DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

DECREASE AND CONQUER

Surabhi Narayan

Department of Computer Science & Engineering

Decrease and Conquer

- **Decrease** or reduce problem instance to smaller instance of the same problem and extend solution.
- **Conquer** the problem by solving smaller instance of the problem.
- **Extend** solution of smaller instance to obtain solution to original problem .
- **Exploit** the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

Decrease and Conquer

3 Types of Decrease and Conquer

Decrease by a constant (usually by 1):
- insertion sort
- graph traversal algorithms (DFS and BFS)
- topological sorting
- algorithms for generating permutations, subsets

Decrease by a constant factor (usually by half)
- binary search and bisection method
- exponentiation by squaring
- multiplication à la russe

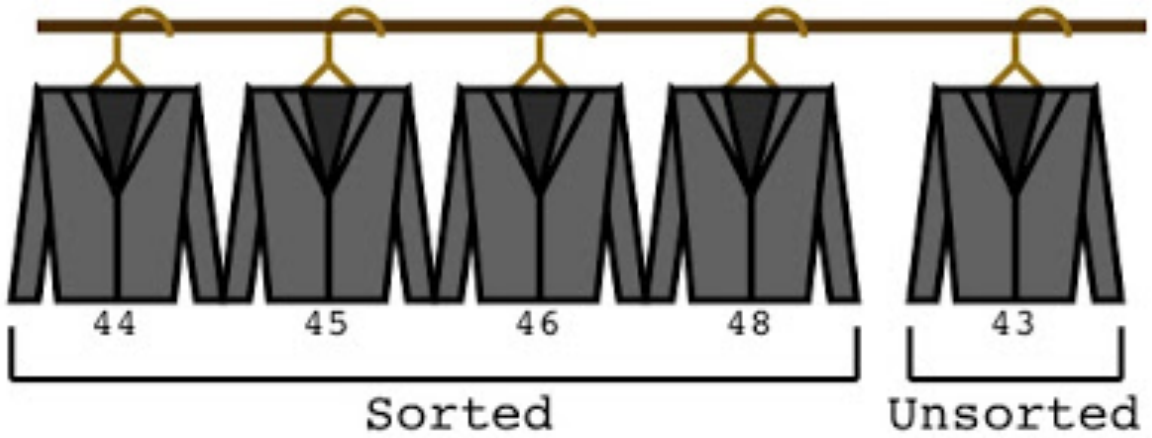
Variable-size decrease
- Euclid's algorithm
- selection by partition
- Nim-like games

This usually results in a recursive algorithm.

Insertion Sort

Imagine a card game

Cards in your hand are sorted.
The dealer hands you exactly one new card.
How would you rearrange your cards



Insertion Sort

- Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.
- grows the sorted array at each iteration
- compares the current element with the largest value in the sorted array.
- If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position.
- This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Insertion Sort

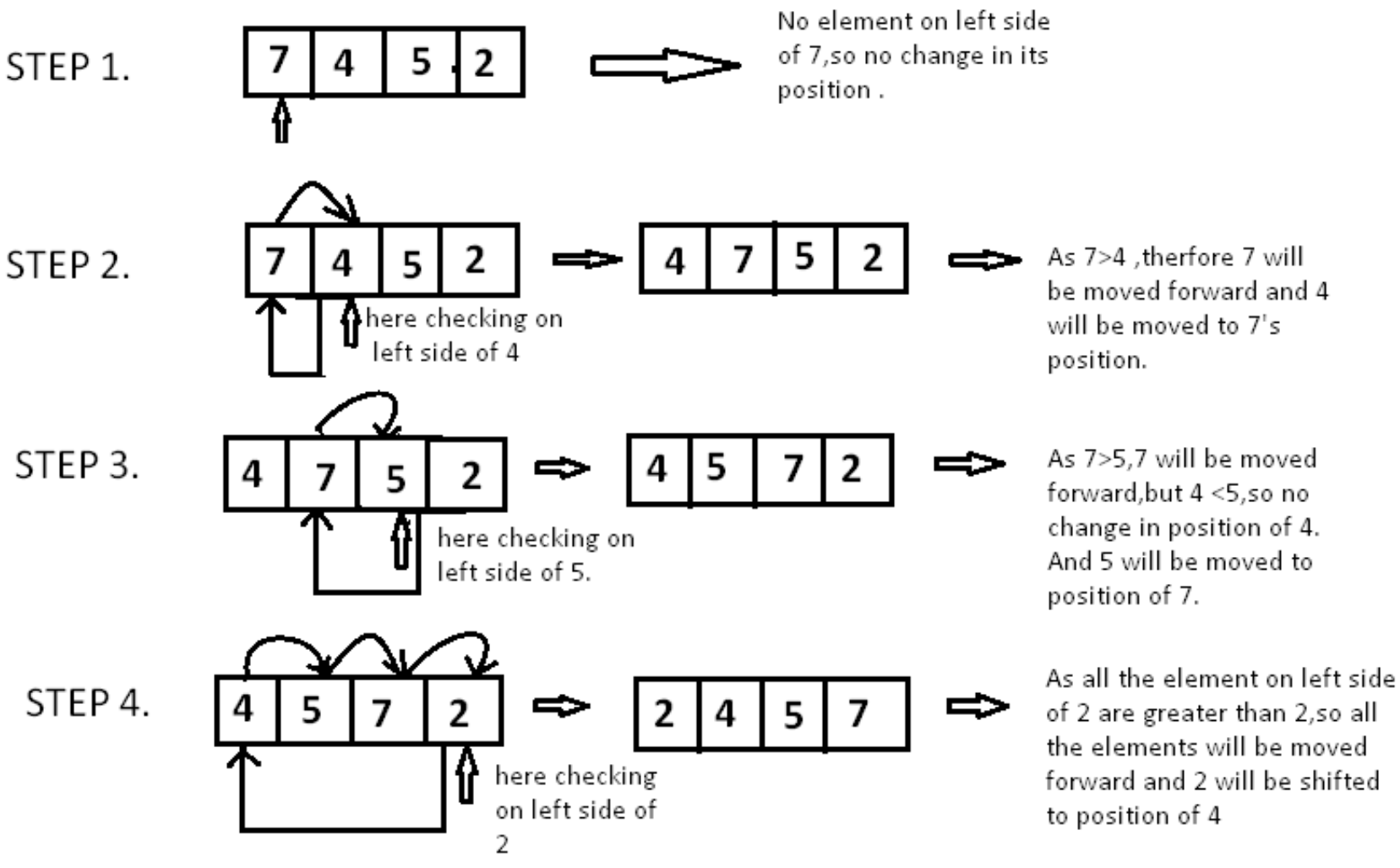
To sort array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]

Usually implemented bottom up (nonrecursively)

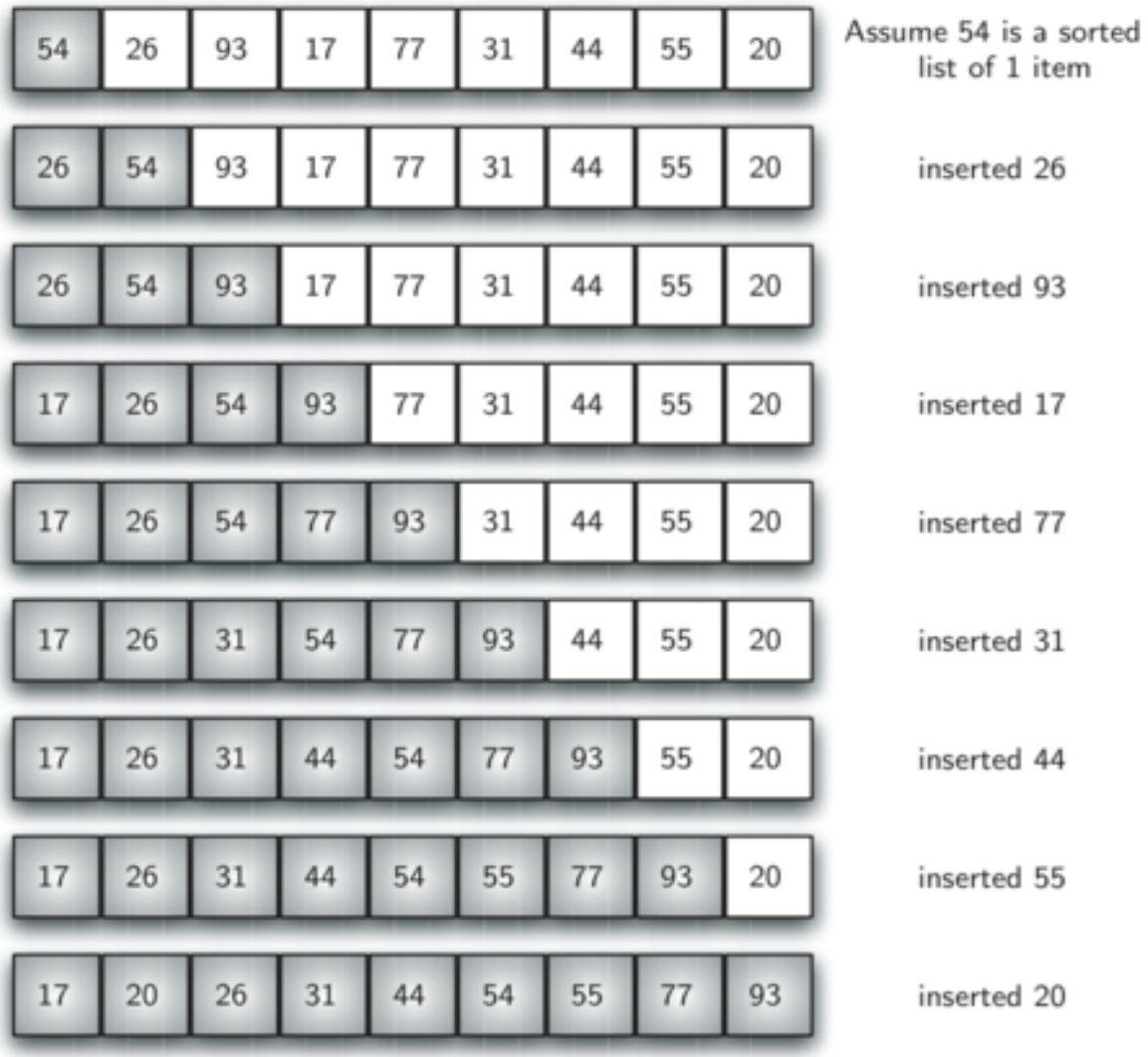
Example: Sort 6, 4, 1, 8, 5

6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
1 4 5 6 8

Insertion Sort



Insertion Sort



ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Time efficiency

$C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

$C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

$C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost sorted arrays)

