot Element

- There are a number of ways to pick the pivot element.
- In this example, we will use the first element in the array:

5	3	1	9	8	2	4	7

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

Pivot

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a[**j**]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p

i

j

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p i j

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

    Increment **i**   until **a[i] ≥ p**   **[till u get greater number than pivot]**

    Decrement **j**   until **a[j] ≤ v**   **[till u get lesser number than pivot]**

        swap **a[i]** and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p

i

j

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a**[j]**

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p

i

j

stop

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a[**j**]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p

i stop

swap

j stop

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

p

i

j

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

    Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

    Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

        swap **a[i]** and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

p

i **j**

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a**[j]**

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

p

stop i j stop

swap

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7

p

i j

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap **a[i]** and a**[j]**

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7

p

j i

swap

stop

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap a[i] and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

p

stop

j i

swap

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap a[i] and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7
p				j	i		

stop

swap

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap a[i] and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7

p

swap

j

i

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j= r+1**

Following are the rules

repeat

  Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]**

    swap a[i] and a[j]

until i ≥ j

swap (a[i],a[j])

swap(a[l],a[j])

return j

0	1	2	3	4	5	6	7
2	3	1	4	5	8	9	7

p

swap

j

i

Quick Sort

Do it yourself

- Obtain the first partition for the following set of elements considering the first element as the pivot element

65, 70, 75, 80, 85, 60, 55, 50, 45

- Apply quicksort to sort the list **E, X, A,M, P, L, E** in alphabetical order

Quick Sort

ALGORITHM *Quicksort*($A[l..r]$)

//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
// l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
    $s \leftarrow Partition(A[l..r])$ //s is a split position
    $Quicksort(A[l..s-1])$
    $Quicksort(A[s+1..r])$

Quick Sort

ALGORITHM *Partition*($A[l..r]$)

//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r ($l < r$)
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value

$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$
    swap($A[i]$, $A[j]$)
until $i \geq j$
swap($A[i]$, $A[j]$) //undo last swap when $i \geq j$
swap($A[l]$, $A[j]$)
return j

Quick Sort –Analysis (Best Case)

- The recurrence relation is given by
- $T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + n & n > 1, c \text{ is a constant} \end{cases}$
- **Solution**

In the given relation a=2, b=2, f(n)=cn , n is power of b so $n=b^k$, $n=2^k$

T(n)=2T(n/2) +n substitute T(n/2)=2T(n/4)+(n/2)

= 2[2[T(n/4)+n/2)] +n

=4T(n/4)+2n substitute T(n/4)=2T(n/8)+(n/4)

=4[2T(n/8)+n/4)+2n

=8T(n/8)+3n

The general pattern ?

