

Computer Algorithms

Unit-2

The Greedy method

The Greedy method

- Straight forward design technique
- It can be applied to wide variety of problems
- Most of such problems have n inputs and required to obtain a subset that satisfies some constraints
- Any subset that satisfies these constraints is called a **feasible solution**
- Find feasible solution that either maximizes or minimizes a given **objective function**.
- The feasible solution that do this, is called **optimal solution**

Subset Paradigm

- Algorithm works in stages, considering one input at a time
- At each stage decision is made regarding, whether a particular input is in an optimal solution
- This is done by considering the inputs in an order determined by some selection procedure.
- If inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution,
- then this input is not added to the partial solution, Otherwise it is added
- Selection procedure is based on objective function

Ordering Paradigm

- Do not call for the selection of an optimal subset
- Make decisions by considering the inputs in some order
- Each decision is made using an optimization criterion that can be computed using decisions already made

Knapsack Problem

- Given n items and a knapsack (or a bag) of capacity m .
- Associated with each item (i) is the weight (w_i) and the profit earned.
- If the fraction of item x_i is placed in knapsack
- then profit of $p_i x_i$ is earned. $0 \leq x_i \leq 1$

Knapsack Problem

- Objective is to obtain a filling of the knapsack that maximizes the total profit.
- Formally the problem can be stated as
- Maximize

$$\sum_{1 \leq i \leq n} p_i x_i$$

- Subject to

$$\sum_{1 \leq i \leq n} w_i x_i \leq m$$

- And $0 \leq x_i \leq 1 \quad 1 \leq i \leq n$

Greedy Approaches

- Sort items based on profit in decreasing order
place items till knapsack is full.
- Sort items based on weight in increasing
order place items till knapsack is full.
- Sort items based on profit/weight decreasing,
place items till knapsack is full.

Example

- Example 1

- $n=3$ $m=20$
- $(p_1, p_2, p_3) = (25, 24, 15)$
- $(w_1, w_2, w_3) = (18, 15, 10)$

- Example 2

- $n=7$ $m=15$
- $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (10, 5, 15, 7, 6, 18, 3)$
- $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$

■ $(0.1, 1/2)$ 20 31.5

Example

- Example 3

- $n=4$ $m=25$
- $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$
- $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$

- Example 4

- $n=6$ $c=20$
- $(p_1, p_2, p_3, p_4, p_5, p_6) = (12, 5, 15, 7, 6, 18)$
- $(w_1, w_2, w_3, w_4, w_5, w_6) = (2, 3, 5, 7, 1, 5)$

Example

- Example 5
- $n=4$ $m=30$
- $(p_1, p_2, p_3, p_4) = (27, 20, 24, 15)$
- $(w_1, w_2, w_3, w_4) = (15, 10, 18, 10)$

Job Sequencing with Deadlines

- Given a set of n jobs
 - Each job is associated with an integer deadline $d_i \geq 0$ and profit $p_i \geq 0$
 - For any job profit p_i will be earned iff the job is completed by its deadline
 - Each job takes unit time
 - Only one machine (server) is available.

Job Sequencing with Deadlines

- Feasible solution:

subset J of jobs such that

each job in J is completed within its

deadline

- Optimal solution:

Feasible solution with highest profit

Solution

- Greedy Method
 - Sorting jobs based on profit ↓
 - Sorting jobs based on deadlines ↑
- Implementation
 - Using array

Example

12	9	10	5	15	2	20
6	1	4	6	1	3	3
20	15	12	10	9	5	2
3	1	6	4	1	6	3

Array



1	2	3	4	5	6	7
---	---	---	---	---	---	---

Example

- $n=4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$
 $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Feasible Solution	Processing Sequence	Value
• (1,2)	2,1	110
• (1,3)	1,3 or 3,1	115
• (1,4)	4,1	127
• (2,3)	2,3	25
• (3,4)	4,3	42
• (1)	1	100
• (2)	2	10
• (3)	3	15
• (4)	4	27

Example

- **Example 1** $n=5$,
- $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$
- $(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

- **Example 2** $n=7$
- $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (3, 5, 20, 18, 1, 6, 30)$
- $(d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2)$

Example

- **Example 3** $n=5$,
- $(p_1, p_2, p_3, p_4, p_5) = (45, 15, 20, 7, 65)$
- $(d_1, d_2, d_3, d_4, d_5) = (1, 3, 2, 1, 2)$

- **Example 4** $n=7$
- $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (50, 15, 18, 16, 8, 25, 60)$
- $(d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2)$

Example

- **Example 5** $n=5$,
- $(p_1, p_2, p_3, p_4, p_5) = (20, 16, 11, 5, 25)$
- $(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 2, 1)$

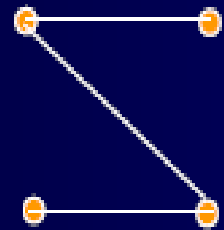
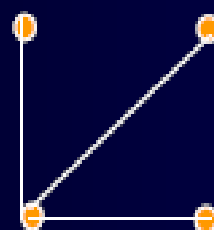
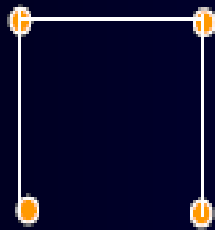
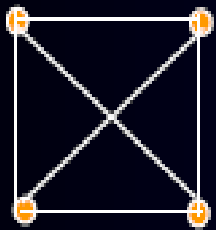
- **Example 6** $n=7$
- $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (45, 5, 20, 18, 6, 30, 70)$
- $(d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2)$

Job Sequencing with Deadlines

- Variants
 - Processing time is different
 - Multiple servers
 - Profit factors different if executed on different server

Minimum Cost Spanning Tree

- Spanning Tree-
- Let $G=(V,E)$ be undirected connected graph.
- A sub-graph $t=(V,E')$ of G is spanning tree
- iff t is tree.



- **Weighted graph:**

- Weights assigned to edges.

- **Minimum Spanning Tree**

- A spanning tree with minimum sum of weights

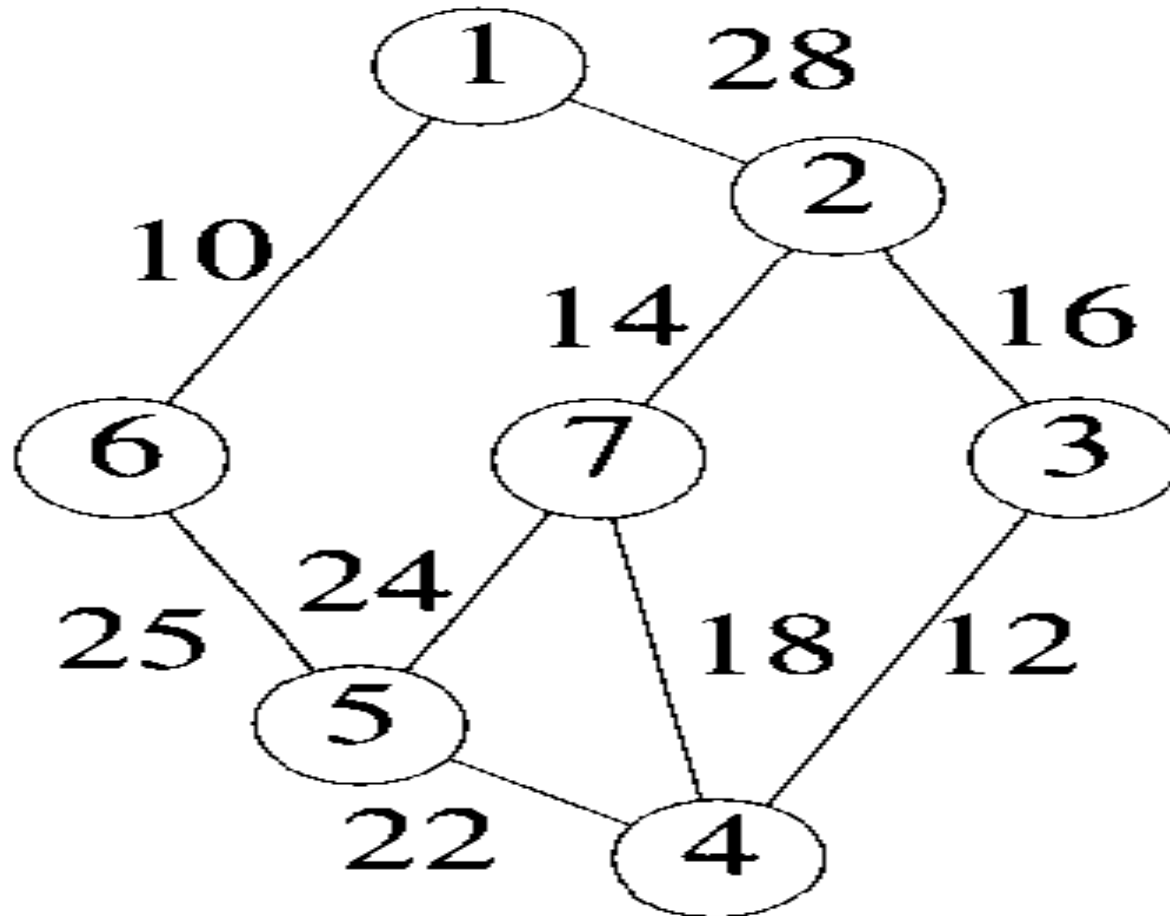
- It includes all vertices connected and sum of weight is minimum

Prim's Algorithm

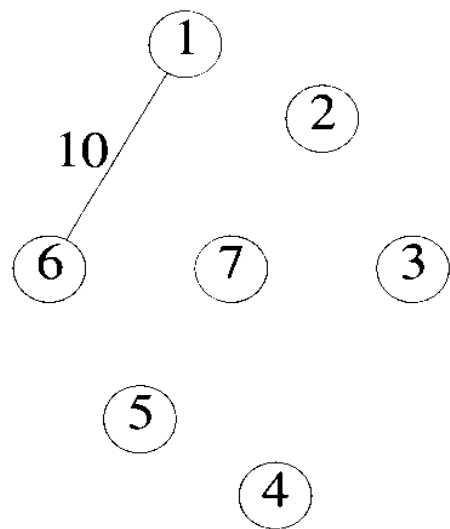
Steps

1. Select edge with smallest weight, include it in subset.
2. Select next adjacent vertex with minimum weight.
3. Include it in the subset if it does not form cycle.
4. Go to step 2 if all vertices are not included in subset.

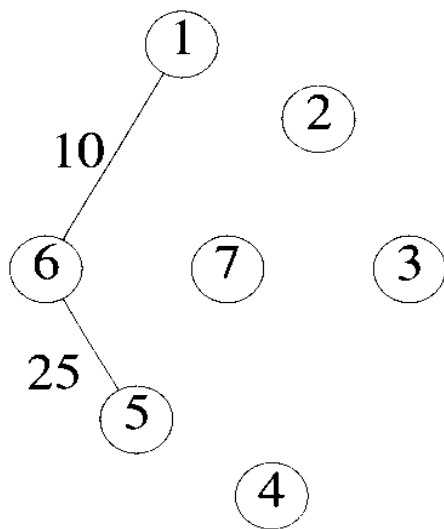
Prim's Algorithm



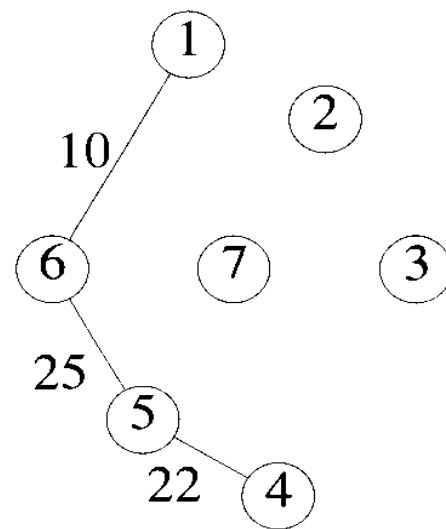
(a)



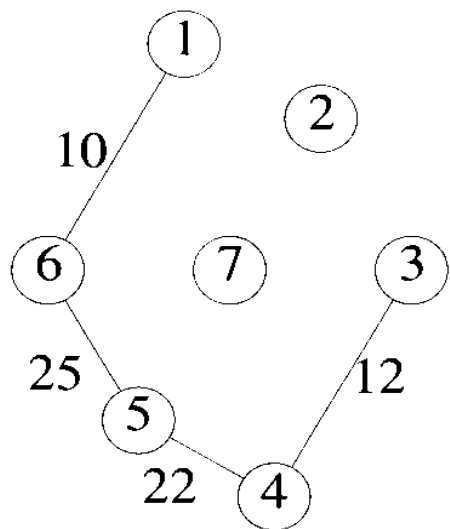
(a)



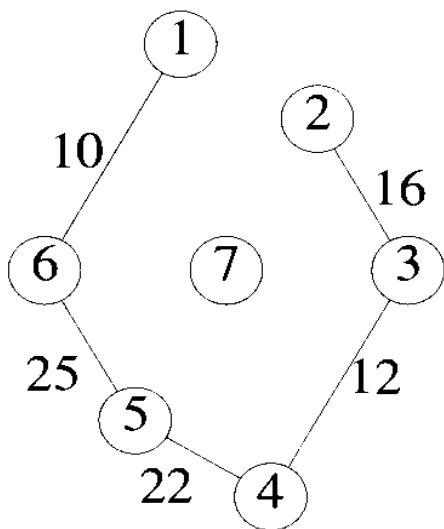
(b)



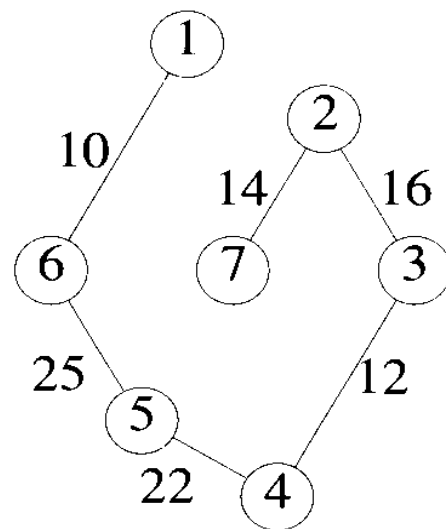
(c)



(d)



(e)



(f)