Space efficiency: in-place

Stability: yes

Best elementary sorting algorithm overall

Binary insertion sort

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

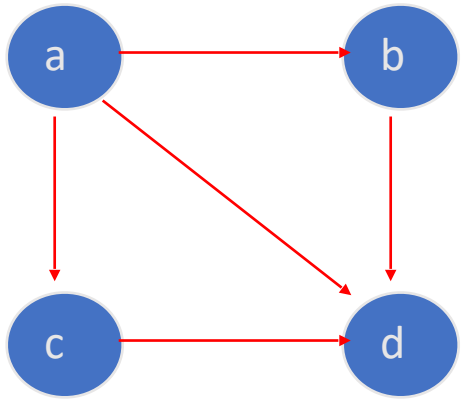
DECREASE AND CONQUER

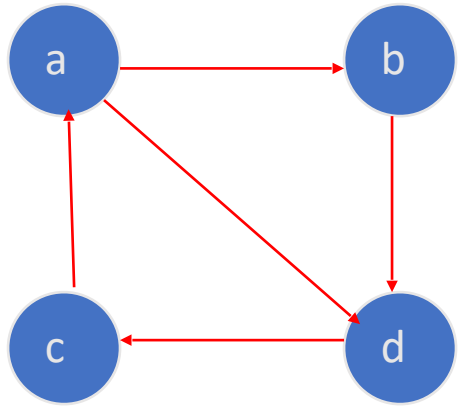
Surabhi Narayan

Department of Computer Science & Engineering

DAGs and Topological Sorting

DAG: a directed acyclic graph, i.e. a directed graph with no (directed) cycles





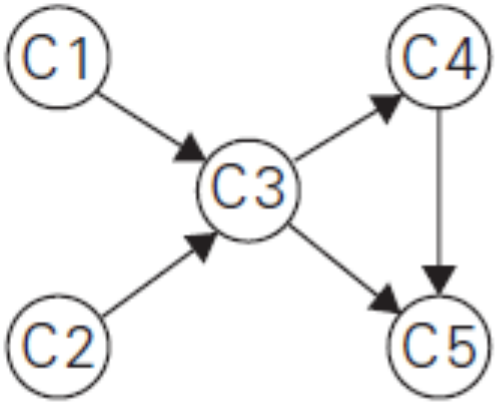
Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

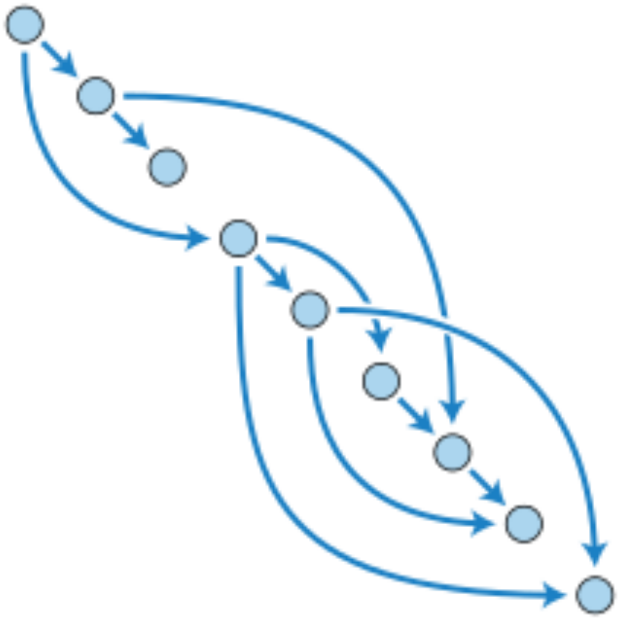
Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (*topological sorting*). Being a dag is also a necessary condition for topological sorting to be possible.

Topological Sorting

Topological Sorting: is listing vertices of a directed graph in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

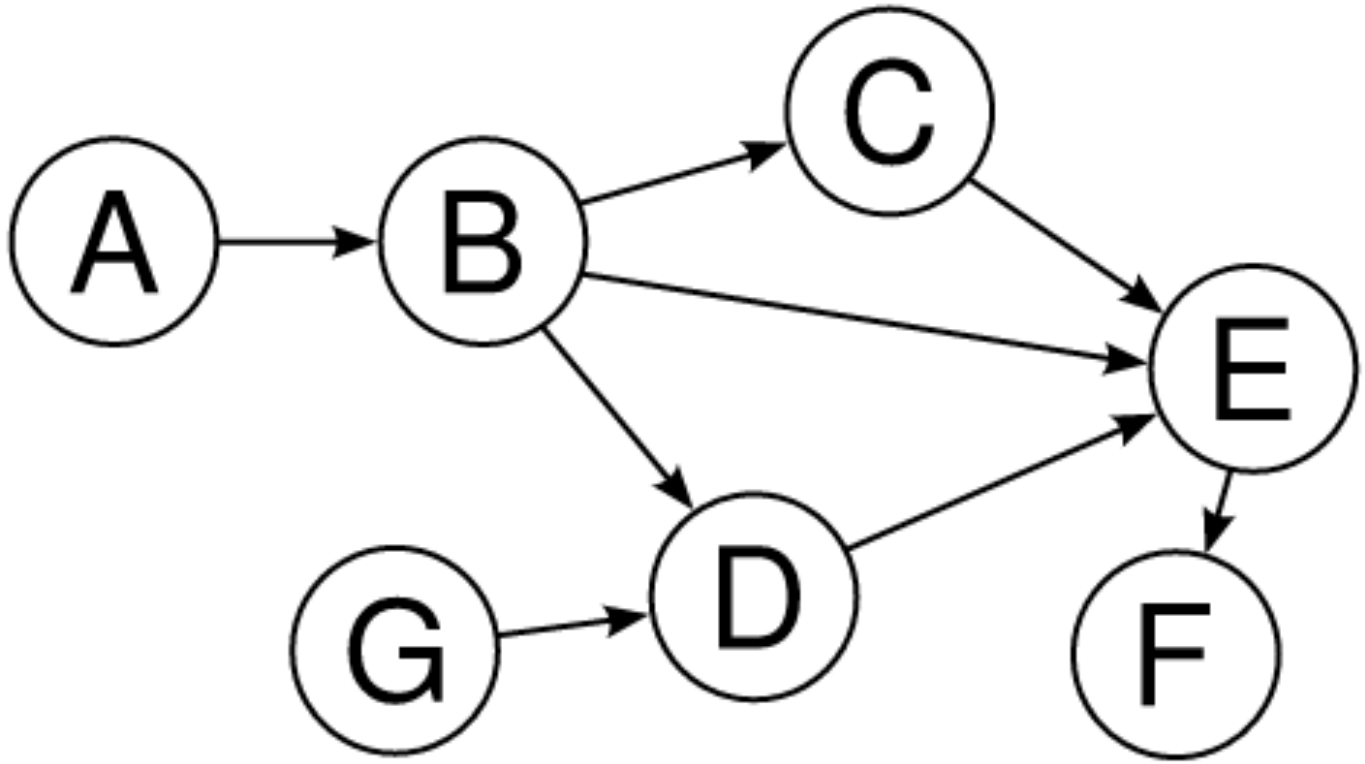
A digraph has a topological sorting iff it is a **dag**.