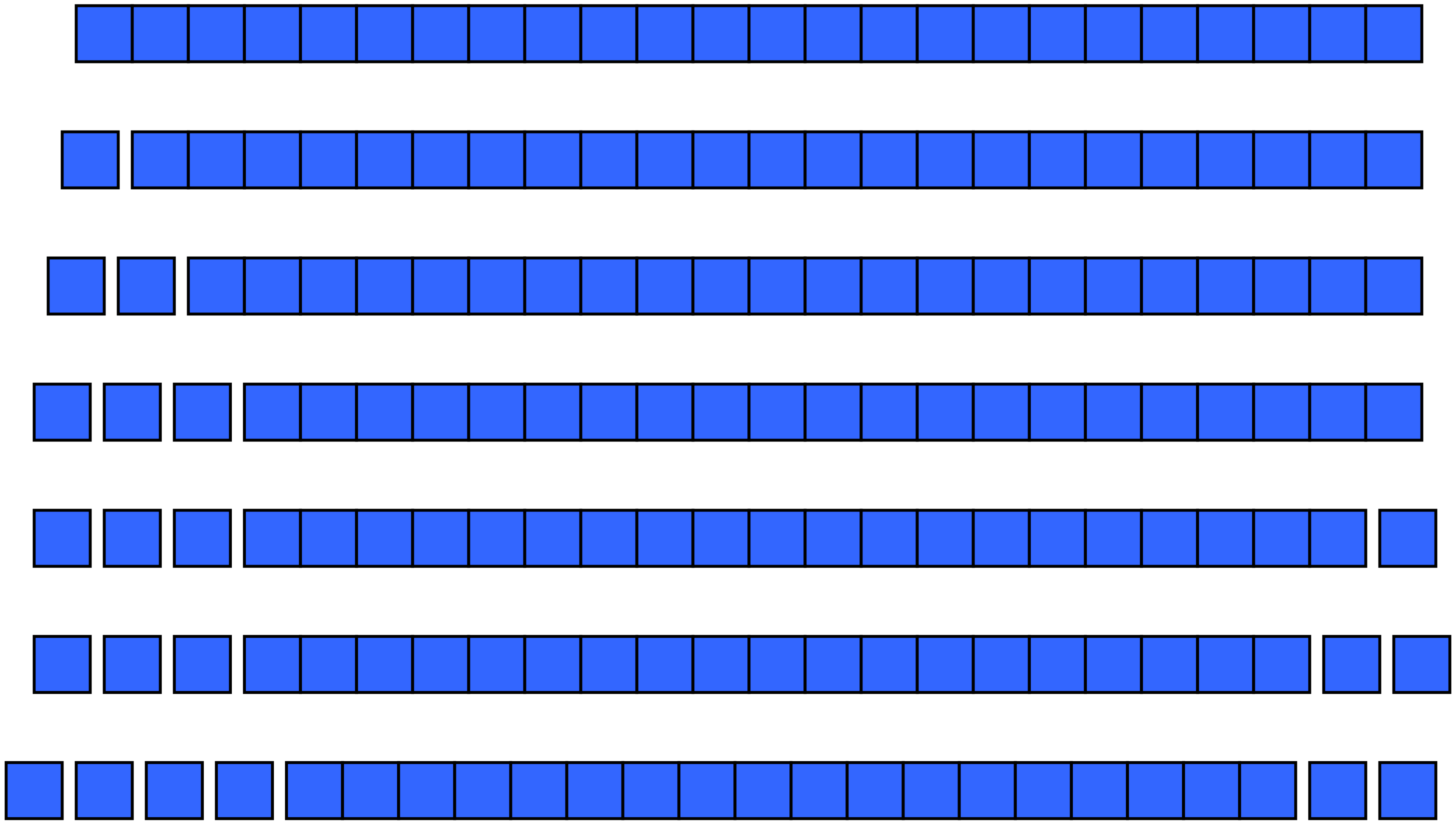
= $2^kT(n/2^k)$+kn $n=2^k$, k=logn

=nT(1)+logn n

= n+cnlogn considering only leading term and ignoring constants we get

$T(n)_{Best} = \Theta(n\log n)$

Quick Sort- Analysis (Worst Case)



Quick Sort- Analysis (Worst Case)

- The recurrence relation for worst case analysis is given by

$T(n)=0+T(n-1)+n$

Average case

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n.$$

Design and Analysis of Algorithms

ESTD:2001

An Institute with a Difference

Divide and Conquer

Manjula L

Asst. Prof. Dept. of CSE

RNSIT, Bengaluru, India

Strassen's Matrix Multiplication

- Let A and B be two n x n matrices
- The product matrix C= AB is also an n x n matrix whose i, jthelement is formed by taking the elements in the ith row of **A**and jth column of **B** and multiplying them to get
- $C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$for all i and j between 1 and n
- To compute C(i, j) using the formula above how many multiplications are needed?
- Consider an example

$$A=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \text{ then } C=\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

- Where,

$c_{11}=a_{11}b_{11}+a_{12}b_{21}$

$c_{12}=a_{11}b_{12}+a_{12}b_{22}$

$c_{21}=a_{21}b_{11}+a_{22}b_{21}$

$c_{22}=a_{21}b_{12}+a_{22}b_{22}$

8 multiplications

Time complexity ? **$\Theta(n^3)$**

Strassen's Matrix Multiplication

- Can we use **Divide and Conquer** approach to multiply two **n x n** matrices ?
- Let's assume that **n** is power of **2**, i.e., there exists a non-negative constant **k** such that $\mathbf{n=2^k}$
- If **n** is not power of **2** then add enough rows and columns of **zeros** to both **A** and **B** so that the resultant dimensions are power of **2**.
- Here is the application of Divide and Conquer approach