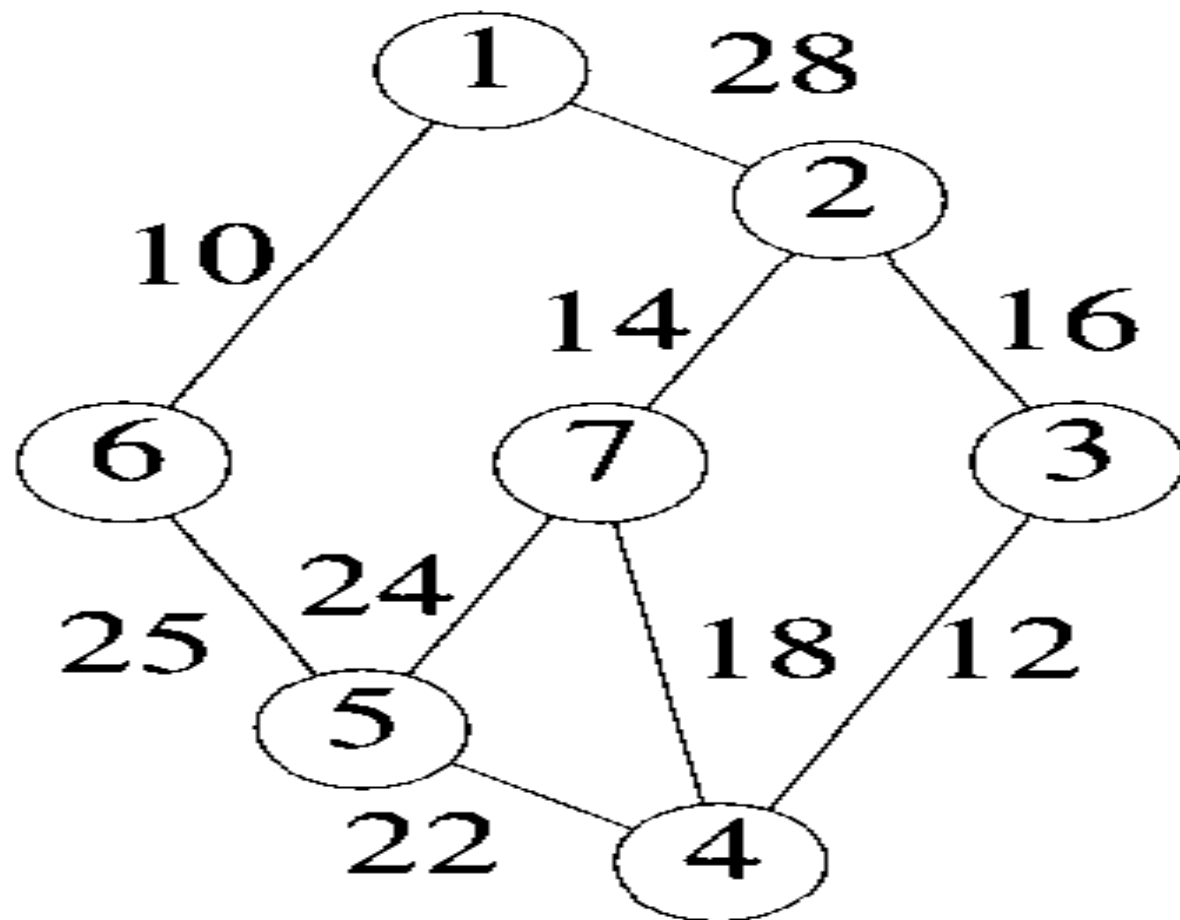
Prim's Algorithm

- After every step, graph we get is connected
- Complexity – $O(n^2)$

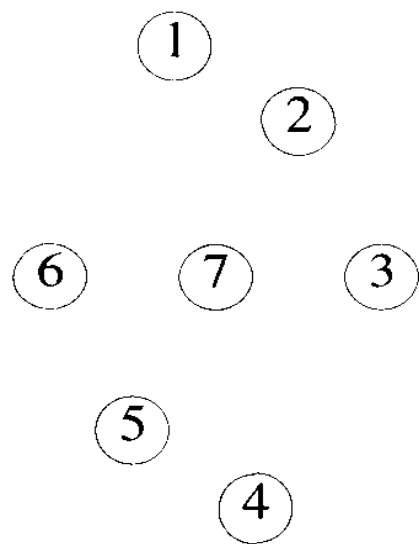
Kruskal's Algorithm

Steps

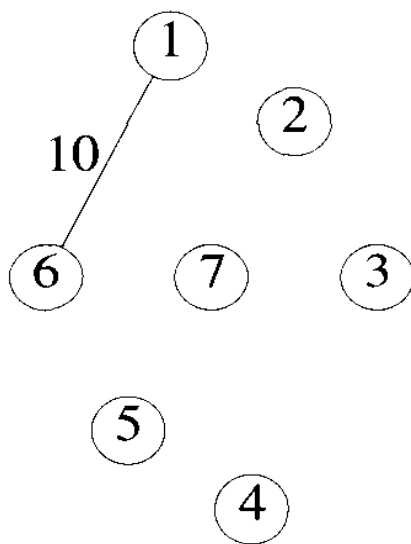
1. Sort the edge list on weight in non decreasing order.
2. Select next edge and include it in the subset if it does not form cycle.
3. Go to step 2 if all vertices are not included in subset.



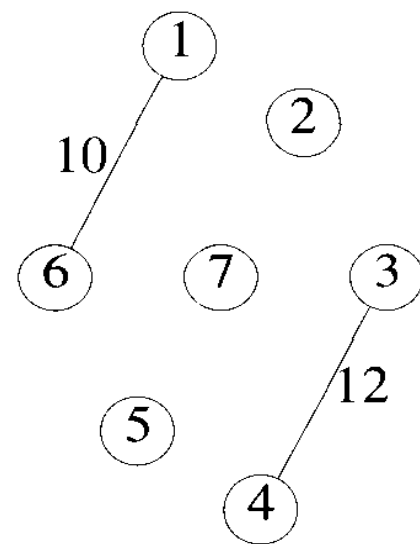
(a)



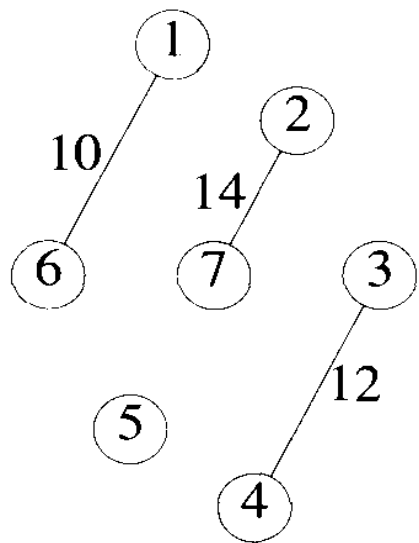
(a)



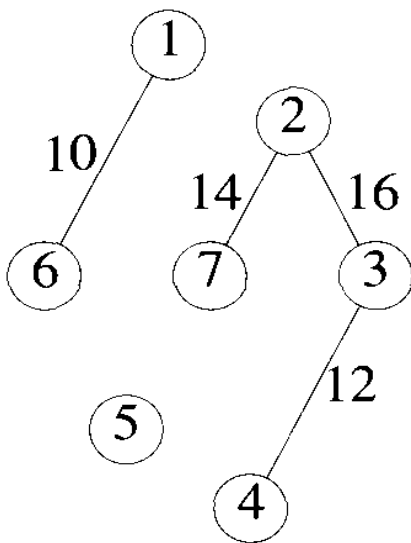
(b)



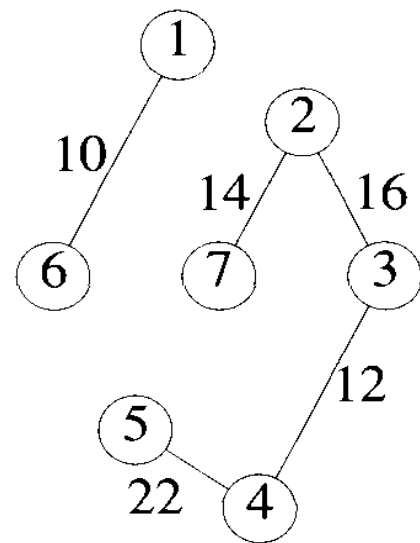
(c)



(d)



(e)



(f)

Kruskal's Algorithm

- After every step graph we get may not be connected graph
- Complexity – $O(E \log E)$

Application of MST

- Network bandwidth management-
 - Minimum bandwidth required to pass message from one node to another

Optimal Storage on Tapes

- There are n programs that are to be stored on a computer tape of length l
- Associated with each program i is a length l_i , $1 \leq i \leq n$
- All programs can be store on tape if and only if the sum of the length of the programs is at most l
- Whenever a program is to be retrieved from this tape, the tape is initially positioned at the front.

Optimal Storage on Tapes

- If the programs are stored in order
- $i_1, i_2, i_3, \dots, i_n$
- The time t_j needed to retrieve the program i_j is proportional to
$$\sum_{1 \leq k \leq j} i_k$$
- If all programs are retrieved equally often, then expected or mean retrieval time is (MRT) is
$$(1/n) \sum_{1 \leq j \leq n} t_j$$

Optimal Storage on Tapes

- In the optimal storage on tape problem
- We are required to find a permutation for the n programs so that
- When they are stored on the tape in this order
- MRT is minimized
- This problem fits into Ordering Paradigm

$$\text{Minimize } d(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_i i_k$$

$O(n \log n)$

Sort in non-decreasing order

Example

- $n=3$ $(l_1, l_2, l_3) = (5, 10, 3)$
- There are $n!$ possible orderings
- Ordering $d(l)$
- 1,2,3 $=38$
- 1,3,2 $=31$
- 2,1,3 $=43$
- 2,3,1 $=41$
- 3,1,2 $=29$
- 3,2,1 $=34$

Optimal Merge Pattern

- When two or more sorted files are to be merged together,
- the merge can be accomplished by repeatedly merging sorted files in pairs.
- Approach
 - Sort the files based on no of records
 - Form pairs
 - Merge pairs, Go to above step till complete
 - Wrong results.

Optimal Merge Pattern

- Revised Approach
 - Sort the files based on no of records
 - Merge first two files, Go to above step till complete
 - Wrong results.

Optimal Merge Pattern

- Re-revised Approach
 - 1 Sort the files based on no of records
 - 2 Select two smallest files,
 - 3 merge them,
 - 4 Place the resultant file at proper position in sorted list
 - 5 repeat step 2 to 4 till all files are merged.

- Example.

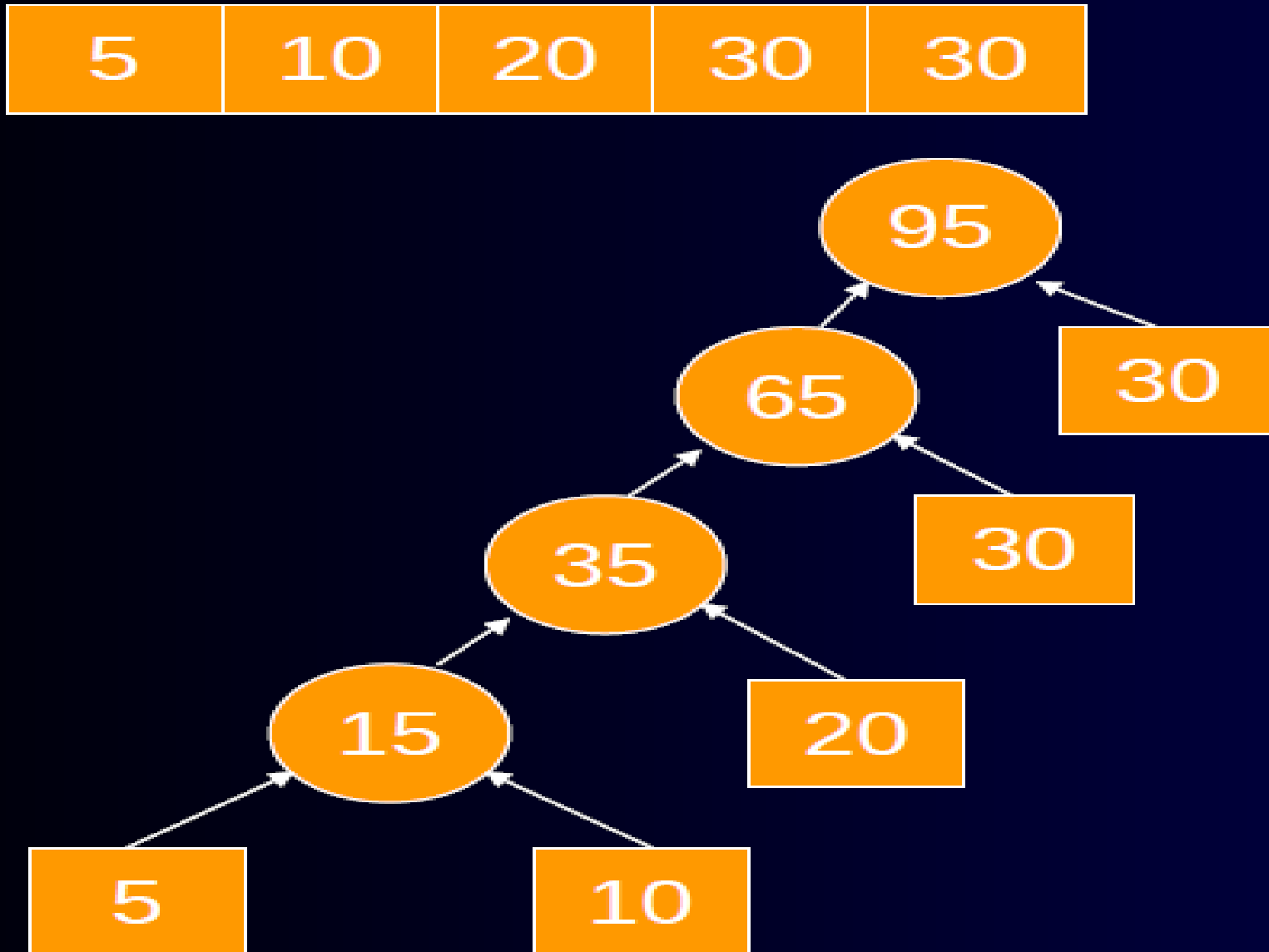
1	2	3	4	5
10	5	9	6	12

- Find an optimal binary merge pattern for ten files whose lengths are

Optimal Merge Pattern

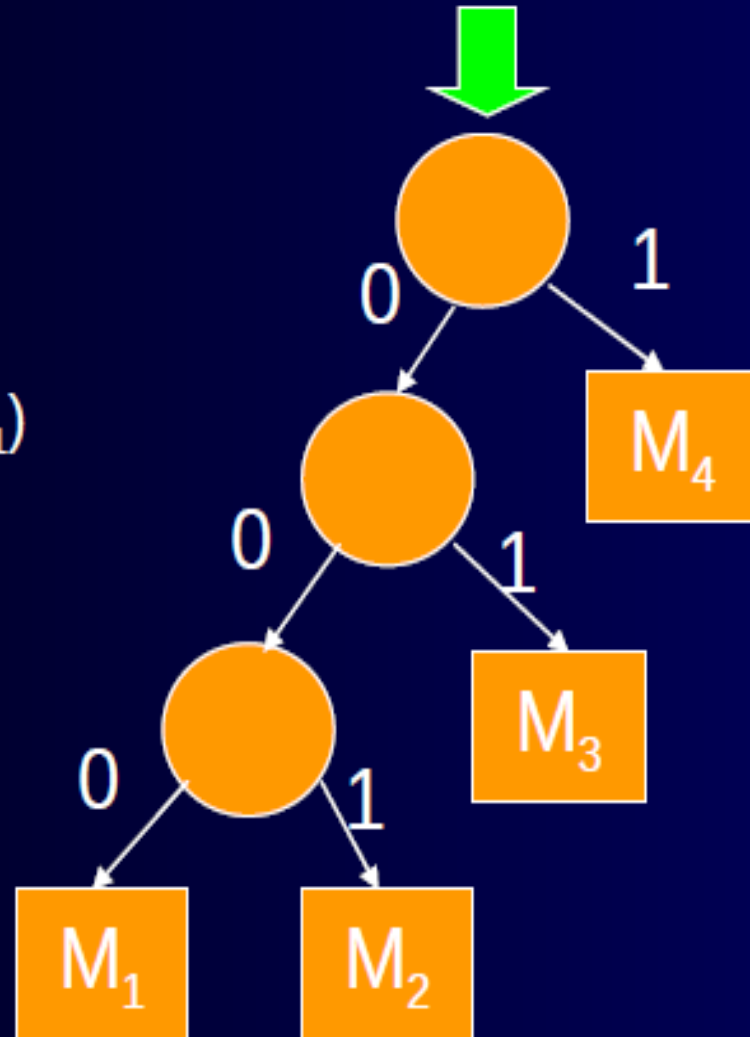
- Find an optimal binary merge pattern for ten files whose lengths are
- 28,32,12,5,84,53,91,35,3, and 11.

Optimal Merge Pattern



Optimal Merge Pattern

- Variants
 - N-way merge
 - Multiple servers
- Applications
 - Huffman code
 - $p(M_4) \geq p(M_3) \geq p(M_2) \geq p(M_1)$
 - $M_3 \sim 01$



Huffman Code

- Obtain a set of optimal Huffman codes for the messages (M_1, \dots, M_7) with relative frequencies
- $(q_1, \dots, q_7) = (4, 5, 7, 8, 10, 12, 20)$.
- Draw the decode tree for this set of codes.

SINGLE-SOURCE SHORTEST PATHS

- Graphs can be used to represent the highway structure of a state or country
- vertices representing cities and edges representing roads
- The edges can then be assigned weights
- which may be either the distance between two cities connected by the edge or the average time to drive

SINGLE-SOURCE SHORTEST PATHS

- A motorist wishing to drive from city A to B
- would be interested in answers to the following questions:
- Is there a path from A to B
- If there is more than one path from A to B, which is the shortest path ?