Finding a **Topological Sorting** of the vertices of a dag:

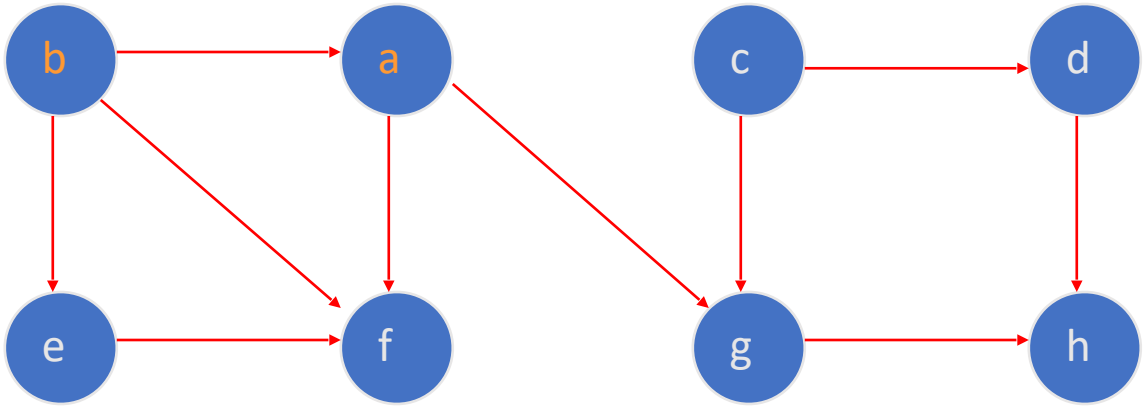
- **DFS-based** algorithm
- **Source-removal** algorithm



DFS-based algorithm for topological sorting

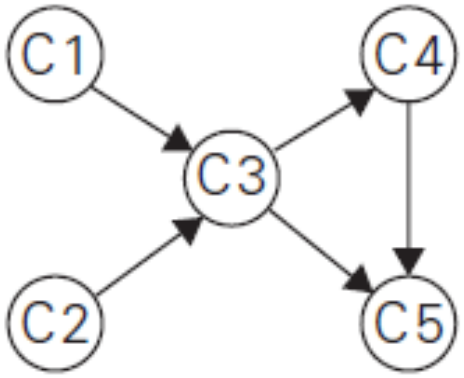
- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

Example:



Efficiency: The same as that of DFS.

DFS-based algorithm for finding **Topological Sorting**



(a)

$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

(b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:

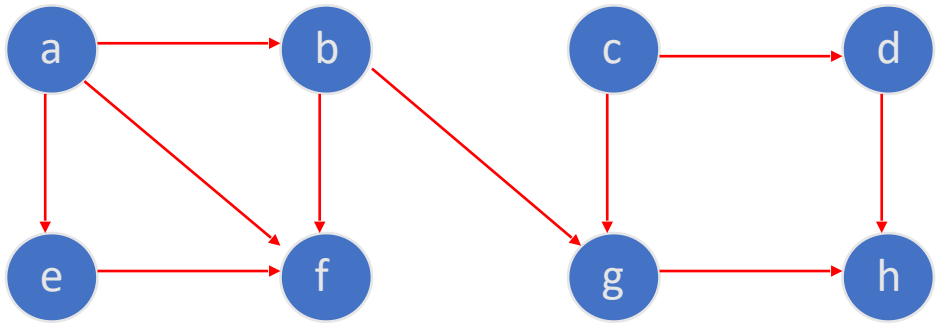
(c)

Decrease and Conquer

Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left or there is no source among the remaining vertices (not a dag)

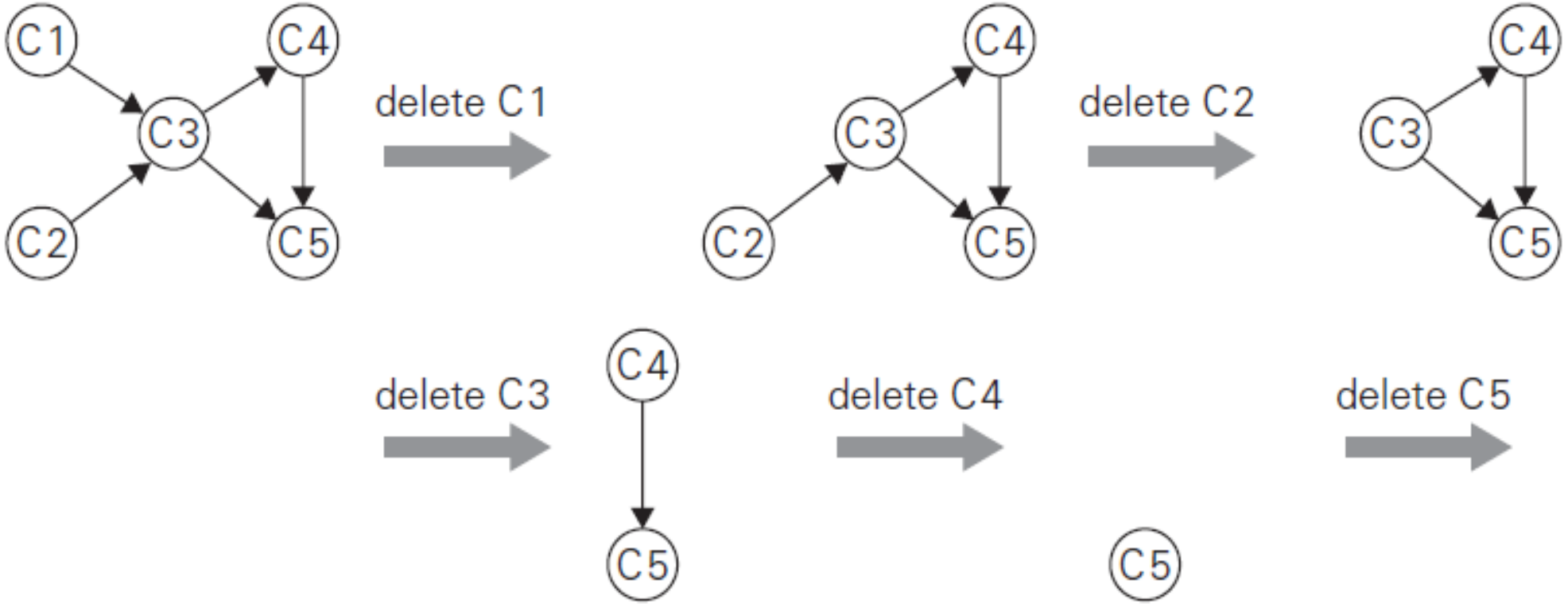
Example:



"Invert" the adjacency lists for each vertex to count the number of incoming edges by going thru each adjacency list and counting the number of times that each vertex appears in these lists. To remove a source, decrement the count of each of its neighbors by one.

Algorithm SourceRemoval_Toposort(V, E)
L ← Empty list that will contain the sorted vertices
S ← Set of all vertices with no incoming edges
while S is non-empty **do**
    remove a vertex v from S
    add v to *tail* of L
    for each vertex m with an edge *e* from v to m **do**
        remove edge e from the graph
        if m has no other incoming edges **then**
            insert m into S
if graph has edges **then**
    return error (not a DAG)
else return L (a topologically sorted order)