$$A = \begin{array}{cc} M & N \\ \multicolumn{2}{c}{\left[\begin{array}{c|c} \quad & \quad \\ \hline \quad & \quad \end{array}\right]} \\ O & P \end{array} \qquad B = \begin{array}{cc} Q & R \\ \multicolumn{2}{c}{\left[\begin{array}{c|c} \quad & \quad \\ \hline \quad & \quad \end{array}\right]} \\ S & T \end{array}$$

$$APrep = \left[\begin{array}{cc|cc} 0 & 8 & 0 & 0 \\ 10 & 1 & 0 & 0 \\ \hline 8 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

$$BPrep = \left[\begin{array}{cc|cc} 9 & 6 & 4 & 8 \\ -3 & 2 & 7 & 1 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

$$a = \begin{bmatrix} 0 & 8 \\ 10 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad c = \begin{bmatrix} 8 & 6 \\ 0 & 0 \end{bmatrix} \quad d = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$e = \begin{bmatrix} 9 & 6 \\ -3 & 2 \end{bmatrix} \quad f = \begin{bmatrix} 4 & 8 \\ 7 & 1 \end{bmatrix} \quad g = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad h = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Strassen's Matrix Multiplication

- Consider the following situation

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

- Then

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

- Where

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Strassen's Matrix Multiplication

- Consider the following matrices and compute the product using Strassen's Method
- A= $\begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix}$ B= $\begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix}$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

P= (2+5) (1+7)= 7* 8= 56

Q= (3+5)* 1= 8

R= 2* (3-7)= -8

S= 5*(4-1)= 15

T= (2+4) *7= 42

U= (3-2)* (1+3)= 4

V= (4-5)* (4+7)=-11

C11 = 56+ 15-42-11= 18

C12= -8+ 42= 34

C21= 8+15 = 23

C22= 56-8 -8+4 = 44

Strassen's 4X4 Matrix Multiplication

$$\begin{pmatrix} 5 & 2 & 6 & 1 \\ 0 & 6 & 2 & 0 \\ 3 & 8 & 1 & 4 \\ 1 & 8 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 5 & 8 & 0 \\ 1 & 8 & 2 & 6 \\ 9 & 4 & 3 & 8 \\ 5 & 3 & 7 & 9 \end{pmatrix} = \begin{pmatrix} 96 & 68 & 69 & 69 \\ 24 & 56 & 18 & 52 \\ 58 & 95 & 71 & 92 \\ 90 & 107 & 81 & 142 \end{pmatrix}$$

I now want to use strassen's method which I learned as follows:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

Time efficiency

- M(n)= 7 M(n/2)   for n>1, M(1)=1
- Since n= 2 ^ k

- M(2^ K)= 7 M (2^ (k-1))

- 7 ^ i M(2 ^ (k-k)
- 7^k
- K=log2 n
- n ^ 2.807

Pros of Divide and Conquer Strategy

- Solving difficult problems
- Algorithm efficiency
- Parallelism – Suitable for multiprocessor machines
- Memory access - *optimal* cache-oblivious algorithms