SINGLE-SOURCE SHORTEST PATHS

- The length of a path is defined to be the sum of the weights of the edges on that path.
- The starting vertex of the path is referred to as the **source**, and the last vertex the **destination**.
- The graphs are digraphs to allow for one-way streets.
- In the problem we consider, we are given a directed graph $G = (V, E)$,
- a weighting function cost for the edges of G ,
- and a source vertex v_0 .

SINGLE-SOURCE SHORTEST PATHS

- The problem is to determine the shortest paths from v_0 to all the remaining vertices of G .
- It is assumed that all the weights are positive.
- The shortest path between v_0 and some other node v is an ordering among a subset of the edges.
- Hence this problem fits the ordering paradigm.

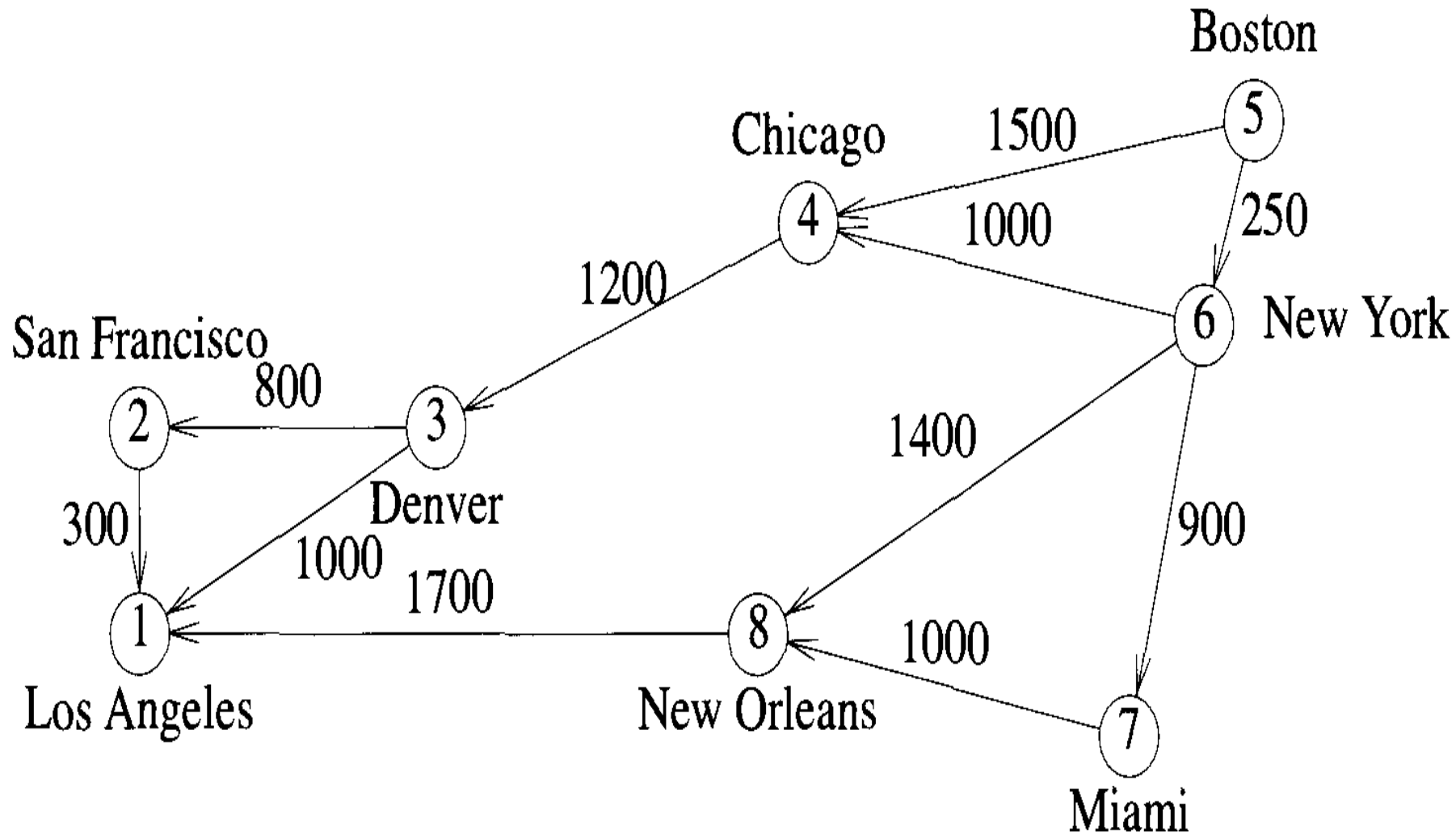
Single Source Shortest Path

- Motorist wish to visit all other cities from city A
- First shortest path to nearest city is generated, then shortest path to the second nearest city and so on...
- Form subset S containing all the cities visited.

Single Source Shortest Path: Observations

- If next shortest path is to u , then the path begins at v_0 and ends at u and goes through only those vertices in S .
- Distance of next path generated must have minimum distance amongst all vertices not in S .
- Vertex u becomes member of S .

SINGLE-SOURCE SHORTEST PATHS



(a) Digraph

SINGLE-SOURCE SHORTEST PATHS

Iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	----	$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
1	{5}	6	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
2	{5,6}	7	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	{5,6,7}	4	$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	$+\infty$	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

SINGLE-SOURCE SHORTEST PATHS

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Thank You

Computer Algorithms

Unit-1

Chapter- 1

Introduction

History of Algorithm

- The word algorithm comes from the name of a Persian author ,Abu Ja'far Mohammed bin Musa al Khowarazimi(825 A.D.).
- This word has taken on a special significance in computer science,
- “algorithm” has come to refer to a method that can be used by a computer for the solution of a problem.
- That is what makes algorithm different from words such as process, technique, or method.

Definition of Algorithm

- An algorithm is a finite set of instructions that, if followed ,accomplishes a particular task.
- All algorithms must satisfy following criteria.
 1. **Input** – Zero or More
 2. **Output** – At lease one
 3. **Definiteness** – Each instruction clear and unambiguous
 4. **Finiteness** – Terminate after finite number of steps
 5. **Effectiveness** – Every instruction must be basic and feasible

Areas of study

1. How to device algorithms

Divide and Conquer, Greedy Method, Dynamic Programming

2. How to validate algorithms

Show that it computes correct answer for all possible legal inputs

3. How to analyze algorithms

Time and space requirement

4. How to test a program

Debugging and profiling

Debugging

- Process of executing programs on sample data sets
- to determine whether faulty results occur, and if so correct them
- Debugging can only point to the presence of errors, but not to their absence

Profiling (Performance Measurement)

- Process of executing a correct program on data sets
- and measuring the time and space it takes to compute the results
- It helps for improvement

- We can describe an algorithm in many ways .
- We can use natural language like English,
if we choose this operation , the resulting
operations instructions are definite.
- Graphical representations called **Flow Charts**
are another possibility,
but they work well only if the algorithm is
small and simple.

Difference between Pseudo Code and Flowchart

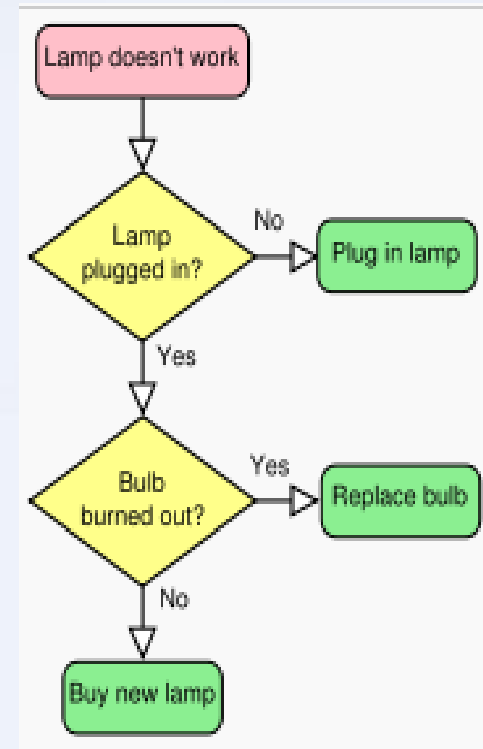
Algorithm Lamp:

Begin

```
if Lamp_plugged_in then
  if Bulb_burned_out then
    Replace bulb
  else buy new lamp.
else plug in lamp.
```

End

Pseudo Code



Flow Chart

Algorithm Specification

- Pseudo code Conventions

```
while <condition> do  
{  
    <statement 1>  
    ⋮  
    <statement n>  
}
```

```
for variable := value1 to value2 step step do  
{  
    <statement 1>  
    ⋮  
    <statement n>  
}
```

A **repeat-until** statement is constructed as follows:

```
repeat  
    <statement 1>  
    ⋮  
    <statement n>  
until <condition>
```

```
if <condition> then <statement>  
if <condition> then <statement 1> else <statement 2>
```

Algorithm Specification

- Pseudo code Conventions

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

Recursive Algorithms

- Fibonacci Series
- Factorial of a Number
- GCD
- Towers of Hanoi

Recursive Vs Iterative

RECURSION	ITERATIONS
Recursive function – is a function that is partially defined by itself	Iterative Instructions –are loop based repetitions of a process
Recursion is usually slower than iteration due to overhead of maintaining stack	Iteration does not use stack so it's faster than recursion
Recursion uses more memory than iteration	Iteration consume less memory
Infinite recursion can crash the system	infinite looping uses CPU cycles repeatedly
Recursion makes code smaller	Iteration makes code longer

Recursive Algorithms

- Fibonacci Series

```
int fibonacci(int n)
{
    if (n <= 1)
        return n;
    else
        return (fibonacci(n-1) + fibonacci(n-2));
}
```


Recursive Algorithms

- Factorial of a Number

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return (n * factorial(n-1));
}
```

Recursive Algorithms

- **GCD - Euclid's Algorithm**

```
int gcd(int m, int n)
{
    if ((m % n) == 0)
        return n;
    else return gcd(n, m % n);
}
```

$\text{gcd}(468, 24)$ $\text{gcd}(24, 12) \Rightarrow 12$

$\text{gcd}(135, 19)$ $\text{gcd}(19, 2)$ $\text{gcd}(2, 1) \Rightarrow 1$

Recursive Algorithms

- **GCD - Dijkstra's Algorithm**

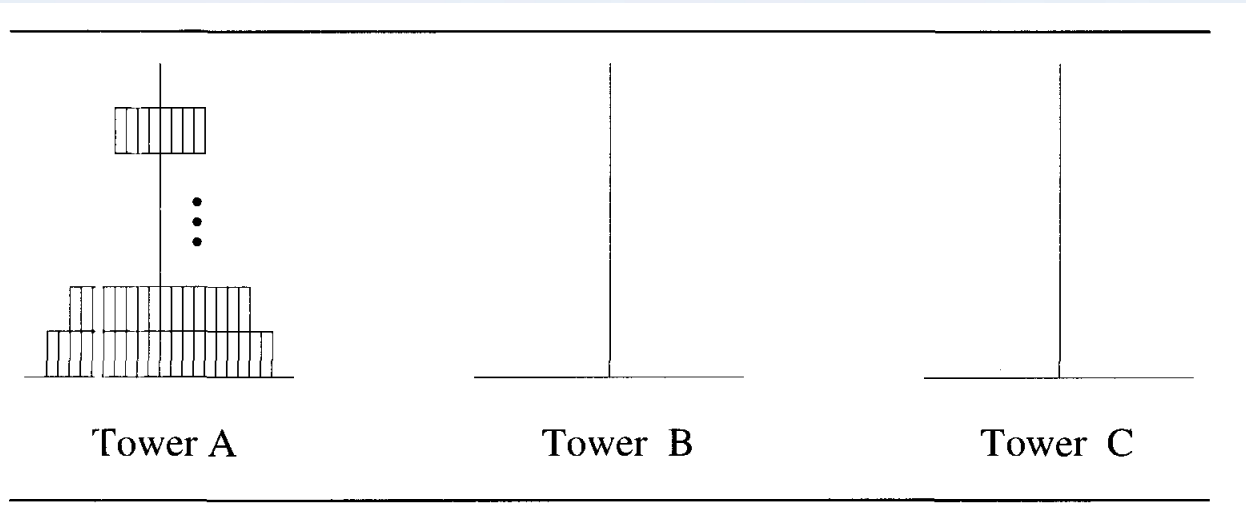
```
int gcd(int m, int n)
{
    if(m == n)
        return m;
    else if (m > n)
        return gcd(m-n, n);
    else
        return gcd(m, n-m);
}
```

Recursive Algorithms

- **GCD - Dijkstra's Algorithm**
- $\text{gcd}(468, 24)$
- $\text{gcd}(444, 24)$
- $\text{gcd}(420, 24) \dots$
- $\text{gcd}(36, 24)$
- $\text{gcd}(12, 24)$ (Now n is bigger)
- $\text{gcd}(12, 12)$ (Same) $\Rightarrow 12$

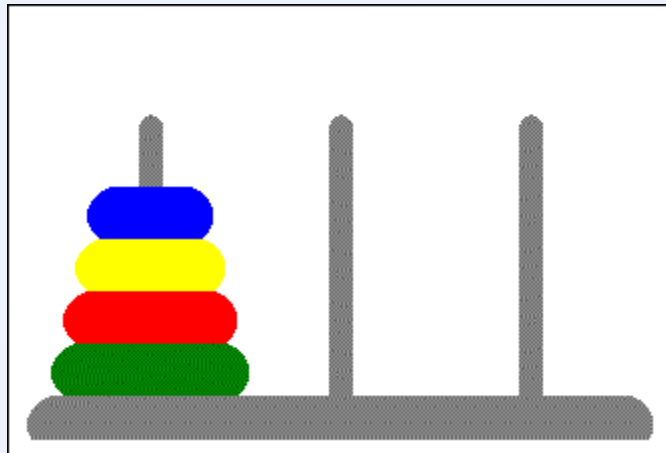
Recursive Algorithms

- Towers of Hanoi



Recursive Algorithms

- Towers of Hanoi



Recursive Algorithms

- Towers of Hanoi

```
procedure Hanoi(n: integer; source, dest, by: char);
Begin
  if (n=1)
    then print('Move the plate from ', source, ' to ', dest)
  else
    begin
      Hanoi(n-1, source, by, dest);
      print('Move the plate from ', source, ' to ', dest);
      Hanoi(n-1, by, dest, source);
    end;
End;
```

Recursive Algorithms

```
#include <stdio.h>
void towers(int, char, char, char);
int main()
{ int num;
  printf("Enter the number of disks : ");
  scanf("%d", &num);
  printf("The sequence of moves involved in the
    Tower of Hanoi are :\n");
  towers(num, 'A', 'C', 'B');
  return 0;
}
```


Recursive Algorithms