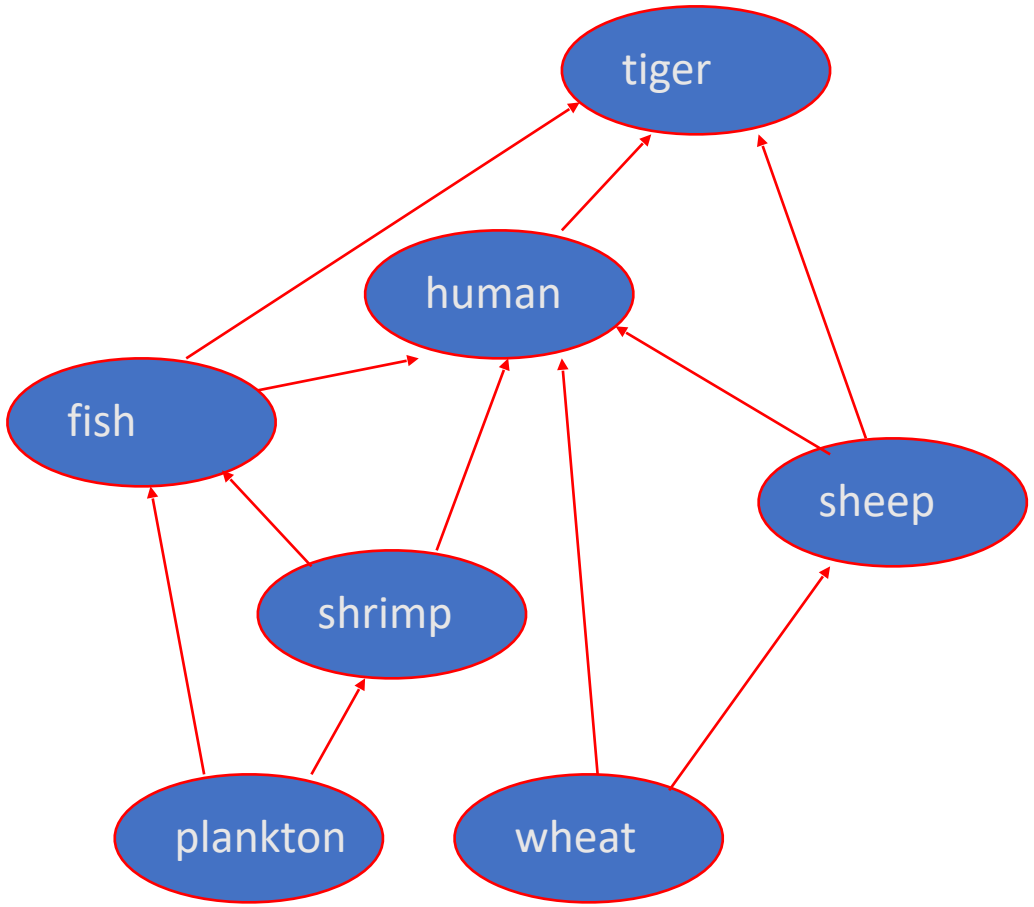
Decrease and Conquer



The solution obtained is C1, C2, C3, C4, C5

PES UNIVERSITY ONLINE

Order the following items in a food chain



THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

DECREASE AND CONQUER

Surabhi Narayan

Department of Computer Science & Engineering

Combinatorial Objects

- Permutations
- Combinations
- Subsets of a given set

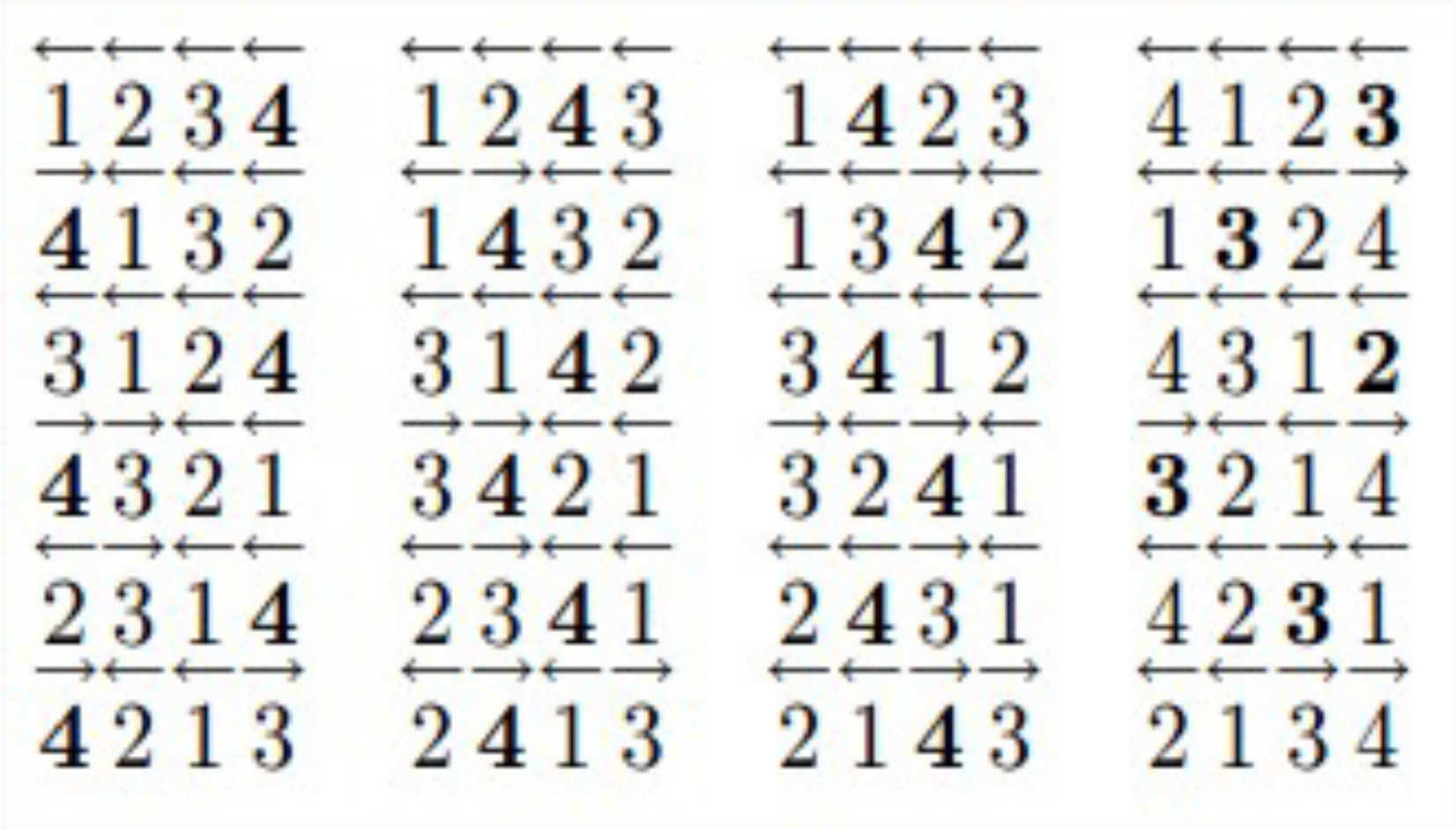
Generating Permutations

- Underlying set elements are to be permuted
- Decrease and conquer approach
- Satisfies the minimal change requirement
- Example: Johnson- Trotter algorithm

Generating Permutations

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list


1 2 3 4
4 1 3 2
3 1 2 4
4 3 2 1
2 3 1 4
4 2 1 3
1 2 4 3
1 4 3 2
3 1 4 2
3 4 2 1
2 3 4 1
2 4 1 3
1 4 2 3
1 3 4 2
3 4 1 2
3 2 4 1
2 4 3 1
2 1 4 3
4 1 2 3
1 3 2 4
4 3 1 2
3 2 1 4
4 2 3 1
2 1 3 4

Generating Subsets:

Knapsack problem needed to find the most valuable subset of items that fits a knapsack of a given capacity.

Powerset: set of all subsets of a set. Set A={1, 2, ..., n} has $\mathbf{2^n}$ subsets.

Generate all subsets of the set A={1, 2, ..., n}.

Any **decrease-by-one** idea?
# of subsets of { } = $2^0 = 1$, which is **{ }** itself
Suppose, we know how to generate all subsets of {1,2,...,n-1}
Now, how can we generate all subsets of {1,2,...,n} ?

Generating Subsets:

All subsets of {1,2,...,n-1}: 2^{n-1} such subsets

All subsets of {1,2,...,n}:
2^{n-1} subsets of {1,2,...,n-1} and
another 2^{n-1} subsets of {1,2,...,n-1} having **'n'** with them.

That adds up to all 2^n subsets of {1,2,...,n}

0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Alternate way of Generating Subsets:

Knowing the binary nature of either having **n**th element or not, any idea involving binary numbers itself?

One-to-one correspondence between all 2^n bit strings $b_1b_2...b_n$ and 2^n subsets of $\{a_1, a_2, \ldots, a_n\}$.

Each bit string $b_1b_2...b_n$ could correspond to a subset.
In a bit string $b_1b_2...b_n$, depending on whether b_i is 1 or 0, a_i is in the subset or not in the subset.

000	001	010	011	100	101	110	111
$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Generating Subsets in Squashed order:

Squashed order: any subset involving a_j can be listed only after all the subsets involving $a_1, a_2, \ldots, a_{j-1}$

Both of the previous methods does generate subsets in squashed order.

000	001	010	011	100	101	110	111
$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Generating Subsets in Squashed order:

Squashed order: any subset involving a_j can be listed only after all the subsets involving $a_1, a_2, \ldots, a_{j-1}$

Can we do it with minimal change in bit-string (actually, just one-bit change to get the next bit string)? This would mean, to get a new subset, just change one item (remove one item or add one item).

Binary reflected gray code:

000 001 011 010 110 111 101 100

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

DECREASE AND CONQUER

Surabhi Narayan

Department of Computer Science & Engineering

Fake-Coin Problem

Among n identical-looking coins, one is fake.With a balance scale, we can compare any two sets of coins.

The problem is to design an efficient algorithm for detecting the fake coin.

The most natural idea for solving this problem is to divide n coins into two piles of $n/2$ coins each, leaving one extra coin aside if n is odd, and put the two piles on the scale.

If the piles weigh the same, the coin put aside must be fake; otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

$W(n) = W(n/2) + 1$ for $n > 1$, $W(1) = 0$.

$W(n) = \log_2 n$.

It would be more efficient to divide the coins not into two but into *three* piles of about $n/3$ coins each.

Russian Peasant Multiplication

Let n and m be positive integers whose product we want to compute, and let us measure the instance size by the value of n.
if n is even, an instance of half the size has to deal with $n/2$, and we have an obvious formula relating the solution to the problem's larger instance to the solution to the smaller one:
$n \cdot m = (n/2) * 2m$

If n is odd, we need only a slight adjustment of this formula:
$n \cdot m = ((n-1)/2) \cdot 2m + m$.
Using these formulas and the trivial case of $1 \cdot m = m$ to stop, we can compute product $n \cdot m$ either recursively or iteratively.

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

(b)

FIGURE 4.11 Computing 50 · 65 by the Russian peasant method.

Josephus problem

Named for Flavius Josephus, a famous Jewish historian who participated in and chronicled the Jewish revolt of 66–70 c.e. against the Romans.
Josephus, as a general, managed to hold the fortress of Jotapata for 47 days, but after the fall of the city he took refuge with 40 diehards in a nearby cave. There, the rebels voted to perish rather than surrender. Josephus proposed that each man in turn should dispatch his neighbor, the order to be determined by casting lots. Josephus contrived to draw the last lot, and, as one of the two surviving men in the cave, he prevailed upon his intended victim to surrender to the Romans.