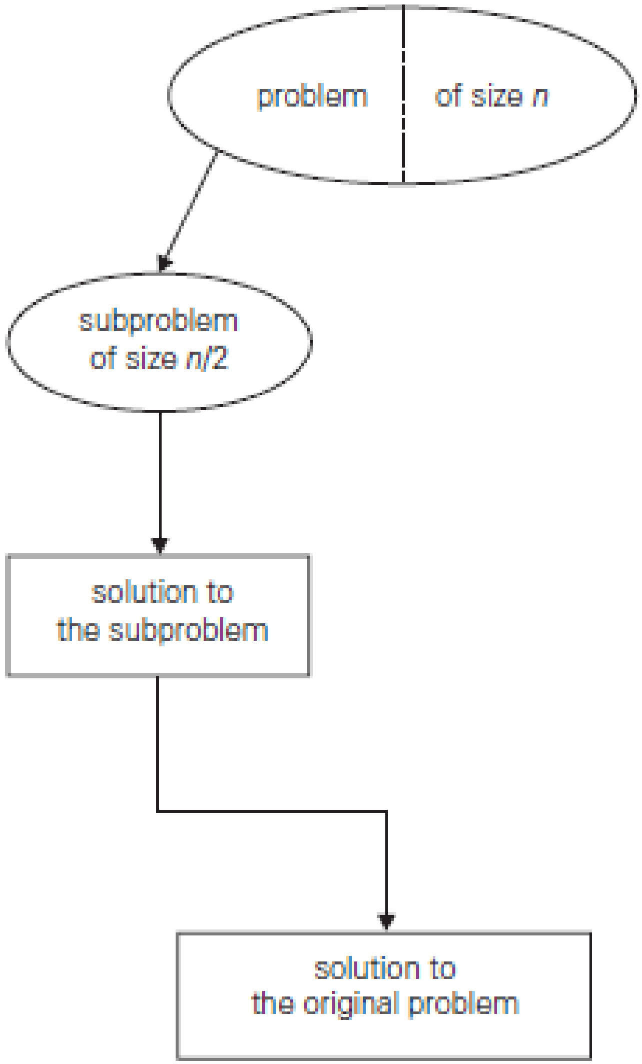
Cons of Divide and Conquer Strategy

- **Divide and Conquer** strategy **uses** recursion that makes it a little slower and if a little error occurs in the code the program may enter into an infinite loop.
- Usage of explicit stacks may make **use** of extra space.

Decrease and Conquer

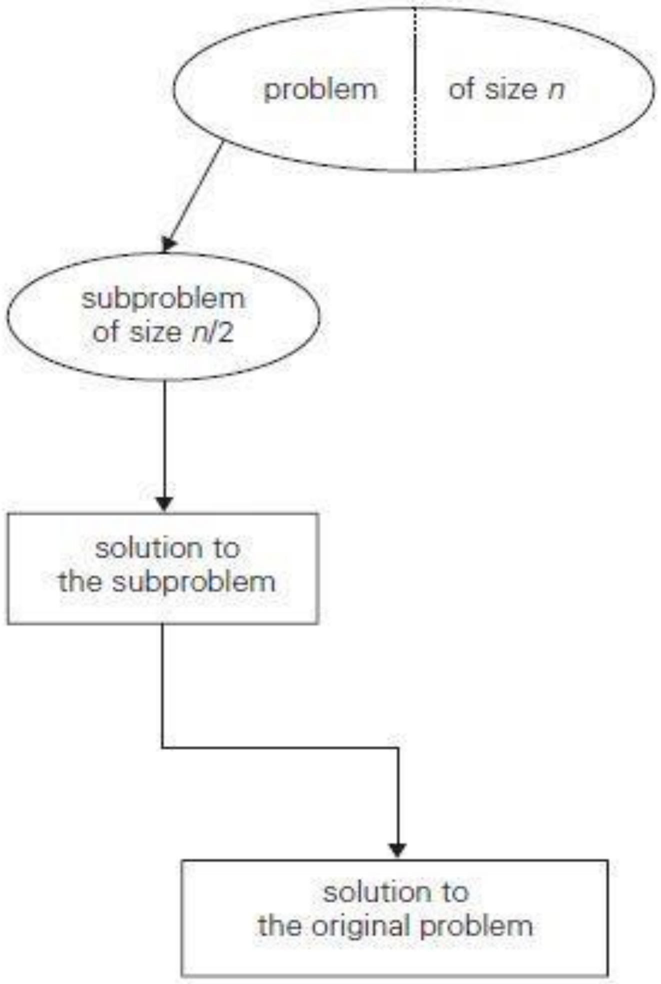
- This technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion).
- There are three major variations of decrease-and-conquer:
  - Decrease by a constant.
  - Decrease by a constant factor.
  - Variable size decrease.

Decrease by a constant



$$f(n) = \begin{cases} f(n-1)\cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

Decrease by a constant Factor



$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

Variable size decrease

- $gcd(m, n) = gcd(n, m \bmod n)$.

THANK YOU