```
void towers(int num, char frompeg, char topeg,
            char auxpeg)
{ if (num == 1)
  { printf("\n Move disk 1 from peg %c to peg %c",
    frompeg, topeg);
    return;
  }
  towers(num - 1, frompeg, auxpeg, topeg);
  printf("\n Move disk %d from peg %c to peg
    %c", num, frompeg, topeg);
  towers(num - 1, auxpeg, topeg, frompeg);
}
```

Recursive Algorithms

Enter the number of disks : 3

The sequence of moves involved in the Tower of Hanoi are :

Move disk 1 from peg A to peg C

Move disk 2 from peg A to peg B

Move disk 1 from peg C to peg B

Move disk 3 from peg A to peg C

Move disk 1 from peg B to peg A

Move disk 2 from peg B to peg C

Move disk 1 from peg A to peg C

Performance Analysis

- Many Criteria upon which we can judge algorithm –
 - Does it do, what we want it to do?
 - Does it work correctly according to original specification of the task
 - Is there documentation that describes how to use it and how it works?
 - Modularity?
 - Is code reachable?

Space / Time Complexity

- The Space Complexity of an algorithm is the amount of memory it needs to run to completion
- The Time Complexity of an algorithm is the amount of computer time it needs to run to completion

- Performance Analysis can be divided into two phases

1. Priori Estimates

- Performance Analysis

2. Posteriori Testing

- Performance Measurement

Space Complexity

- Space needed by algorithm is sum of
 1. **A fixed part** – which is independent of characteristics of the inputs and outputs
e.g. Instruction Space, Space for variables
 2. **A variable part** – which consists of space needed by component variables whose size is dependent on the particular problem instance being solved
e.g. Space needed by reference variable, recursion stack space

$$S(P) = c + S_p$$

Time Complexity

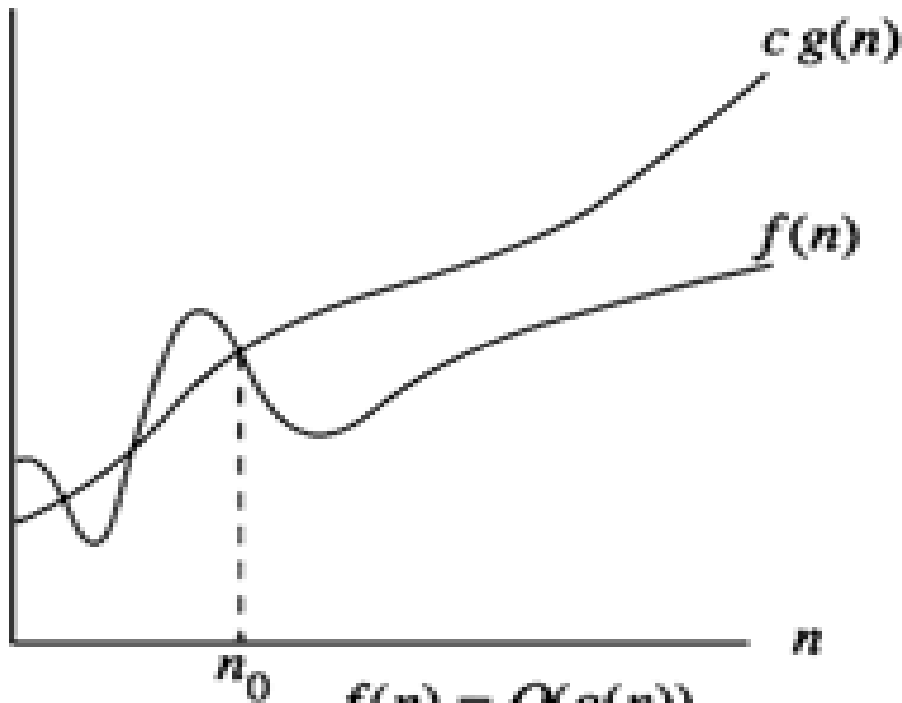
- The time $T(P)$ taken by a program P is the sum of the compile time and the run time (execution time)
- Compile time does not depend on instance characteristics
- Program compiled Once can be run several times
- So only run time is concerned with Time Complexity
- t_p
Key Operations like Comparison are considered

Asymptotic Notation (O , Ω , Θ)

- O (Big oh) –

The function $f(n) = O(g(n))$

iff there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n , $n > n_0$



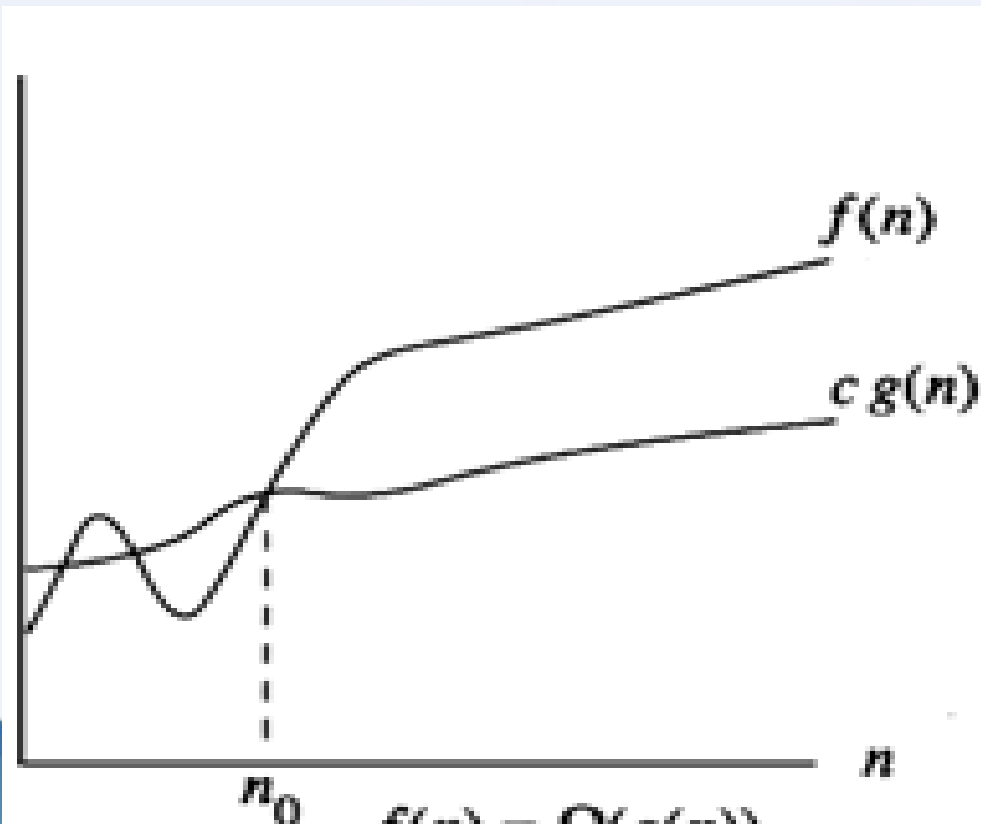
It specifies Upper Bound
Worst Case complexity.

Asymptotic Notation (O , Ω , Θ)

- Ω (Omega) –

The function $f(n) = \Omega(g(n))$

iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n , $n > n_0$



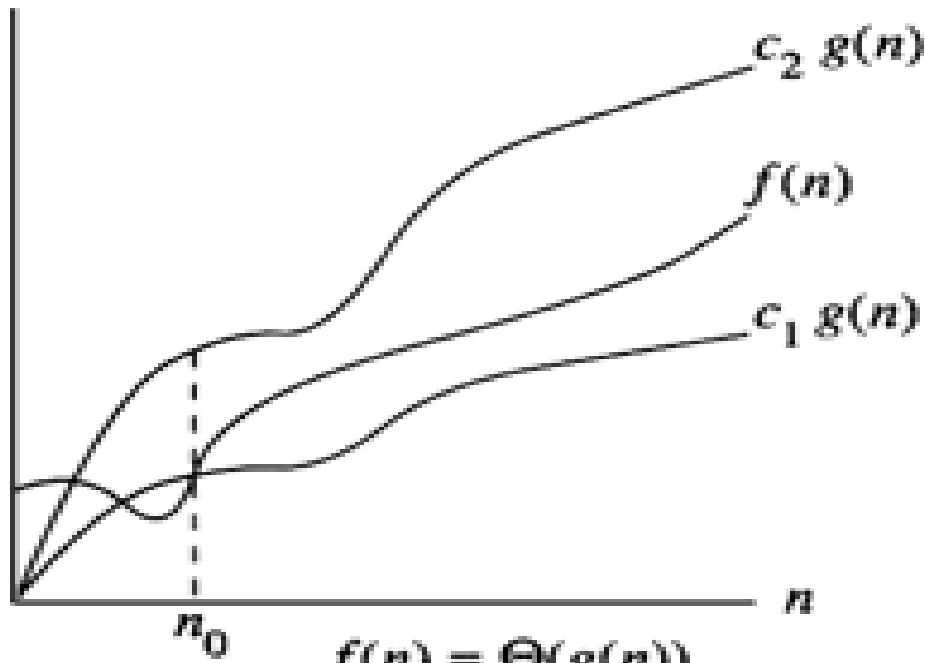
It specifies Lower Bound
Best Case complexity.

Asymptotic Notation (O , Ω , Θ)

- Θ (Theta) –

The function $f(n) = \Theta(g(n))$

iff there exist positive constants c_1 , c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all n , $n > n_0$



It specifies average case complexity.

Difference between best case and worst case complexity is very less.

- **o (Little Oh)**
 - The function $f(n)=o(g(n))$ iff
 - $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$
 - $g(x)$ grows much faster than $f(x)$
 - the growth of $f(x)$ is nothing compared to that of $g(x)$.
- **ω (Little Omega)**
 - The function $f(n)=\omega(g(n))$ iff
 - $\lim_{n \rightarrow \infty} g(n)/f(n) = 0$

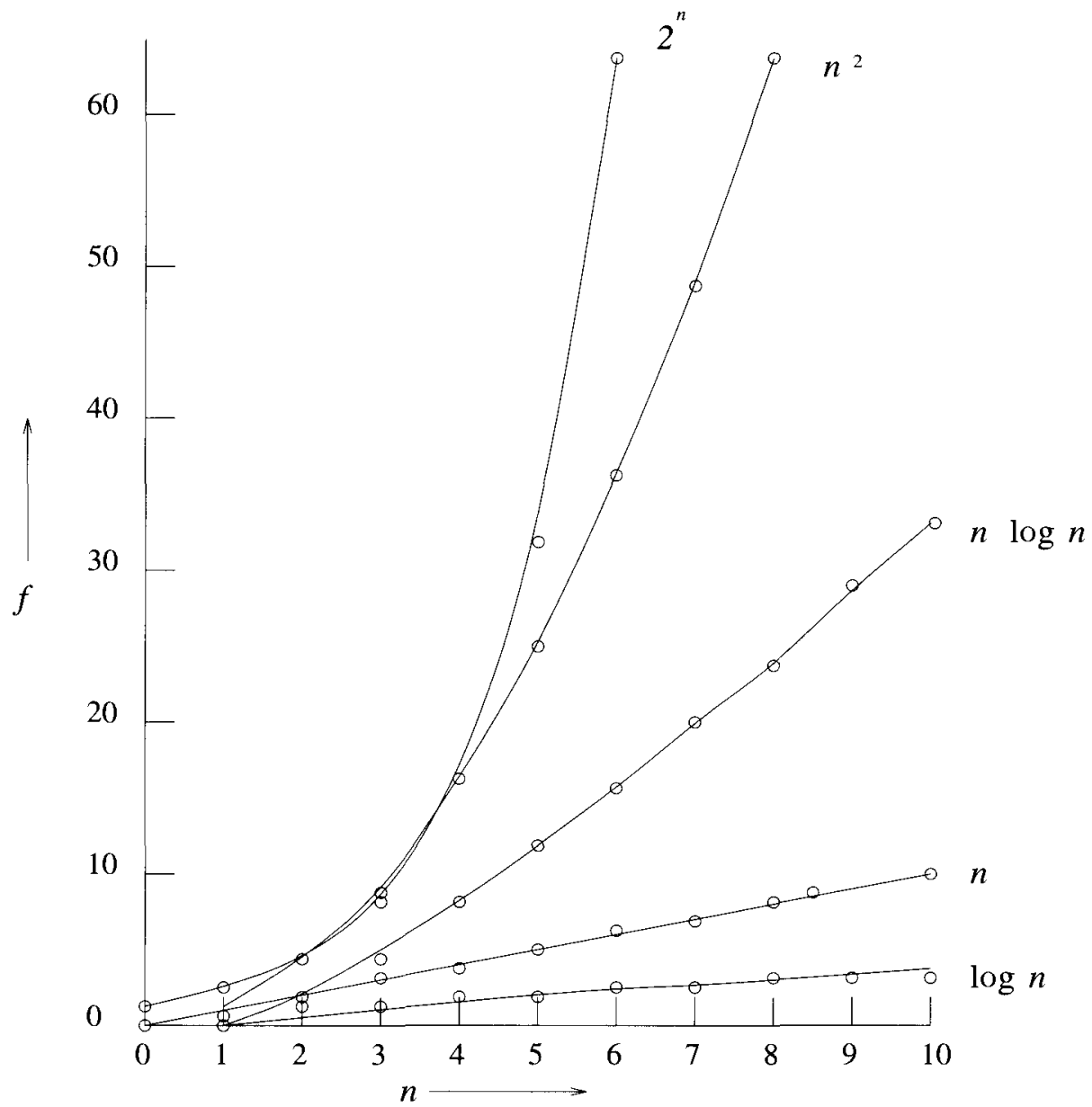
Practical Complexities

Time complexity		Example
$O(1)$	<i>constant</i>	Adding to the front of a linked list
$O(\log N)$	<i>log</i>	Finding an entry in a sorted array
$O(N)$	<i>linear</i>	Finding an entry in an unsorted array
$O(N \log N)$	<i>n-log-n</i>	Sorting n items by 'divide-and-conquer'
$O(N^2)$	<i>quadratic</i>	Shortest path between two nodes in a graph
$O(N^3)$	<i>cubic</i>	Simultaneous linear equations
$O(2^N)$	<i>exponential</i>	The Towers of Hanoi problem

Practical Complexities

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10,000^2$	minutes
Polynomial	$O(n^k)$	$10,000^k$	hours
Exponential	$O(c^n)$	$2^{10,000}$	intractable
Factorial	$O(n!)$	$10,000!$	intractable

Measures of Efficiency for $n = 10,000$



Function values

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

RANDOMIZED ALGORITHMS

- Probability Theory
- The probability of an event E is denoted to be
- $|E| / |S|$, where S is the sample space.
- Mutual exclusion
- Two events E_1 and E_2 are said to be mutually exclusive
- if they do not have any common sample points,
- that is $E_1 \cap E_2 = \Phi$

RANDOMIZED ALGORITHMS

- A randomized algorithm is one that makes use of a randomizer
- such as a random number generator.
- Some of the decisions made in the algorithm depend on the output of the randomizer.

RANDOMIZED ALGORITHMS

- Since the output of any randomizer might differ in an **unpredictable way** from run to run,
- The output of a randomized algorithm could also **differ** from run to run for the **same input**.
- The **execution time** of a randomized algorithm could also vary from run to run for the same input.

RANDOMIZED ALGORITHMS

- Randomized algorithms can be categorized into two classes:
- Las Vegas algorithms
- always produce the same (correct) output for the same input.
- The execution time of a Las Vegas algorithm depends on the **output of the randomizer**.

RANDOMIZED ALGORITHMS

- Monte Carlo algorithms.
- **outputs might differ** from run to run for the same Input
- Consider any problem for which there are only two possible answers, say, **yes and no**.
- If a **Monte Carlo algorithm** is employed to solve such a problem,
- then the algorithm might give incorrect answers depending on the output of the randomizer.
- We require that the probability of an incorrect answer from a Monte Carlo algorithm be **low**.

RANDOMIZED ALGORITHMS

- Typically, for a fixed input, a **Monte Carlo algorithm** does not display much variation in execution time between runs,
- whereas in the case of a **Las Vegas algorithm** this variation is significant.

Las Vegas algorithms - Example

- Randomized Quick Sort
- Identifying the Repeated Element
- Consider an array $a[]$ of n numbers that has $n/2$ distinct elements and $n/2$ copies of another element.
- The problem is to identify the repeated element.
- Solutions –
 - $n/2+1$ unique elements so time is $n/2+2$
 - Sorting – $O(n \log n)$

Las Vegas algorithms - Example

- Randomized Algorithm
- Generate 2 indices between 1 and n randomly
- Compare the values indicated by these indices
- If values are same, repeated element is found
- Complexity – $O(\log n)$

Monte Carlo algorithms - Example

- Primality Testing
- Given a number n , find if it is prime or not
- Prime number is divisible by 1 and that number
- Test if it is divisible by any number from 1 to root of that number
- If not divisible – it is prime number
- Time required - root of (n)
- Randomized algorithm can be used for primality testing – $O(\log n)$

PRIMALITY TESTING

- Deterministic algorithm for testing whether an integer **n is a prime** is simply a test whether any of the integers between 2 and root of n divides n.
- If number of digit is more than **40** then complexity is **exponential** to the input size
- It will take millions of years.
- Recently, M. Agrawal, N. Kayal, and N. Saxena have designed a polynomial algorithm for primality.

PRIMALITY TESTING

- **Fermat's little theorem** : Let n be a prime number and let a : any number that is not divisible by n ; then,
- Based on the primality test given,

$$a^{n-1} \equiv 1 \pmod{n}.$$

PRIMALITY TESTING

- choose a base **a** at random from $\{2, \dots, n-1\}$
- and return true (that is, the number is prime) if and only if $a^{n-1} = 1 \pmod{n}$.
- Most composite numbers n fail the Fermat test for many integers **a between 2 and $n-1$.**
- Thus, for such numbers the Monte Carlo algorithm has a high probability of being correct.

Computer Algorithm

Unit-1

Chapter 2

Divide and Conquer

Divide and Conquer

- Function to compute on n inputs
- Split the inputs into k distinct subsets $1 < k \leq n$
- Yielding k sub problems
- These sub problems are solved
- Then combine sub solutions into a solution of the whole

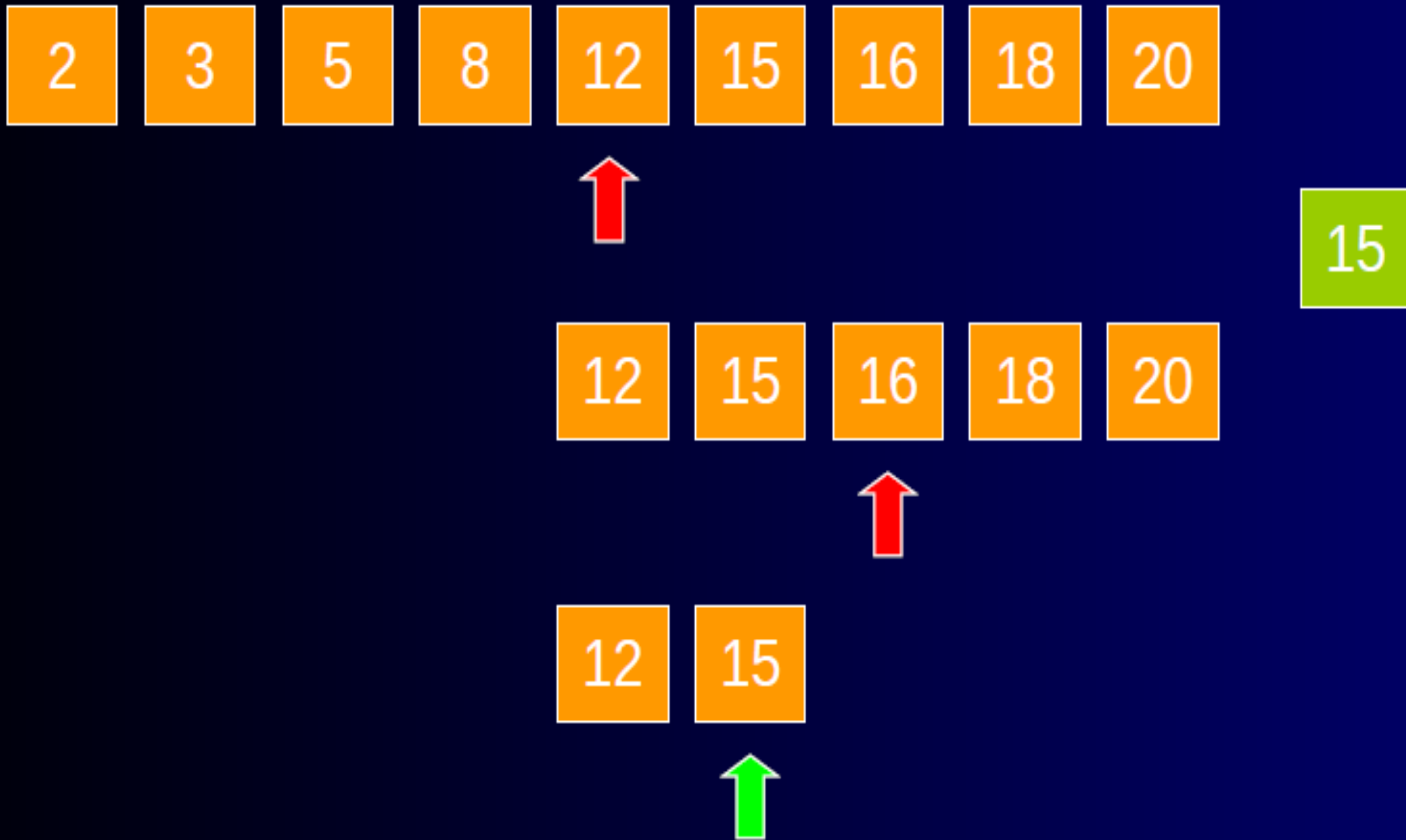
Binary Search

- Statement: Let $a_i, 1 \leq i \leq n$, be the list of elements that are stored in non-decreasing order, determine if the element x is present in the list.
- Search solutions: Approaches
 - Linear
 - D&C: Binary
- Binary search

Divide the list in two equal parts, look for middle element
If $x \leq \text{middle}$, discard second half else discard first half
Repeat till

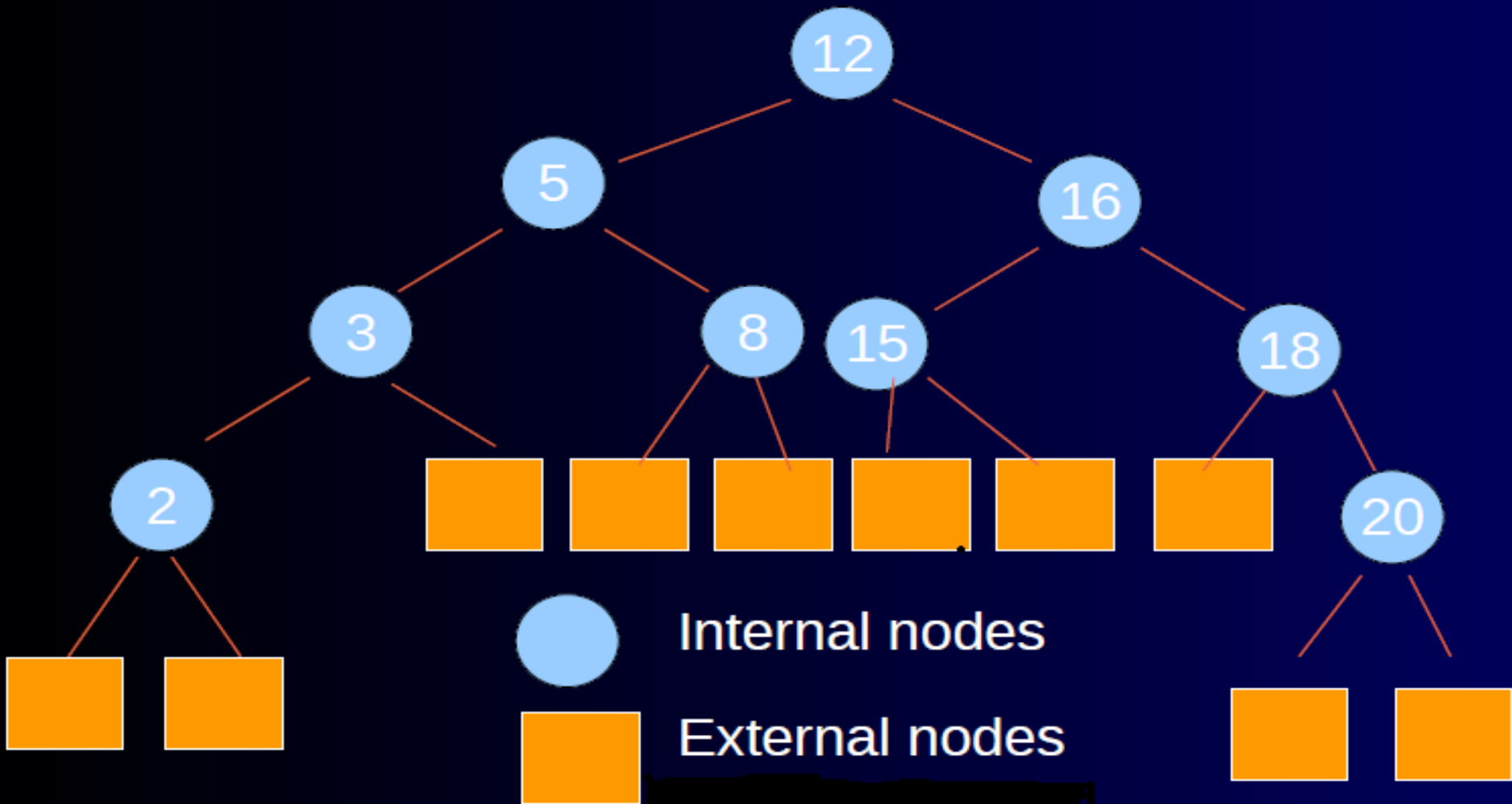
 - element is found
 - elements not present in the list

Binary Search : Demonstration



Binary Search : Decision Tree

$n=9$