Josephus Problem

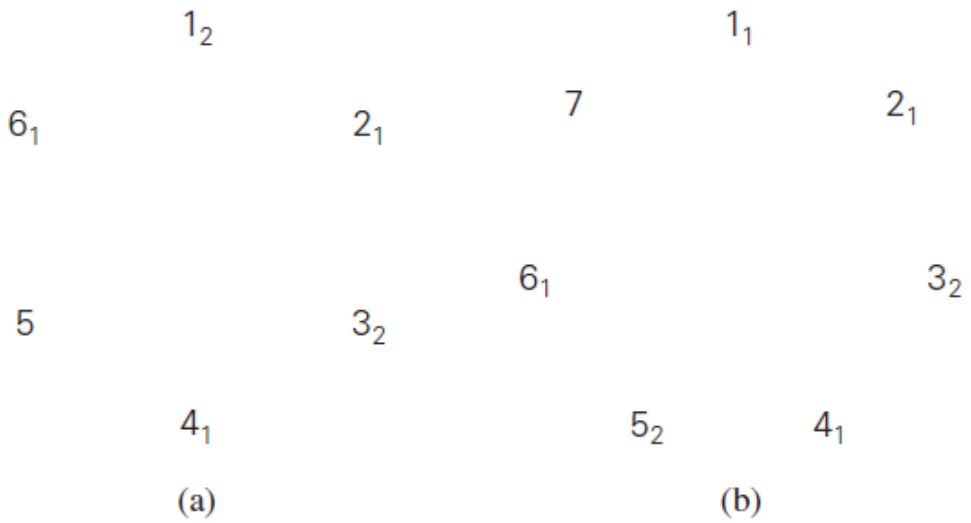


FIGURE 4.12 Instances of the Josephus problem for (a) $n = 6$ and (b) $n = 7$. Subscript numbers indicate the pass on which the person in that position is eliminated. The solutions are $J(6) = 5$ and $J(7) = 7$, respectively.

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS
UE19CS251

Shylaja S S
Department of Computer Science
& Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Divide and Conquer: Binary Search

Major Slides Content: Anany Levitin

Shylaja S S

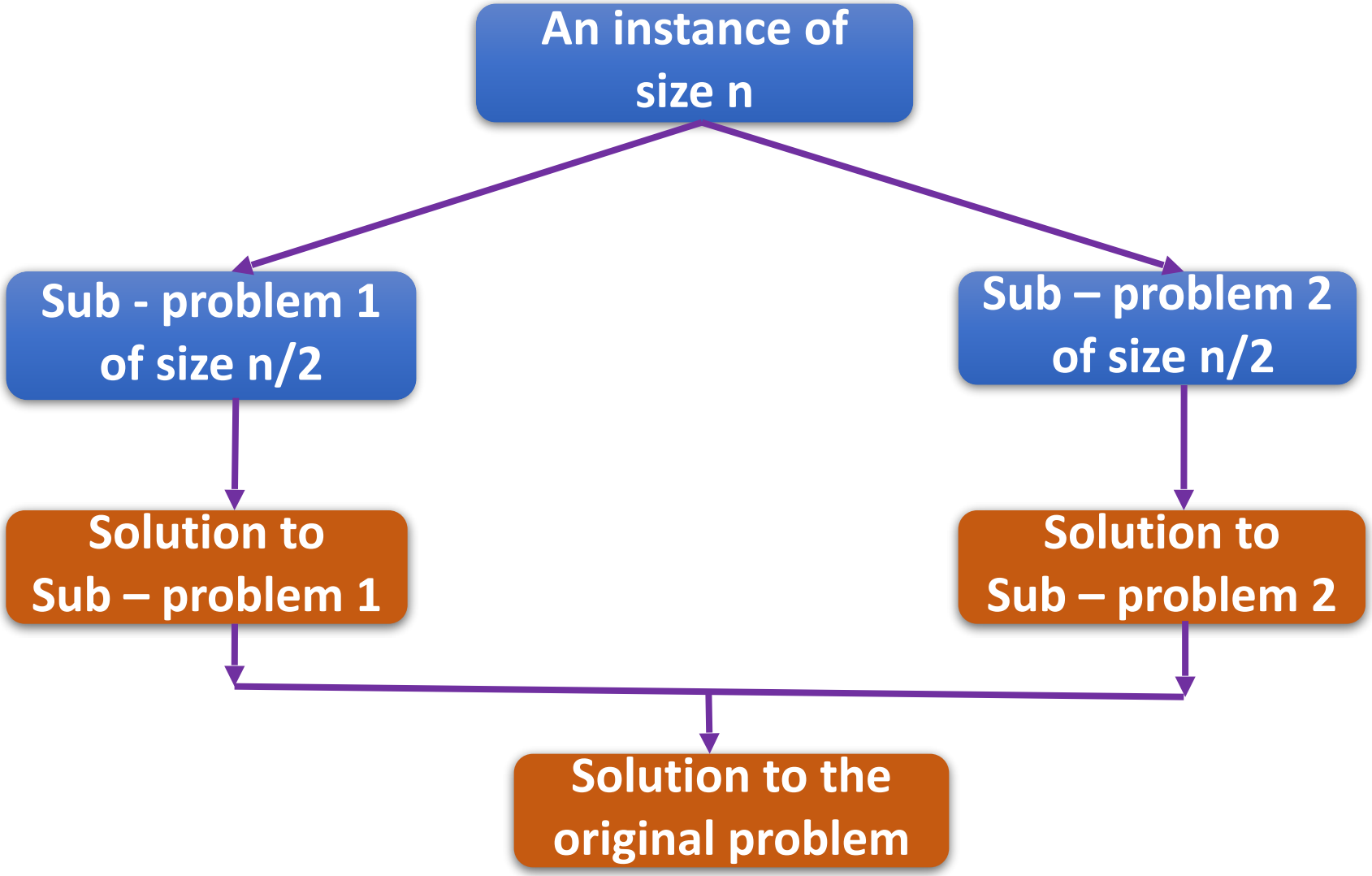
Department of Computer Science & Engineering

Divide and Conquer – Idea

- Divide and Conquer is one of the most well – known algorithm design strategies
- The principle underlying Divide and Conquer strategy can be stated as follows:
  - Divide the given instance of the problem into two or more smaller instances
  - Solve the smaller instances recursively
  - Combine the solutions of the smaller instances and obtain the solution for the original instance

Divide and Conquer – Idea

- Divide and Conquer



General Divide and Conquer

Recurrence

- In the most typical cases of Divide and Conquer, a problem's instance of size n can be divided into b instances of size n/b, with a of them needing to be solved
- Here a and b are constants; a >= 1 and b >= 1
- Assuming that size n is a power of b, we get the following recurrence for the running time:

$$T(n) = a * T(n/b) + f(n)$$

- f(n) is a function that accounts for the time spent on dividing the problem and combining the solutions

Master Theorem

Recurrence

- For the recurrence:

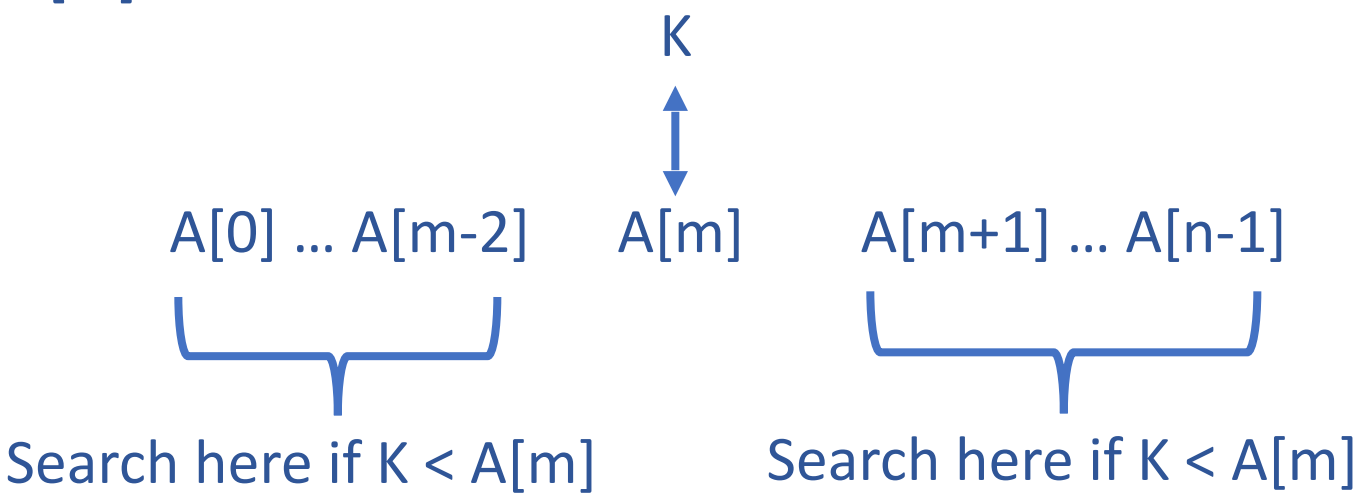
$$\mathbf{T(n) = a * T(n/b) + f(n)}$$

- If $f(n) \in \Theta(n^d)$, where d >= 0 in the recurrence relation, then:
  - If $a < b^d$, $T(n) \in \Theta(n^d)$
  - If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
  - If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

- Analogous results hold for O and Ω as well!

Binary Search - Idea

- Binary Search is a remarkably efficient algorithm for searching in a sorted array
- It works by comparing the 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]



Binary Search - Algorithm

ALGORITHM BinarySearch(A[0 .. n -1], K)
// Implements non recursive binary search
// Input: An array A[0 .. n - 1] sorted in ascending order and
a // search key K
// Output: An index of the array's element that is equal to K
or // -1 if there is no such element
l ← 0; r ← n-1
while l ≤ r do
$m \leftarrow \lfloor (l + r)/2 \rfloor$
if K = A[m] return m
else if K < A[m] r ← m-1
else l←m+1
return -1

DESIGN AND ANALYSIS OF ALGORITHMS

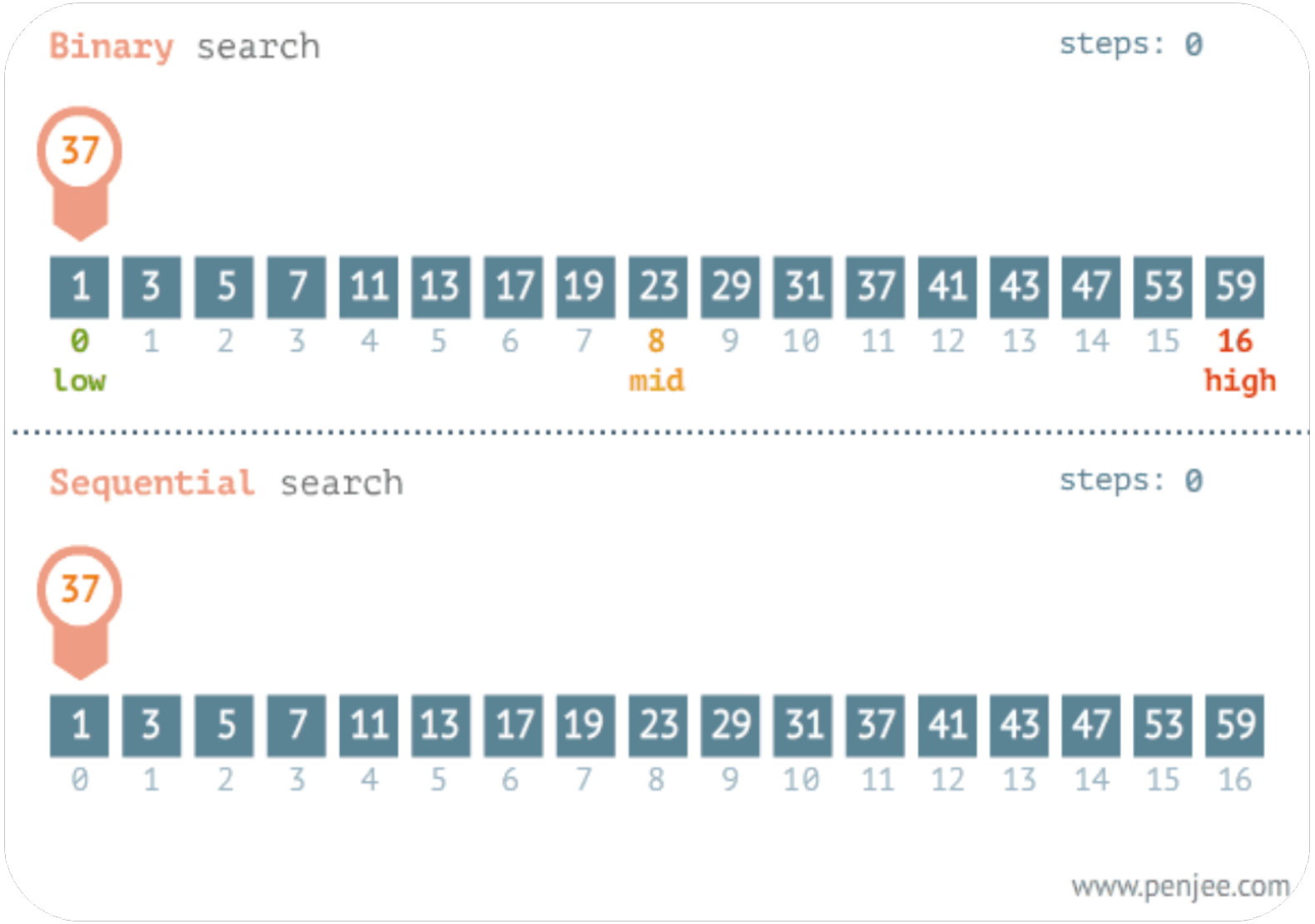
Binary Search - Example

Search Key K = 70

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
iteration1	*l*						*m*						*r*
iteration2								*l*		*m*			*r*
iteration3								*l,m*	*r*				

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Search Vs Linear Search



Binary Search - Analysis: Worst Case

The basic operation is the comparison of the search key with an element of the array

The number of comparisons made are given by the following recurrence:

$$C_{worst}(n) = C_{worst}\left(\left\lfloor n/2 \right\rfloor\right) + 1 \; for \; n > 1, C_{worst}(1) = 1$$

For the initial condition $C_{worst}(1) = 1$, we obtain:

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1$$

For any arbitrary positive integer, n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$$

Binary Search - Analysis: Average Case

$$C_{avg} \approx \log_2 n$$

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS
UE19CS251

Shylaja S S
Department of Computer Science
& Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Merge Sort

Major Slides Content: Anany Levitin

Shylaja S S

Department of Computer Science & Engineering

Merge Sort - Idea

- Split array A[0..n-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A

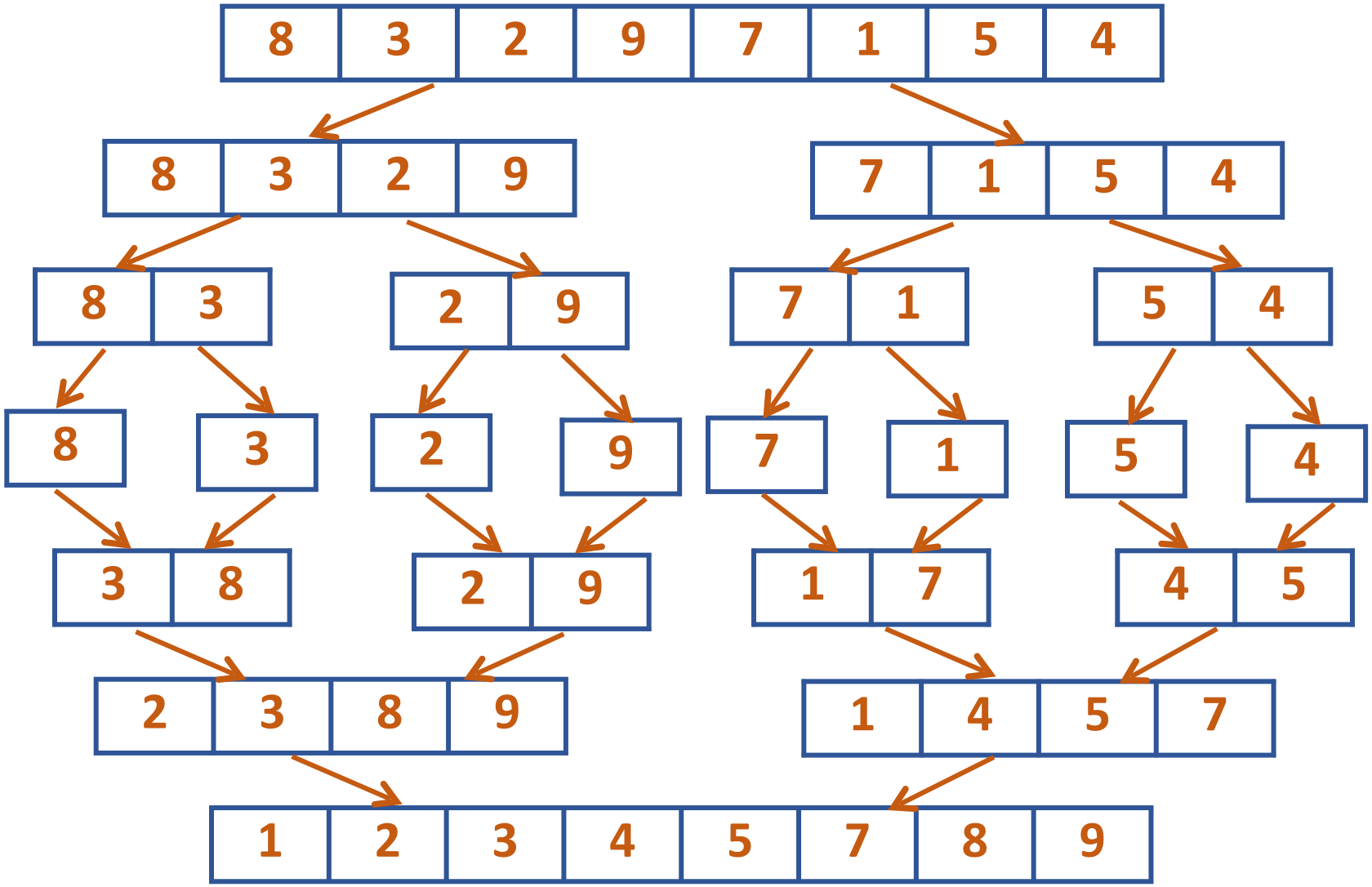
Merge Sort - Algorithm

ALGORITHM Mergesort(A[0 .. n-1])
//Sorts array A[0 .. n-1] by recursive mergesort
//Input: An array A[0 .. n-1] of orderable elements
//Output: Array A[0 .. n-1] sorted in non decreasing order
if n > 1
  copy A[0 .. $\lfloor n/2 \rfloor$ -1] to B[0 .. $\lfloor n/2 \rfloor$ -1]
  copy A[$\lfloor n/2 \rfloor$ .. n -1] to C[0..$\lceil n/2 \rceil$ -1]
  Mergesort(B[0 .. $\lfloor n/2 \rfloor$ - 1])
  Mergesort(C[0 ..$\lceil n/2 \rceil$ -1])
  Merge(B, C, A)