```

1  Algorithm BinSrch( $a, i, l, x$ )
2  // Given an array  $a[i : l]$  of elements in nondecreasing
3  // order,  $1 \leq i \leq l$ , determine whether  $x$  is present, and
4  // if so, return  $j$  such that  $x = a[j]$ ; else return 0.
5  {
6      if ( $l = i$ ) then // If Small( $P$ )
7      {
8          if ( $x = a[i]$ ) then return  $i$ ;
9          else return 0;
10     }
11     else
12     { // Reduce  $P$  into a smaller subproblem.
13          $mid := \lfloor (i + l) / 2 \rfloor$ ;
14         if ( $x = a[mid]$ ) then return  $mid$ ;
15         else if ( $x < a[mid]$ ) then
16             return BinSrch( $a, i, mid - 1, x$ );
17         else return BinSrch( $a, mid + 1, l, x$ );
18     }
19 }

```



```

1  Algorithm BinSearch( $a, n, x$ )
2  // Given an array  $a[1 : n]$  of elements in nondecreasing
3  // order,  $n \geq 0$ , determine whether  $x$  is present, and
4  // if so, return  $j$  such that  $x = a[j]$ ; else return 0.
5  {
6       $low := 1; high := n;$ 
7      while ( $low \leq high$ ) do
8      {
9           $mid := \lfloor (low + high)/2 \rfloor;$ 
10         if ( $x < a[mid]$ ) then  $high := mid - 1;$ 
11         else if ( $x > a[mid]$ ) then  $low := mid + 1;$ 
12         else return  $mid;$ 
13     }
14     return 0;
15 }

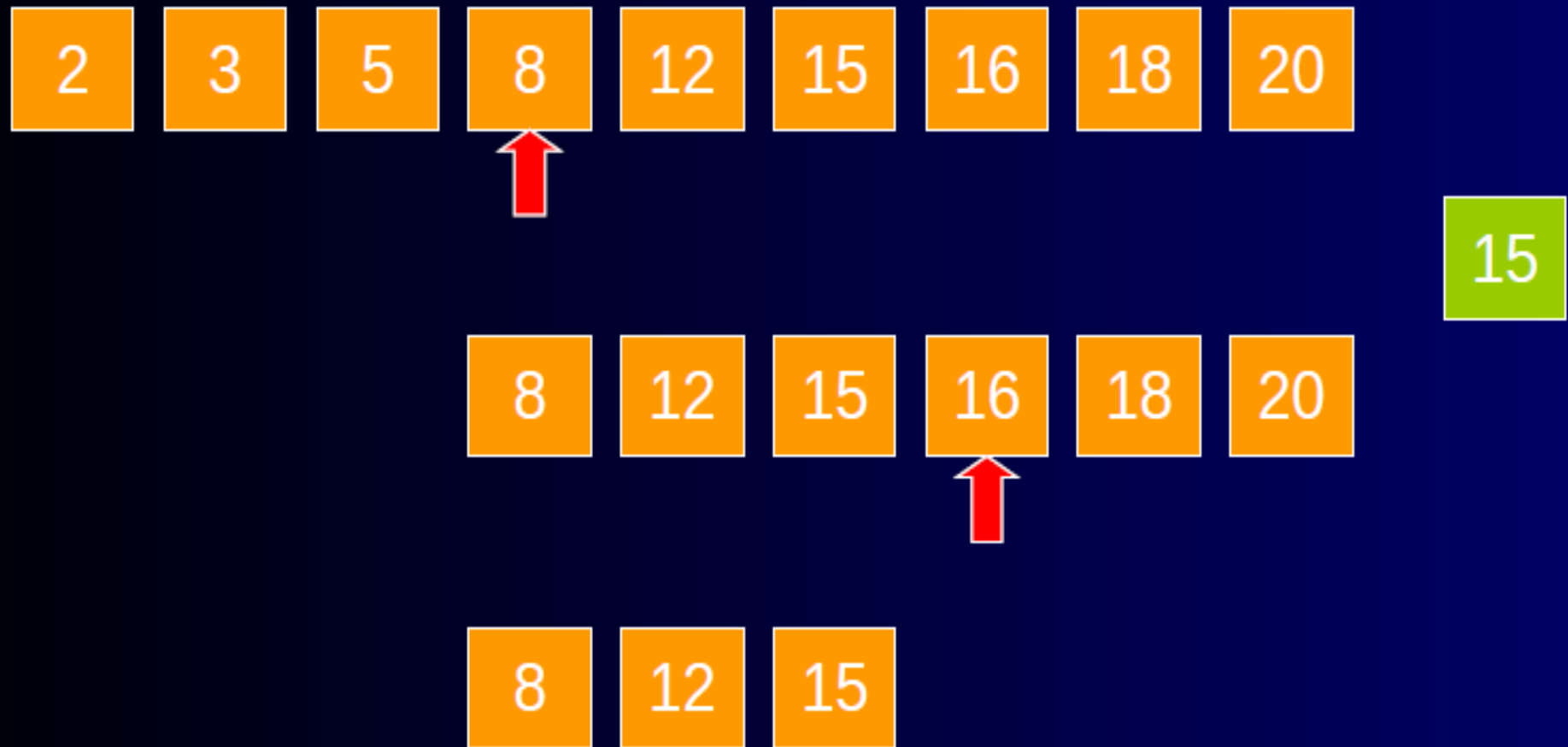
```

Algorithm 3.4 Iterative binary search

Binary Search : Analysis

- Successful search
 - Best: $\Theta(1)$
 - Average: $\Theta(\log n)$
 - Worst: $\Theta(\log n)$
- Unsuccessful search
 - Best:
 - Average: $\Theta(\log n)$
 - Worst:

Ternary Search: Demonstration



Min-Max Algorithm

- Problem is to find Minimum and Maximum from the set of n elements
- $\text{Min} = \text{Max} = a[1]$
 - For ($i=2$ to n)
 - { if ($a[i] < \text{Min}$) then $\text{Min} = a[i]$;
 - if ($a[i] > \text{Max}$) then $\text{Max} = a[i]$;} }
- Complexity: $2(n-1)$
- Best, Worst, Average Case

```
1  Algorithm StraightMaxMin( $a, n, max, min$ )
2  // Set  $max$  to the maximum and  $min$  to the minimum of  $a[1 : n]$ .
3  {
4       $max := min := a[1]$ ;
5      for  $i := 2$  to  $n$  do
6          {
7              if ( $a[i] > max$ ) then  $max := a[i]$ ;
8              if ( $a[i] < min$ ) then  $min := a[i]$ ;
9          }
10 }
```

Algorithm 3.5 Straightforward maximum and minimum

MIN-MAX

- When $a[i] < \text{Min}$,
- it can not be $> \text{Max}$
- Complexity –
- Best Case – $(n-1)$
- Worst Case – $2(n-1)$
- Average Case $< 2(n-1)$

MIN-MAX

- Use Divide and Conquer
- Split list in two sub-lists of equal size
- Continue the above step till sub-list has only two elements
- Find Min, Max of each group
- Compare Min, Max of each group and
- find the same for bigger groups