Merge Sort - Algorithm

```
ALGORITHM  Merge(B[0 .. p- 1],  C[0 .. q  -1], A[0 .. p  + q  -1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0 .. p -1] and C[0 .. q -1] both sorted
//Output: Sorted array A[0 .. p  + q  -1] of the elements of B and
   C
i←0;  j←0;  k←0
while i < p and j < q do
   if B[i] ≤ C[j]
            A[k] ←    B[i];  i ←    i  + 1
   else  A[k] ←C[j];  j←    j  + 1
   k←k+1
if i =  p
   copy C[j .. q-1] to A[k .. p + q - 1]
else
   copy B[i .. p -1] to A[k .. p + q -1]
```

Merge Sort - Example



Merge Sort - Algorithm

- Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is:

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ [for } n > 1], C(1) = 0$$

- The number of key comparisons performed during the merging stage in the worst case is:

$$C_{merge}(n) = n - 1$$

- Using the above equation:

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \text{ [for } n > 1], C_{worst}(1) = 0$$

- Applying Master Theorem to the above equation:

$$C_{worst}(n) \in \Theta(n \log n)$$

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS
UE19CS251

Shylaja S S
Department of Computer Science
& Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Merge Sort

Major Slides Content: Anany Levitin

Shylaja S S

Department of Computer Science & Engineering

Merge Sort - Idea

- Split array A[0..n-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A

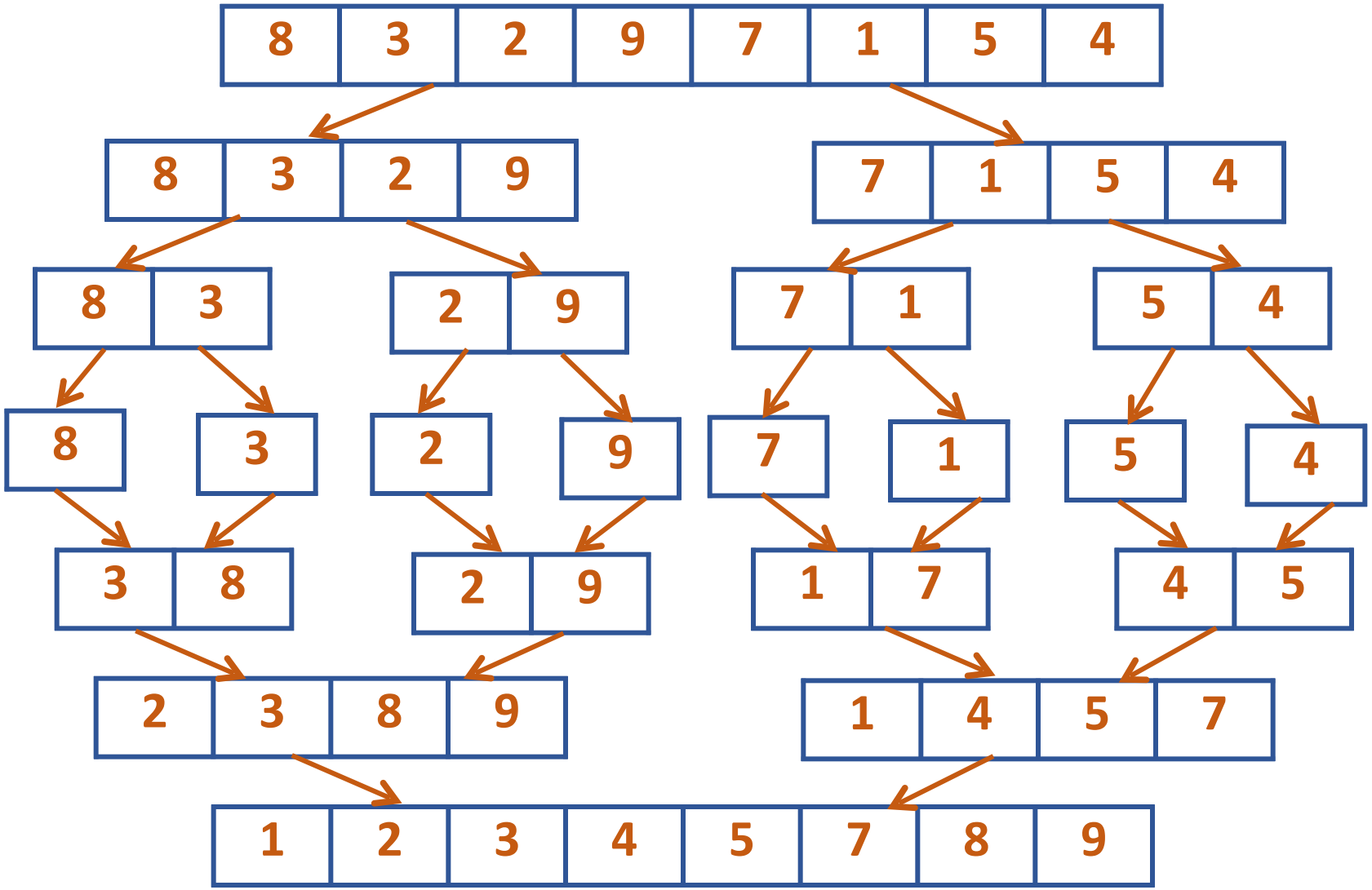
Merge Sort - Algorithm

ALGORITHM Mergesort(A[0 .. n-1])
//Sorts array A[0 .. n-1] by recursive mergesort
//Input: An array A[0 .. n-1] of orderable elements
//Output: Array A[0 .. n-1] sorted in non decreasing order
if n > 1
  copy A[0 .. $\lfloor n/2 \rfloor$ -1] to B[0 .. $\lfloor n/2 \rfloor$ -1]
  copy A[$\lfloor n/2 \rfloor$ .. n -1] to C[0..$\lceil n/2 \rceil$ -1]
  Mergesort(B[0 .. $\lfloor n/2 \rfloor$ - 1])
  Mergesort(C[0 .. $\lceil n/2 \rceil$ -1])
  Merge(B, C, A)

Merge Sort - Algorithm

```
ALGORITHM  Merge(B[0 .. p- 1],  C[0 .. q  -1], A[0 .. p  + q  -1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0 .. p -1] and C[0 .. q -1] both sorted
//Output: Sorted array A[0 .. p  + q  -1] of the elements of B and
   C
i←0;  j←0;  k←0
while i < p and j < q do
   if B[i] ≤ C[j]
            A[k] ←    B[i];  i ←    i  + 1
   else  A[k] ←C[j];  j←    j  + 1
   k←k+1
if i =  p
   copy C[j .. q-1] to A[k .. p + q - 1]
else
   copy B[i .. p -1] to A[k .. p + q -1]
```

Merge Sort - Example



Merge Sort - Algorithm

- Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is:

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ [for } n > 1], C(1) = 0$$

- The number of key comparisons performed during the merging stage in the worst case is:

$$C_{merge}(n) = n - 1$$

- Using the above equation:

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \text{ [for } n > 1], C_{worst}(1) = 0$$

- Applying Master Theorem to the above equation:

$$C_{worst}(n) \in \Theta(n \log n)$$

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS
UE19CS251

Shylaja S S
Department of Computer Science
& Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort

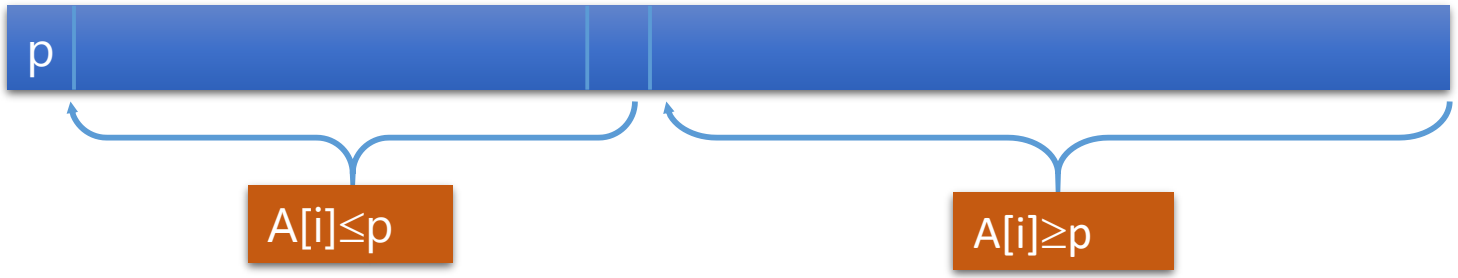
Major Slides Content: Anany Levitin

Shylaja S S

Department of Computer Science & Engineering

Quick Sort

- Select a pivot (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first s positions are smaller than or equal to the pivot and all the elements in the remaining n-s positions are larger than or equal to the pivot



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quick Sort - Algorithm

```
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 non decreasing order
if l  <  r
   s ←Partition(A[l .. r])          //s is a split position
   Quicksort(A[l  .. s - 1])
   Quicksort(A[s + 1 .. r])
```

Quick Sort - Algorithm

```
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 ← A[l]
i  ← l;  j ← r  + 1
repeat
   repeat i ← i + 1 until A[i] ≥ p
   repeat j ← j  - 1 until A[j] ≤ p
   swap(A[i],  A[j])
until i ≥ j
swap(A[i],  A[j])            //undo last swap when i ≥ j
swap(A[l],  A[j])
return  j
```

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort - Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Quick Sort - Analysis: Best Case

- The number of comparisons in the best case satisfies the recurrence:
- $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$

- According to Master Theorem

$$C_{best}(n) \in \Theta(n \log_2 n)$$

Quick Sort - Analysis: Worst Case

- The number of comparisons in the worst case satisfies the recurrence

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \theta(n^2)$$

Quick Sort - Analysis: Average Case

Let $C_{avg}(n)$ be the number of key comparisons made by Quick Sort on a randomly ordered array of size n