```

1  Algorithm MaxMin( $i, j, max, min$ )
2  //  $a[1 : n]$  is a global array. Parameters  $i$  and  $j$  are integers,
3  //  $1 \leq i \leq j \leq n$ . The effect is to set  $max$  and  $min$  to the
4  // largest and smallest values in  $a[i : j]$ , respectively.
5  {
6      if ( $i = j$ ) then  $max := min := a[i]$ ; // Small( $P$ )
7      else if ( $i = j - 1$ ) then // Another case of Small( $P$ )
8          {
9              if ( $a[i] < a[j]$ ) then
10                 {
11                      $max := a[j]; min := a[i];$ 
12                 }
13             else
14                 {
15                      $max := a[i]; min := a[j];$ 
16                 }
17             }
18         else
19             { // If  $P$  is not small, divide  $P$  into subproblems.
20               // Find where to split the set.
21                  $mid := \lfloor (i + j) / 2 \rfloor$ ;
22               // Solve the subproblems.
23                 MaxMin( $i, mid, max, min$ );
24                 MaxMin( $mid + 1, j, max1, min1$ );
25               // Combine the solutions.
26                 if ( $max < max1$ ) then  $max := max1$ ;
27                 if ( $min > min1$ ) then  $min := min1$ ;
28             }
29 }

```

MIN-MAX

- Many types of problems are solvable by reducing a problem of size n into some number a of independent subproblems, each of size $\leq \lceil n/b \rceil$, where $a \geq 1$ and $b > 1$.
- The time complexity to solve such problems is given by a recurrence relation:

$$- T(n) = a T(\lceil n/b \rceil) + g(n)$$

Time for each subproblem

Time to combine the solutions of the subproblems into a solution of the original problem.

MIN-MAX

- If $n=1$, the number is itself min or max
- If $n>1$, divide the numbers into two lists. Decide the min & max in the first list. Then choose the min & max in the second list.
- Decide the min & max of the entire list.
- Thus,

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

MIN-MAX

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k , then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \sum_{1 \leq i \leq k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

- Put $2^k=n$ then $T(n)=n/2+n-2=3n/2-2$

MIN-MAX

If n is even: $\frac{n}{2} + 2\left(\frac{n}{2} - 1\right) = \frac{3n}{2} - 2$

If n is odd: $\frac{n}{2} - 1 + 2\left(\frac{n-1}{2} - 1\right) + 1 = \frac{n-2}{2} + 2\left(\frac{n-3}{2}\right) + 1 = \frac{3n}{2} - 3$

Which is about $\frac{3}{4}$ of the number of steps required with the obvious method.

- It saves 25%

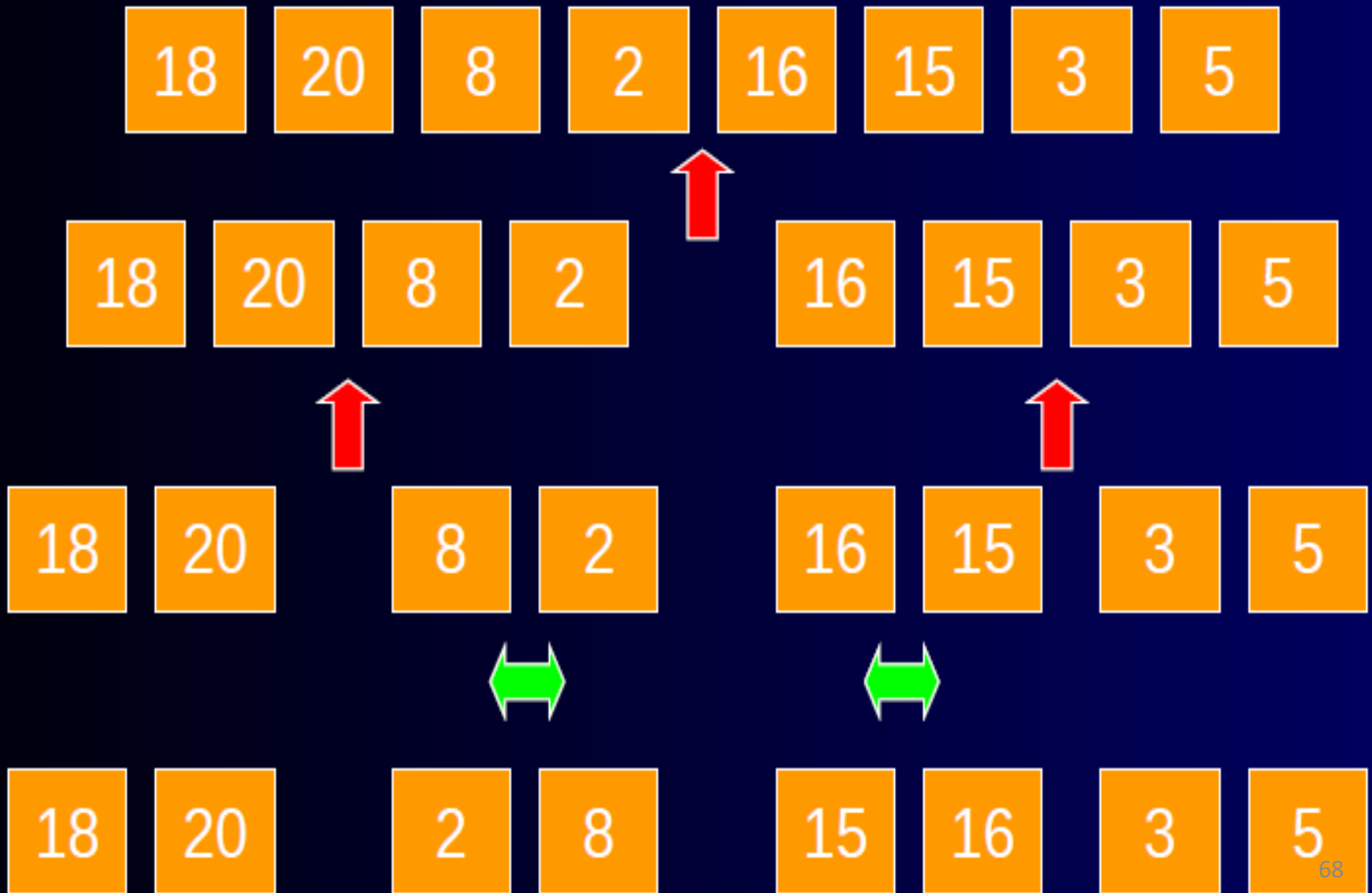
Merge Sort

- Given a sequence of elements (keys)
 $a[1], \dots, a[n]$,
- split it into two sets $a[1] \dots \lfloor n/2 \rfloor$ and
 $a[\lfloor n/2 \rfloor + 1] \dots a[n]$

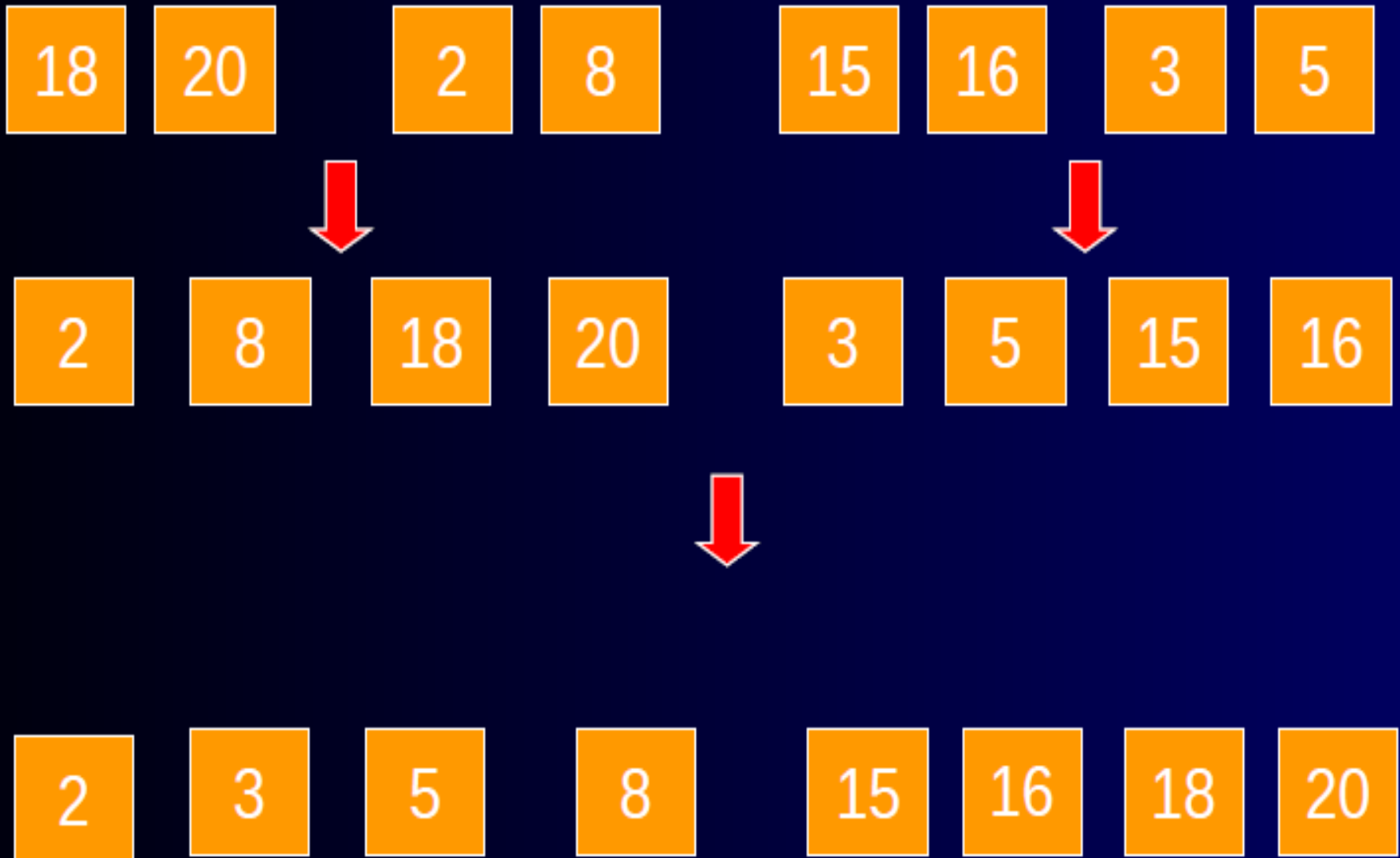
Each set is individually sorted

- and resulting sorted sequences are merged
- to produce a single sorted sequence of n elements

Merge Sort : Split



Merge Sort : Merge



```
1  Algorithm MergeSort(low, high)
2  // a[low : high] is a global array to be sorted.
3  // Small(P) is true if there is only one element
4  // to sort. In this case the list is already sorted.
5  {
6      if (low < high) then // If there are more than one element
7      {
8          // Divide P into subproblems.
9          // Find where to split the set.
10         mid :=  $\lfloor (low + high) / 2 \rfloor$ ;
11         // Solve the subproblems.
12         MergeSort(low, mid);
13         MergeSort(mid + 1, high);
14         // Combine the solutions.
15         Merge(low, mid, high);
16     }
17 }
```

```

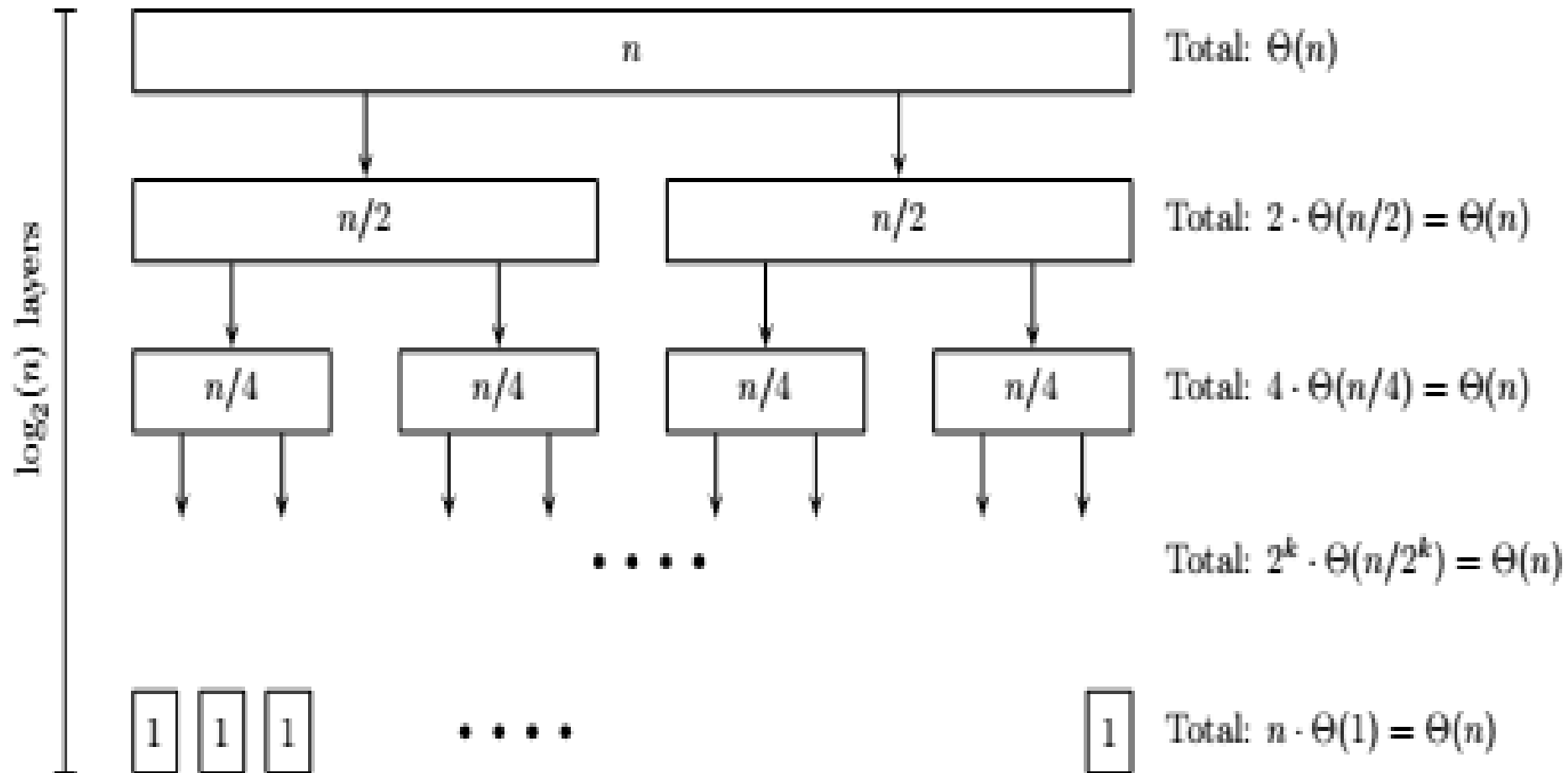
1  Algorithm Merge(low, mid, high)
2  // a[low : high] is a global array containing two sorted
3  // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4  // is to merge these two sets into a single set residing
5  // in a[low : high]. b[ ] is an auxiliary global array.
6  {
7      h := low; i := low; j := mid + 1;
8      while ((h ≤ mid) and (j ≤ high)) do
9      {
10         if (a[h] ≤ a[j]) then
11         {
12             b[i] := a[h]; h := h + 1;
13         }
14         else
15         {
16             b[i] := a[j]; j := j + 1;
17         }
18         i := i + 1;
19     }
20     if (h > mid) then
21         for k := j to high do
22         {
23             b[i] := a[k]; i := i + 1;
24         }
25     else
26         for k := h to mid do
27         {
28             b[i] := a[k]; i := i + 1;
29         }
30     for k := low to high do a[k] := b[k];
31 }

```

Merge Sort : Analysis

- Division into sub-groups - $\log n$ steps
- In each step n comparisons
- So Complexity is $n \log n$

Merge Sort : Analysis



Merge Sort : Analysis

$$\begin{aligned}T(n) &= 2 T(n/2) + n \\&= 2 [2 T(n/4) + n/2] + n \\&= 4 T(n/4) + 2n \\&= 4 [2 T(n/8) + n/4] + 2n \\&= 8 T(n/8) + 3n \\&= 16 T(n/16) + 4n \\&= 2^k T(n/2^k) + k n\end{aligned}$$

Merge Sort : Analysis

$$n/2^k = 1 \quad OR \quad n = 2^k \quad OR \quad \log_2 n = k$$

Continuing with the previous derivation we get the following since $k = \log_2 n$:

$$\begin{aligned} &= 2^k T(n/2^k) + k n \\ &= 2^{\log_2 n} T(1) + (\log_2 n) n \\ &= n + n \log_2 n \\ &= O(n \log n) \end{aligned}$$

Merge Sort : Analysis

➤ Observation

If no. of elements in the list are < 16 then insertion sort performs better than merge sort.

➤ Conclusion

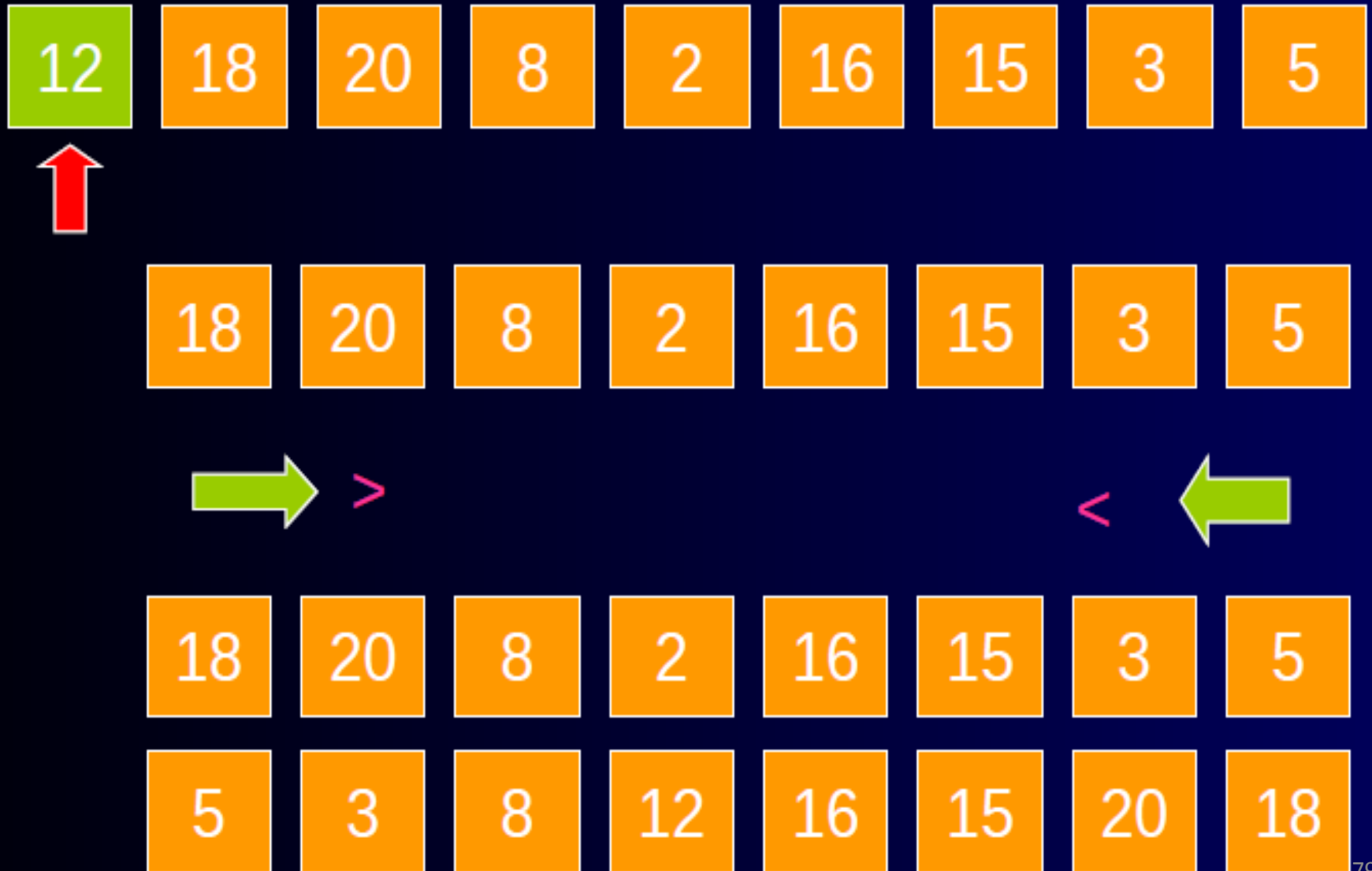
Combine the two methods

```
1  Algorithm InsertionSort( $a, n$ )
2  // Sort the array  $a[1 : n]$  into nondecreasing order,  $n \geq 1$ .
3  {
4      for  $j := 2$  to  $n$  do
5      {
6          //  $a[1 : j - 1]$  is already sorted.
7           $item := a[j]; i := j - 1;$ 
8          while  $((i \geq 1) \textbf{ and } (item < a[i]))$  do
9          {
10              $a[i + 1] := a[i]; i := i - 1;$ 
11          }
12           $a[i + 1] := item;$ 
13      }
14 }
```

Quick Sort

- Division into two sub arrays is done
- in such a way that no merging is necessary later.
- Re-arranging the elements in $a[1:n]$ such that
- $a[i] \leq a[j]$ for all i between 1 and m
- and all j between $m+1$ to n
- for some m , $1 \leq m \leq n$.
- Thus the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted.

Quick Sort



```

1  Algorithm Partition( $a, m, p$ )
2  // Within  $a[m], a[m + 1], \dots, a[p - 1]$  the elements are
3  // rearranged in such a manner that if initially  $t = a[m]$ ,
4  // then after completion  $a[q] = t$  for some  $q$  between  $m$ 
5  // and  $p - 1$ ,  $a[k] \leq t$  for  $m \leq k < q$ , and  $a[k] \geq t$ 
6  // for  $q < k < p$ .  $q$  is returned. Set  $a[p] = \infty$ .
7  {
8       $v := a[m]; i := m; j := p;$ 
9      repeat
10     {
11         repeat
12              $i := i + 1;$ 
13         until ( $a[i] \geq v$ );
14
15         repeat
16              $j := j - 1;$ 
17         until ( $a[j] \leq v$ );
18
19         if ( $i < j$ ) then Interchange( $a, i, j$ );
20     } until ( $i > j$ );
21      $a[m] := a[j]; a[j] := v; \text{ return } j;$ 
22 }

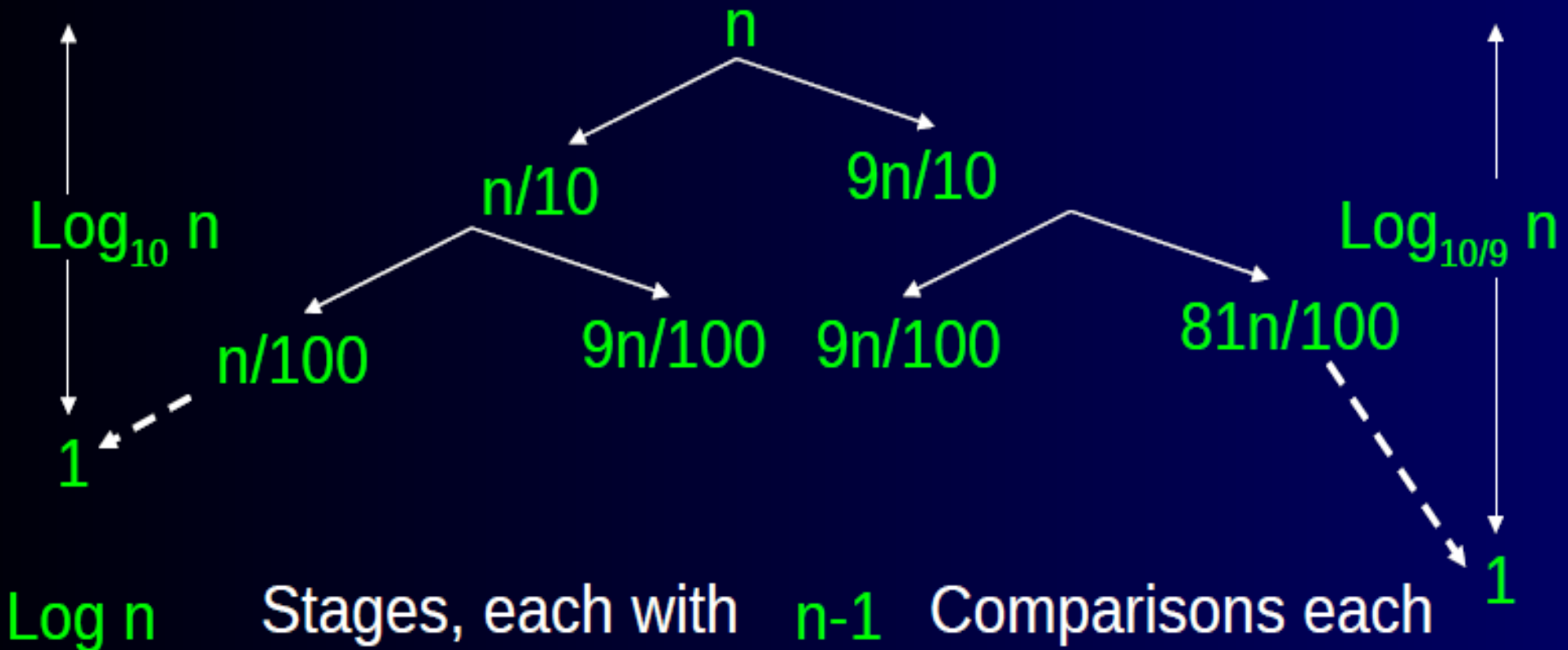
1  Algorithm Interchange( $a, i, j$ )
2  // Exchange  $a[i]$  with  $a[j]$ .
3  {
4       $p := a[i];$ 
5       $a[i] := a[j]; a[j] := p;$ 
6  }

```

```
1  Algorithm QuickSort( $p, q$ )
2  // Sorts the elements  $a[p], \dots, a[q]$  which reside in the global
3  // array  $a[1 : n]$  into ascending order;  $a[n + 1]$  is considered to
4  // be defined and must be  $\geq$  all the elements in  $a[1 : n]$ .
5  {
6      if ( $p < q$ ) then // If there are more than one element
7      {
8          // divide  $P$  into two subproblems.
9           $j := \text{Partition}(a, p, q + 1)$ ;
10         //  $j$  is the position of the partitioning element.
11         // Solve the subproblems.
12         QuickSort( $p, j - 1$ );
13         QuickSort( $j + 1, q$ );
14         // There is no need for combining solutions.
15     }
16 }
```

Quick Sort: Average Case Analysis

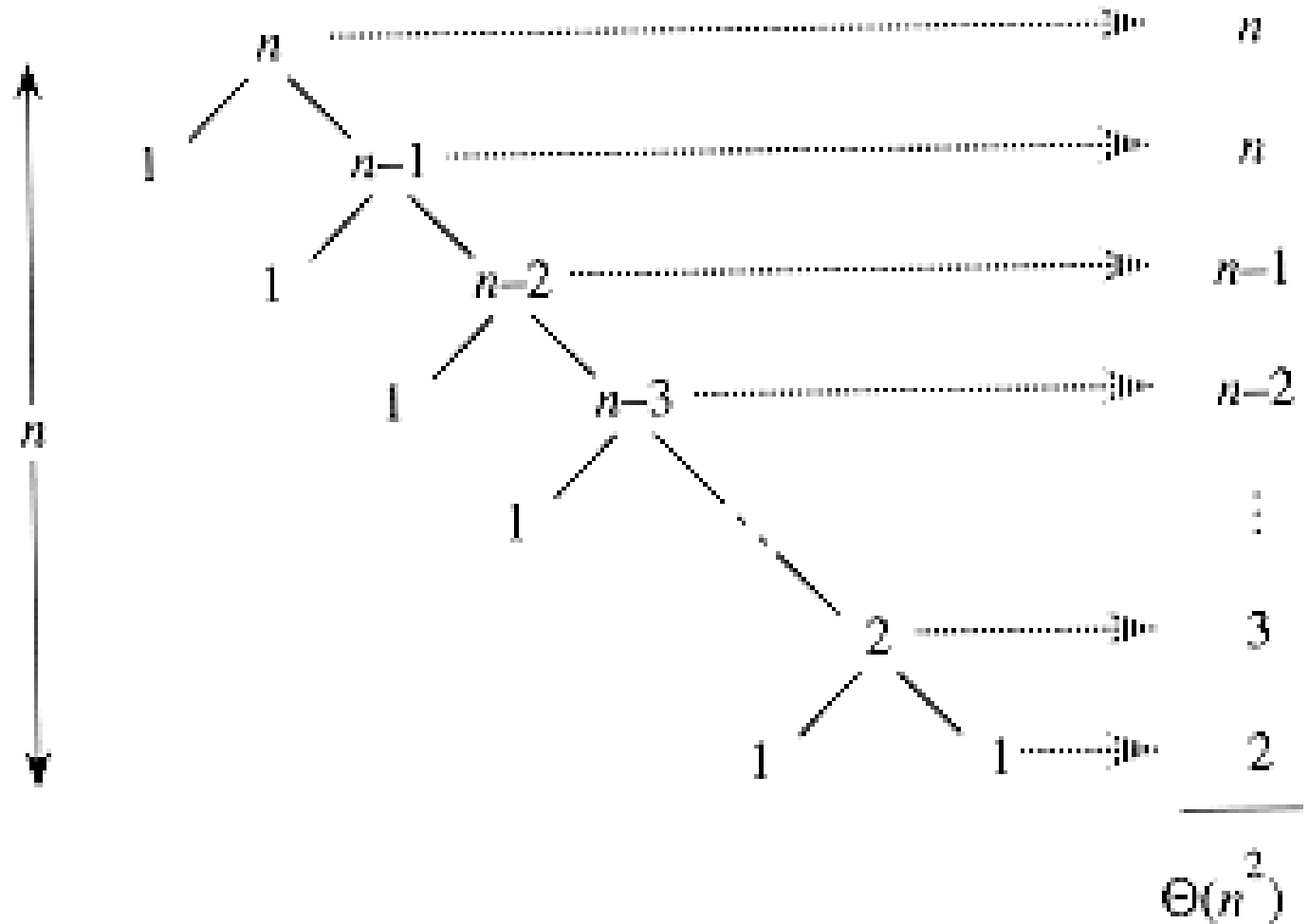
Assume the set gets divided in proportion of 1:9 at every step



$$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Quick Sort : Analysis

- Worst Case
-



$$\begin{aligned}
 T(n) &= T(n-1) + \Theta(n) \\
 &= \sum_{k=1}^n \Theta(k) \\
 &= \Theta\left(\sum_{k=1}^n k\right)
 \end{aligned}$$

Quick Sort : Analysis

- Best Case

Selection of Partition Element

- Selection Approaches
 - Trivial: First/Last
 - Better: Random
 - Ideal: Median
 - Best/Practical: Median of Median

Selection

- Given a list $a[1:n]$, select the k th
($1 \leq k \leq n$) smallest element.

- Use Quick sort? How?

Compare k with index of pivot element

- D&C Approach
- Complexity?

Worst Case - $O(n^2)$

Average Case - $O(n)$

```

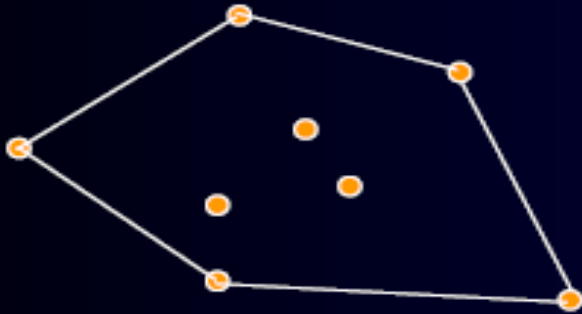
1  Algorithm Select1( $a, n, k$ )
2  // Selects the  $k$ th-smallest element in  $a[1 : n]$  and places it
3  // in the  $k$ th position of  $a[ ]$ . The remaining elements are
4  // rearranged such that  $a[m] \leq a[k]$  for  $1 \leq m < k$ , and
5  //  $a[m] \geq a[k]$  for  $k < m \leq n$ .
6  {
7       $low := 1; up := n$ 
8
9      repeat
10     {
11         // Each time the loop is entered,
12         //  $1 \leq low \leq k \leq up \leq n$ 
13          $j := \text{Partition}(a, low, up);$ 
14         //  $j$  is such that  $a[j]$  is the  $j$ th-smallest value in  $a[ ]$ .
15         if ( $k = j$ ) then return;
16         else if ( $k < j$ ) then  $up := j$ ; //  $j$  is the new upper limit.
17         else  $low := j + 1$ ; //  $j + 1$  is the new lower limit.
18     } until (false);
19 }

```

Algorithm 3.18 Finding the k th-smallest element

Convex Hull

- Convex hull of set of S points in the plane is defined to be
- smallest convex polygon containing all points of S .
- A polygon is defined to be convex if for any two points $P1$ & $P2$ inside the polygon,
the directed line segment from $P1$ to $P2$ is fully contained in polygon.



There are two variants of Convex Hull Problem

1. Obtain the vertices of the convex hull (Extreme Points)
2. Obtain the vertices of convex hull in some order (Clockwise)

- To check whether point $p \in S$ (point on the boundary), look at each possible triplet (Set of three vertices). Check if point p lies within any such triplet, if yes p is not boundary point.
 - Checking if p lies within given triangle (of triplet) can be done in $\Theta(1)$ time.
 - There are n^3 triplets, so for one point it can be done in $\Theta(n^3)$ time.
 - For all points, it will take $\Theta(n^4)$ time.

Quick Hull Algorithm

Let X be set of n points

- Identify two points (P_1, P_2) with largest and smallest x co-ordinate
 - If there is tie, let the point be P_1' and P_1'' (smallest X co-ordinate)
 - Only consider points to the left of $P_1'P_2$ and points to the right of $P_1''P_2$
- Both P_1 and P_2 are part of convex hull
- X be divided into X_1 and X_2 such that X_1/X_2 has all points to the left/right of P_1P_2
- Both X_1 and X_2 includes points P_1 and P_2
- X_1/X_2 is called upper/lower hull
- Convex hull computed for X_1 and X_2 recursively using D&C the union of all these convex hull is final required convex hull.

Finding convex hull of X_1 , with P_1P_2

- Determine point P of X_1 , that belong to convex hull of X_1 and use it for partitioning
- P is obtained by computing area formed by P_1PP_2 for each P of X_1 and picking the largest area
- Let the point be P_3 , the problem now can be subdivided by considering
 - All points of X_1 that are to the left of P_1P_3
 - All points of X_1 that are to the left of P_3P_2
 - Rest all the points are interior to the triangle $P_1P_3P_2$ so they are not considered
- Complexity: Worst case $O(n^2)$, average $O(n \log n)$

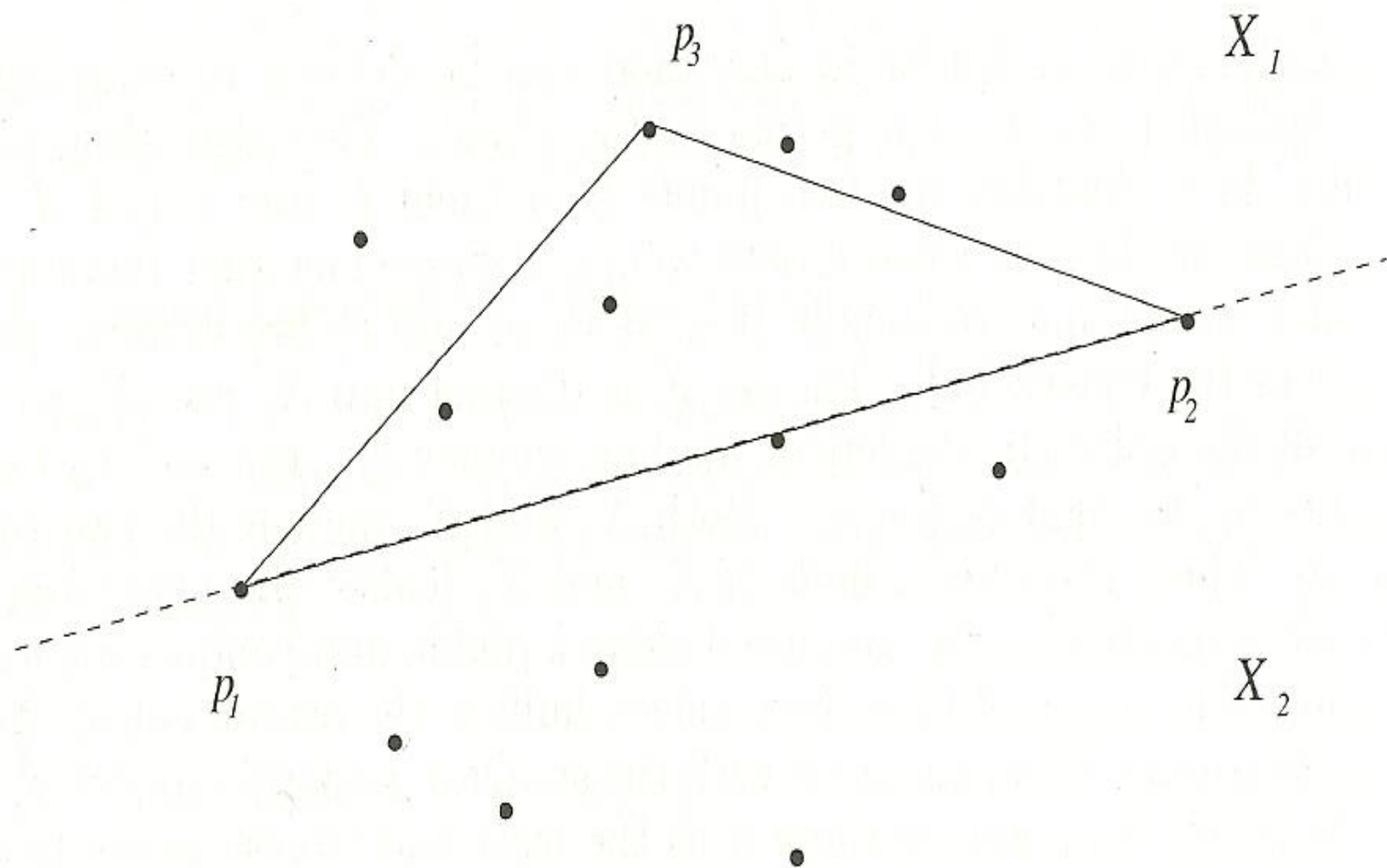
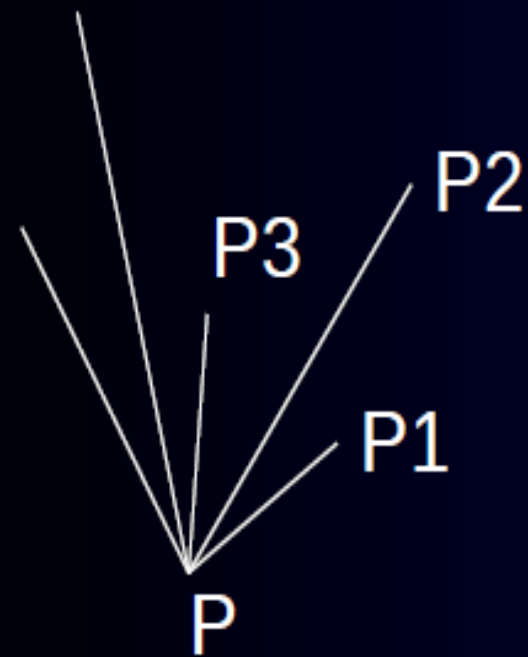


Figure 3.10 Identifying a point on the convex hull of X_1

Graham Scan

- Identify P with lowest y co-ordinate
- Sort the remaining points as per the angle subtained by segment P and that point with X axis



- Lets the sorted list be P_1, P_2, P_3, \dots
- Take three successive points P_1, P_2, P_3 ,
 - if it is left turn include the points in convex hull
 - If it is right turn, exclude middle point P_2

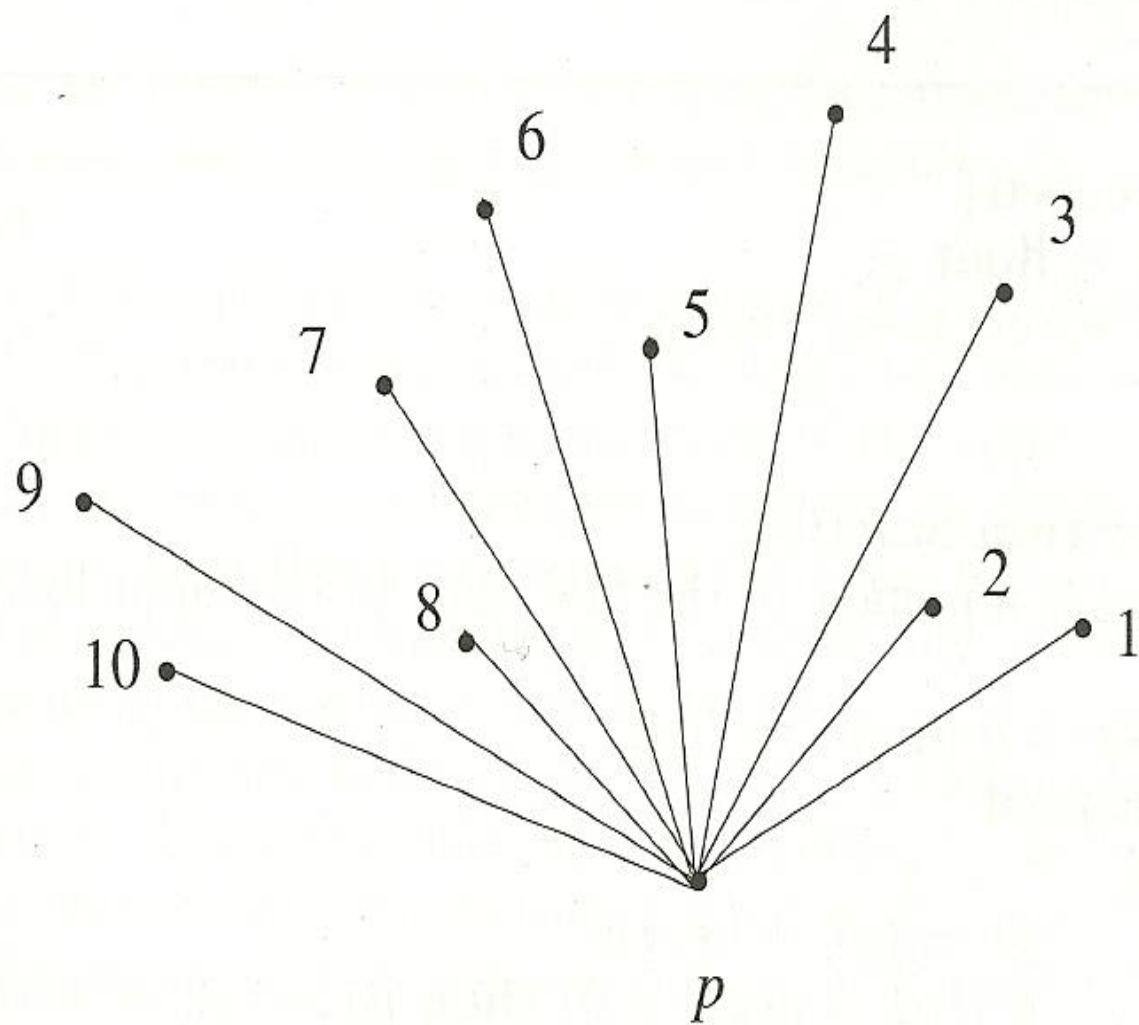


Figure 3.11 Graham's scan algorithm sorts the points first

N log n Divide and Conquer Algorithm

- It is as like Quick Hull
- Partitioning is done according to the x-coordinate values of the points using the median x-coordinate as the splitter
- Upper Hulls are recursively computed for the two halves
- These two hulls are then merged by finding the line of tangent
- (i.e. a straight line connecting a point each from the two halves, such that all the points of X are on one side of the line)

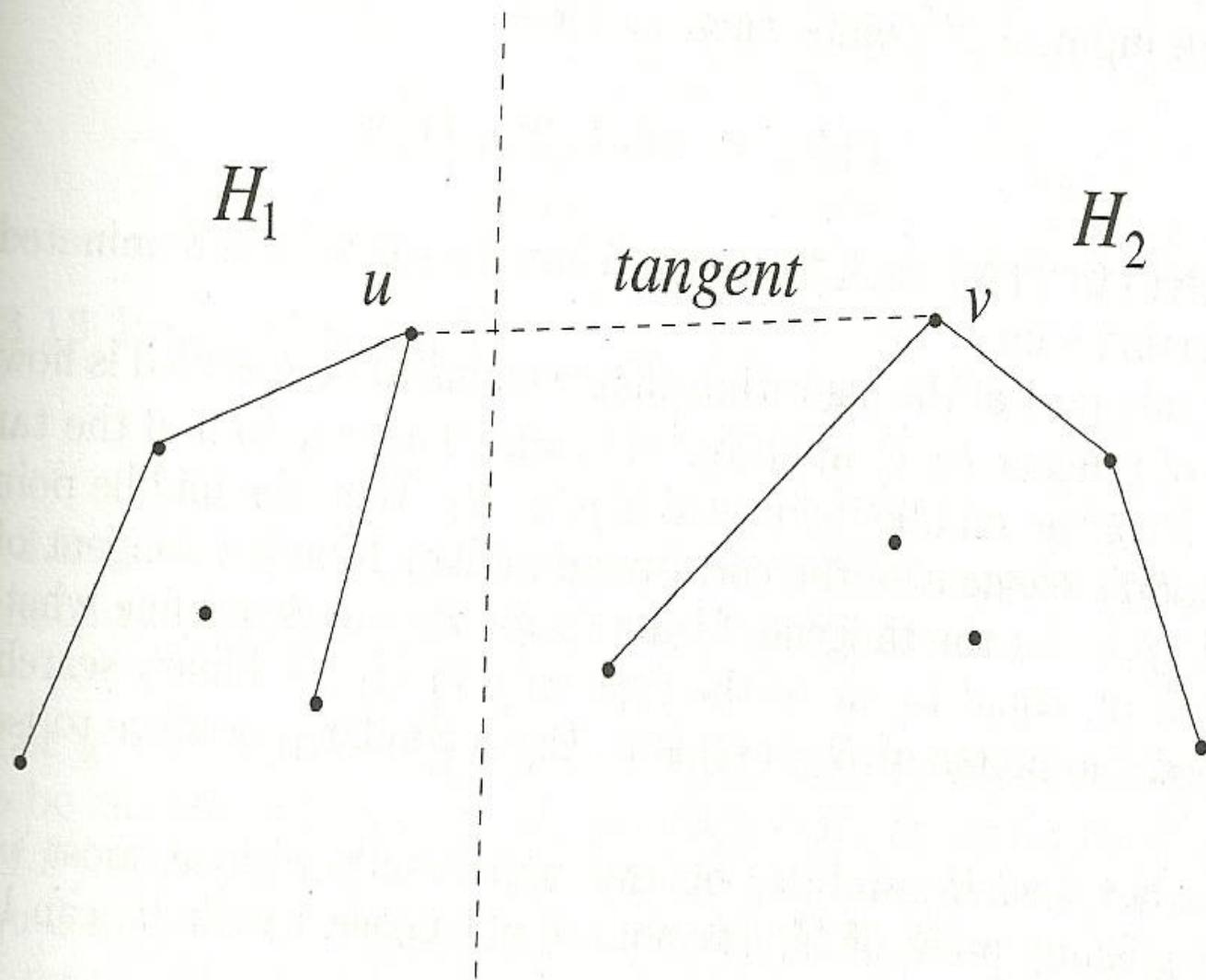


Figure 3.12 Divide and conquer to compute the convex hull

Thank You