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

The solution for the above recurrence is:

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

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS
UE19CS251

Shylaja S S
Department of Computer Science
& Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Tree

Major Slides Content: Anany Levitin

Shylaja S S

Department of Computer Science & Engineering

Binary Tree

- A *binary tree* **T** is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees $\mathbf{T_L}$ and $\mathbf{T_R}$ called as the left and right subtree of the root

- The definition itself divides the Binary Tree into two smaller structures and hence many problems concerning the binary trees can be solved using the Divide – And – Conquer technique

- The binary tree is a Divide – And – Conquer ready structure ☺

Height of a Binary Tree

- Height of a Binary Tree: Length of the longest path from root to leaf

ALGORITHM Height(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if T = Ø return -1

else return max(Height(T_L), Height(T_R))+ 1

Height of a Binary Tree - Analysis

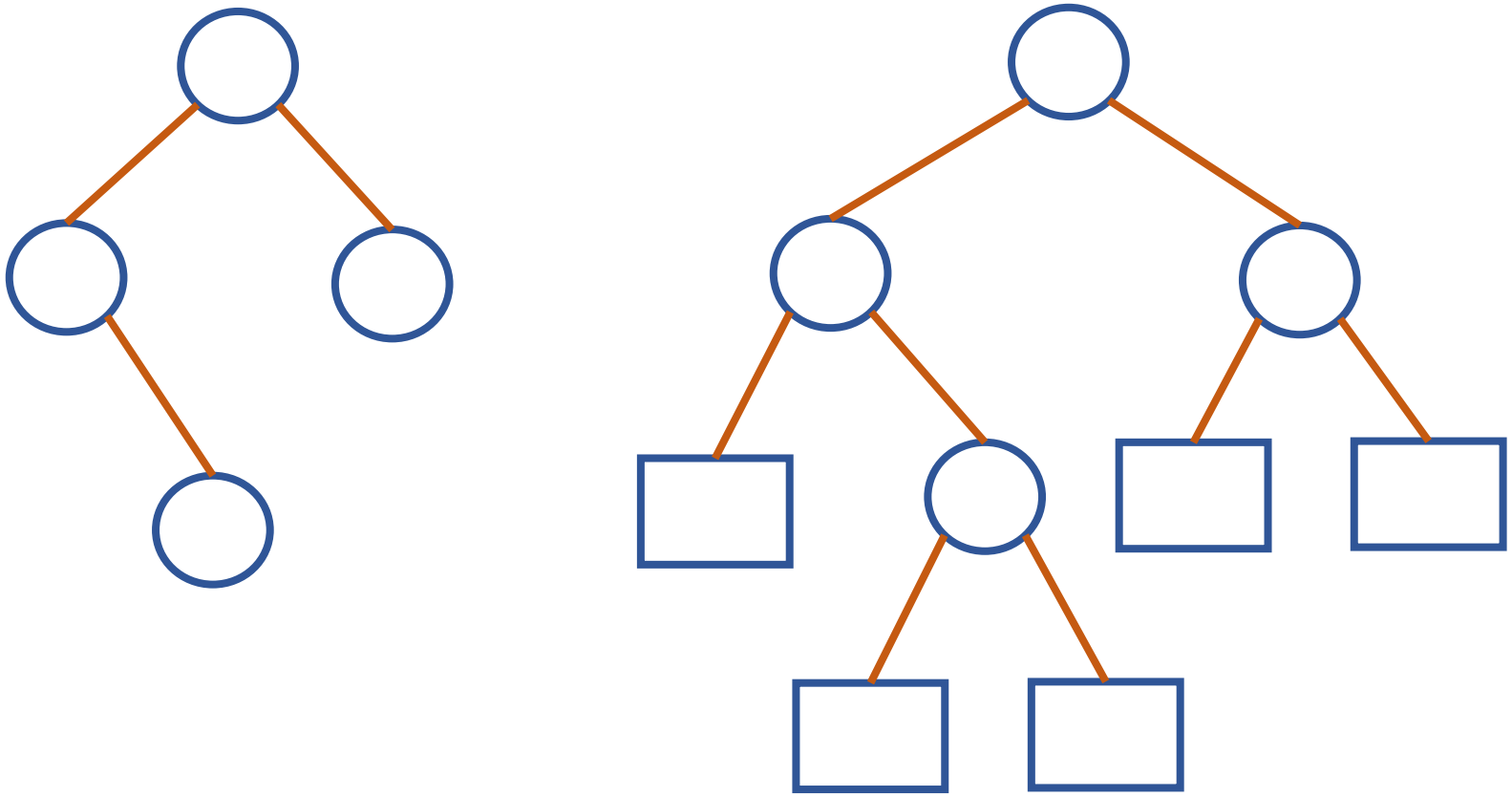
- The measure of input's size is the number of nodes in the given binary tree. Let us represent this number as n(T)
- Basic Operation: Addition
- The recurrence relation is setup as follows:

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1, \quad \text{for } n(T) > 0$$

$$A(0) = 0$$

Height of a Binary Tree - Analysis

- In the analysis of tree algorithms, the tree is extended by replacing empty subtrees by special nodes called external nodes



Height of a Binary Tree - Analysis

- x – Number of external nodes
- n – Number of internal nodes

$$x = n + 1$$

- The number of comparisons to check whether a tree is empty or not:

$$C(n) = n + x = 2n + 1$$

- The number of additions is:

$$A(n) = n$$

Binary Tree Traversals

- The three classic traversals for a binary tree are inorder, preorder and postorder traversals

- In the preorder traversal, the root is visited before the left and right subtrees are visited (in that order)
- In the inorder traversal, the root is visited after visiting its left subtree but before visiting the right subtree
- In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order)

Binary Tree Traversals

```
Algorithm Inorder(T)
if T ≠ ∅
    Inorder(Tleft)
    print(root of T)
    Inorder(Tright)
```

```
Algorithm Preorder(T)
if T ≠ ∅
    print(root of T)
    Preorder(T_left)
    Preorder(T_right)
```

```
Algorithm Postorder(T)
if T ≠ ∅
    Postorder(T_left)
    Postorder(T_right)
    print(root of T)
```

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS
UE19CS251

Shylaja S S
Department of Computer Science
& Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Multiplication of Large Integers and Strassen's Matrix Multiplication

Major Slides Content: Anany Levitin

Shylaja S S

Department of Computer Science & Engineering

Multiplication of large integers - Idea

- Let the two numbers being multiplied be a and b
- a and b are n–digit integers, where n is a positive even number
- Let the first half of a's digits be a_1 and second half be a_0
- Similarly, let the first half of b's digits be b_1 and second half be b_0
- In these notations, $a = a_1a_0$ implies $a = a_1*10^{n/2} + a_0$ and $b = b_1b_0$ implies $b = b_1*10^{n/2} + b_0$

Multiplication of large integers - Idea

$c = a*b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

$= c_2 10^n + c_1 10^{n/2} + c_0$

where

$c_2 = a_1 * b_1$ is the product of their first halves

$c_0 = a_0 * b_0$ is the product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0

Multiplication of large integers - Analysis

- $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$
- Solving it by backward substitutions for $n = 2^k$ yields:

$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2})$

$= \ldots = 3^i M(2^{k-i}) = \ldots = 3^kM(2^{k-k}) = 3^k$

- Since $k = \log_2 n$: $M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.585}$
- The number of additions is given by:

$$A(n) = 3A(n/2) + cn \text{ for } n > 1, A(1) = 1$$

$$A(n) \text{ belongs to } \Theta(n^{\log_2 3})$$

DESIGN AND ANALYSIS OF ALGORITHMS

Multiplication of large integers - Example

$2135 * 4014$

$= (21*10^2 + 35) * (40*10^2 + 14)$

$= (21*40)*10^4 + c1*10^2 + 35*14$

where $c1 = (21+35)*(40+14) - 21*40 - 35*14$, and

$21*40 = (2*10 + 1) * (4*10 + 0)$

$= (2*4)*10^2 + c2*10 + 1*0$

where $c2 = (2+1)*(4+0) - 2*4 - 1*0$, etc.,

Strassen's Matrix Multiplication

- This algorithm was published by V Strassen in 1969
- The principal insight of the algorithm lies in the discovery that we can find product of two 2 – by – 2 matrices A and B with seven multiplications as opposed to the eight required by the Brute – Force algorithm
- This is accomplished by the following formulae:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Strassen's Matrix Multiplication

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$m_2 = (a_{10} + a_{11}) * b_{00}$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Strassen's Matrix Multiplication – General Formula

For any two matrices A and B of size n – by – n, we can divide A, B and the product C as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

The sub – matrices can be treated as numbers to get the correct product

Strassen's Matrix Multiplication – Analysis

- If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two matrices n – by – n, we get the following recurrence relation for it:

$M(n) = 7M(n/2)$ for $n>1$, $M(1) = 1$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \ldots$$
$$= 7^i\, M(2^{k-i}) \ldots = 7^k\, M\,(2^{k-k}) = 7^k$$

Since $k = \log_2 n$,

$$\mathbf{M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}}$$

Strassen's Matrix Multiplication – Analysis

- The number of additions are given by the following recurrence:

  $A(n) = 7\ A(n/2) + 18(n/2)^2$ for $n>1$, $A(1) =0$

- According to Master's Theorem, A(n) belongs to $\Theta(n^{\log_2 7})$

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu