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 option , the resulting

  instructions are definite.
- Graphical representations called **Flow Charts** are another possibility,

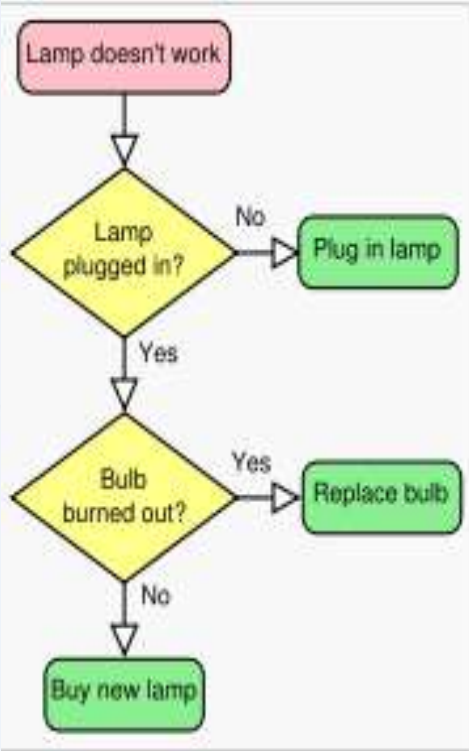
  but they work well only if the algorithm is small and simple.

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* := *value*1 **to** *value*2 **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);
}
```

gcd(468, 24) gcd(24, 12) => 12

gcd(135, 19) gcd(19, 2) gcd(2, 1) => 1

Recursive Algorithms

- GCD - **Dijkstra's Algorithm**

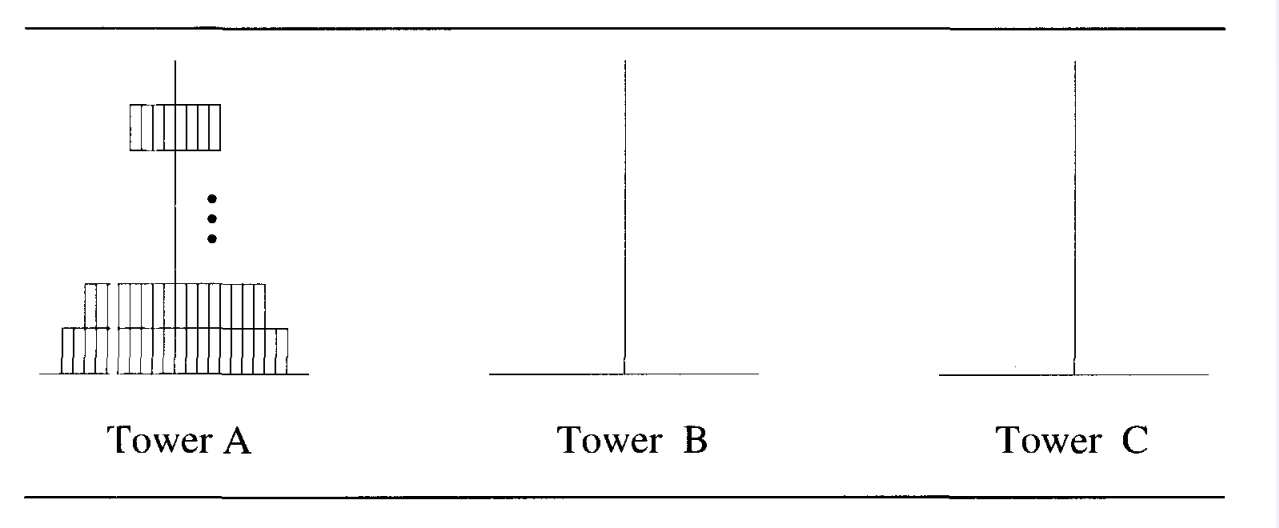
```
int gcd(int m, int n)
{
   if(m == n)
      return m;
   else if (m > n)
      return gcd(m-n, n);
   else
      return gcd(m, n-m);
}
```

Recursive Algorithms

- GCD - **Dijkstra's Algorithm**
- gcd(468, 24)
- gcd(444, 24)
- gcd(420, 24) ...
- gcd(36, 24)
- gcd(12, 24) (Now n is bigger)
- gcd(12, 12) (Same) => 12

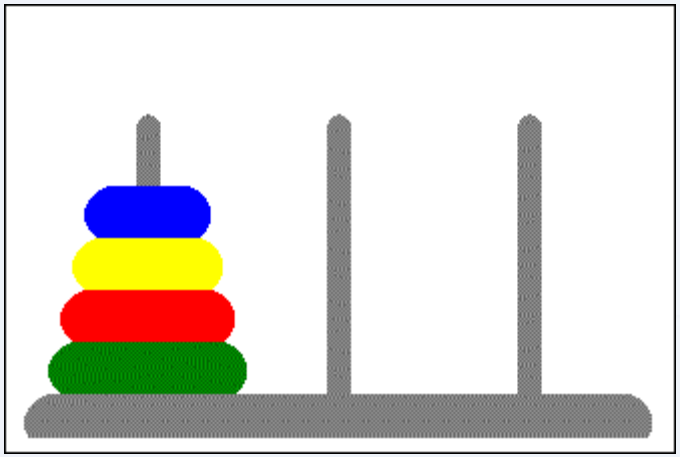
Recursive Algorithms

- 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

  $S(P)=c + Sp$

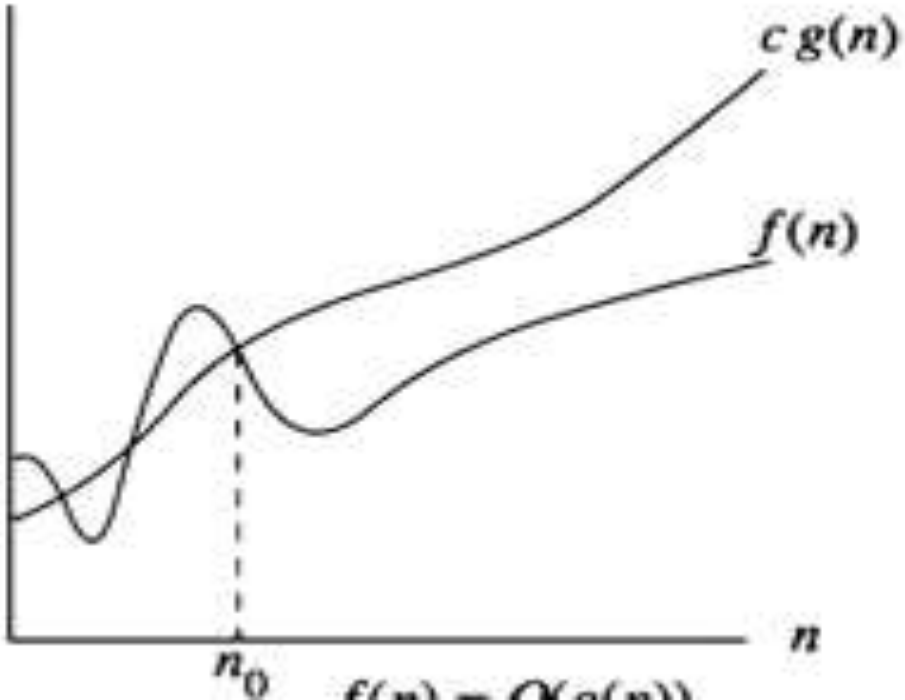
Time Complexity

- 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)<= c * g(n) for all n, n>n_0

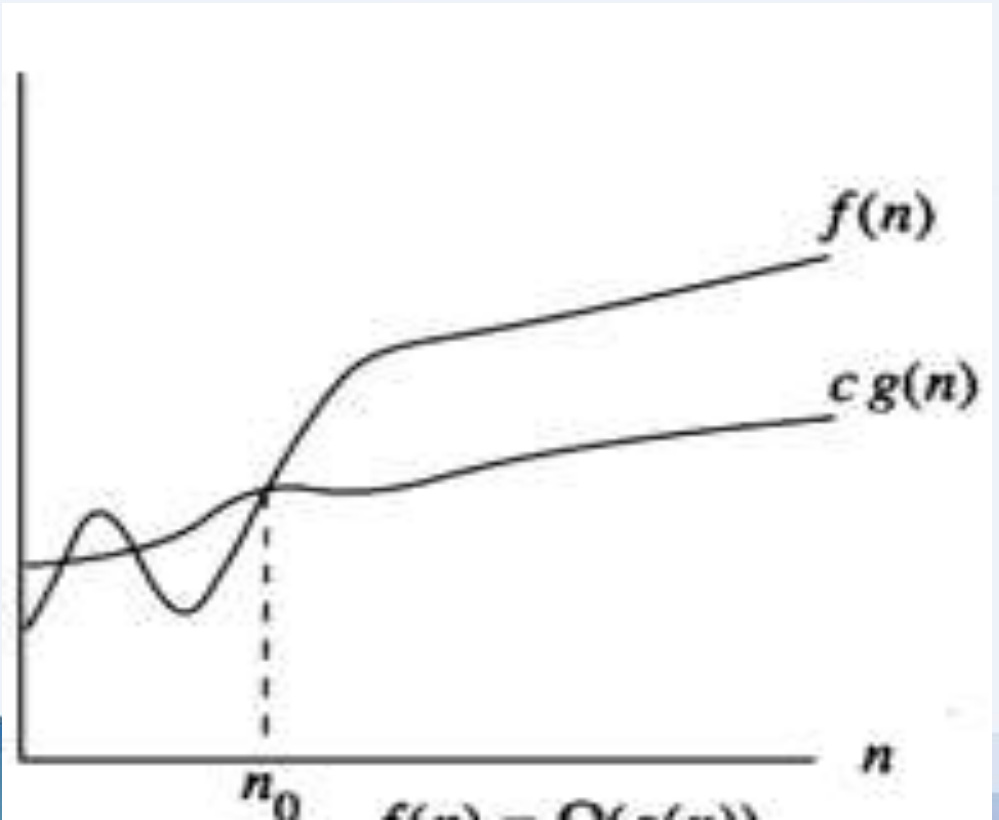


It specifies Upper Bound Worst Case complexity.

Asymptotic Notation (O, Ω, Θ)

- Ω (Omega) –

  The function f(n)= Ω(g(n))
  iff there exist positive constants c and n_0 such that f(n)>= c * g(n) for all n, n>n_0

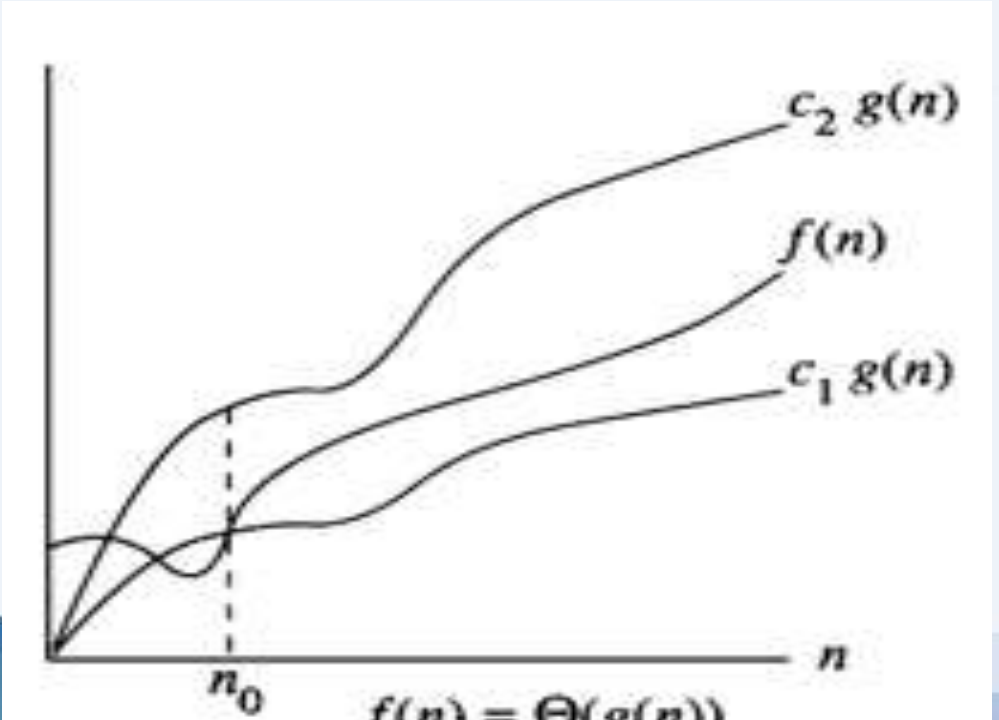


It specifies Lower Bound Best Case complexity.

Asymptotic Notation (O, Ω, Θ)

- Θ (Theta) –

  The function f(n)= Θ(g(n))
  iff there exist positive constants c1, c2 and n_0 such that c1 * g(n) <=f(n)<= c2 * 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
  - $\lim_{n \to \infty} f(n)/g(n) = 0$
- $g(x)$ 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)= ω(g(n)) iff
  - $\lim_{n \to \infty} g(n)/f(n) = 0$

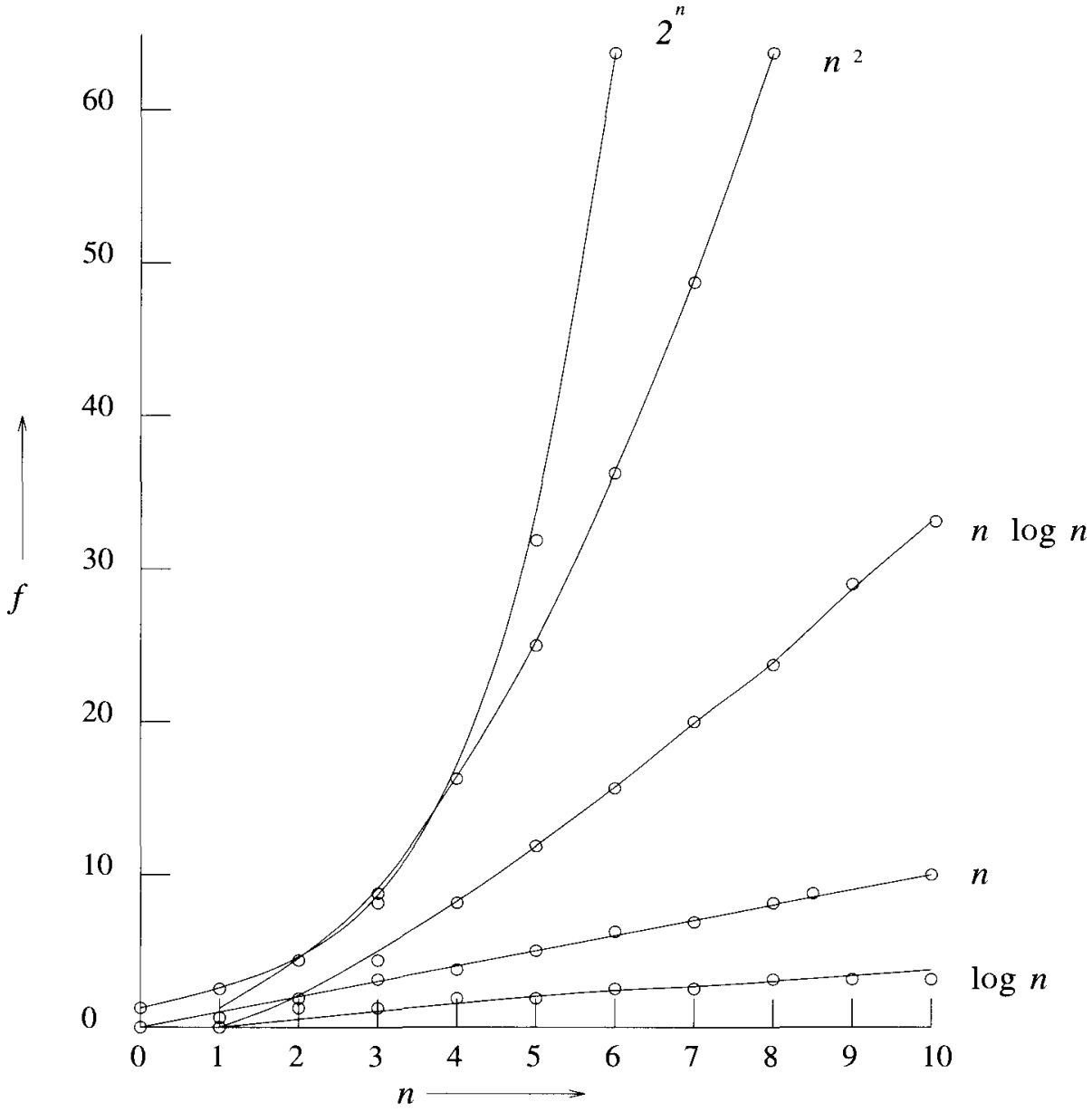
Practical Complexities

Time complexity		Example
O(1)	*constant*	Adding to the front of a linked list
O(log *N*)	*log*	Finding an entry in a sorted array
O(*N*)	*linear*	Finding an entry in an unsorted array
O(*N* log *N*)	*n-log-n*	Sorting n items by ‘divide-and-conquer’
O(N^2)	*quadratic*	Shortest path between two nodes in a graph
O(N^3)	*cubic*	Simultaneous linear equations
O(2^N)	*exponential*	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 $n = 10{,}000$


2^n
n^2
$n \log n$
n
$\log n$
f
n
0
10
20
30
40
50
60
0 1 2 3 4 5 6 7 8 9 10

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 E1 and E2 are said to be mutually exclusive
- if they do not have any common sample points,
- that is $E1 \cap E2 = \Phi$

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, ... , 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<=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, $1 \le i \le n$, 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

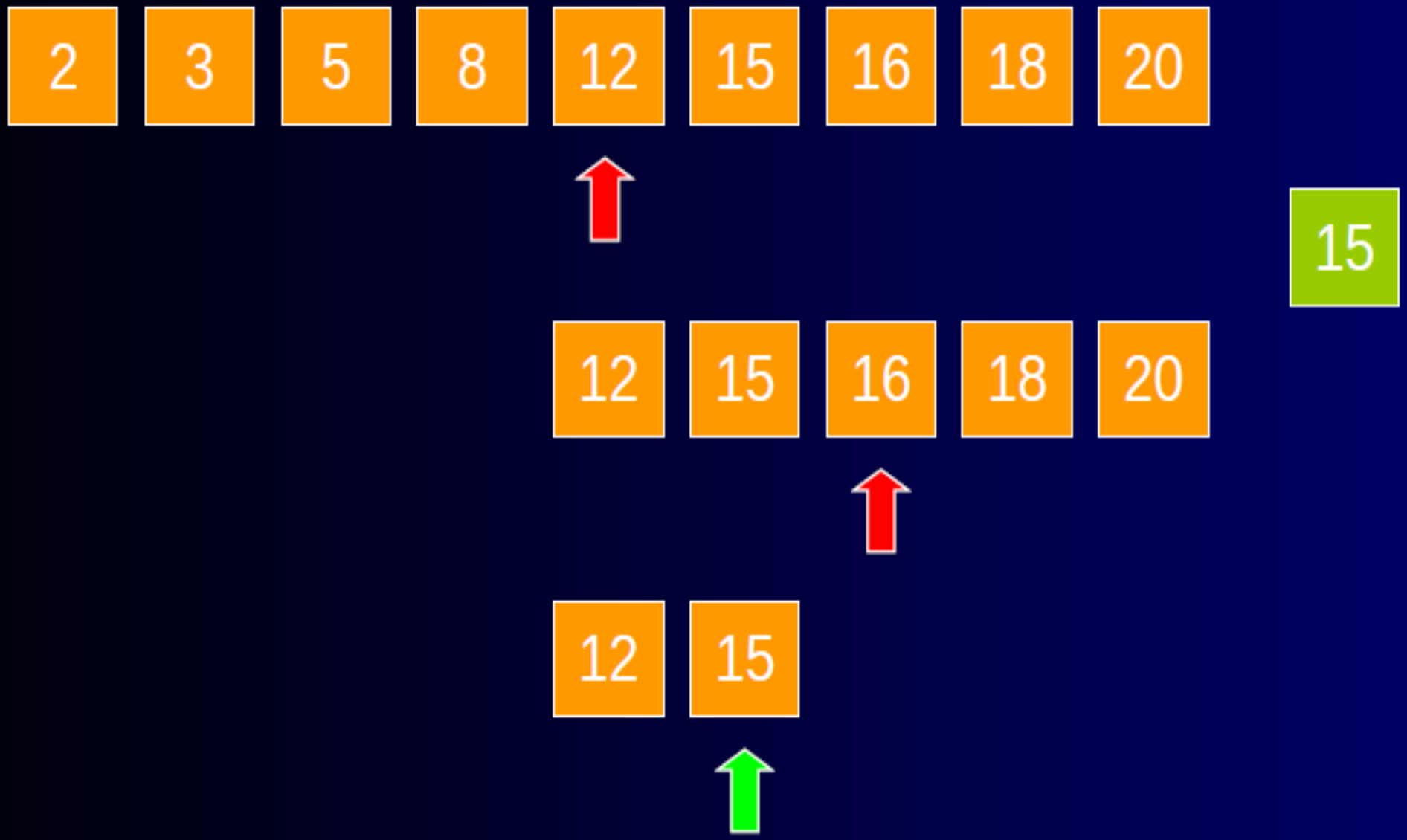
  Divide the list in two equal parts, look for middle element

  If $x \le$ middle, discard second half else discard first half

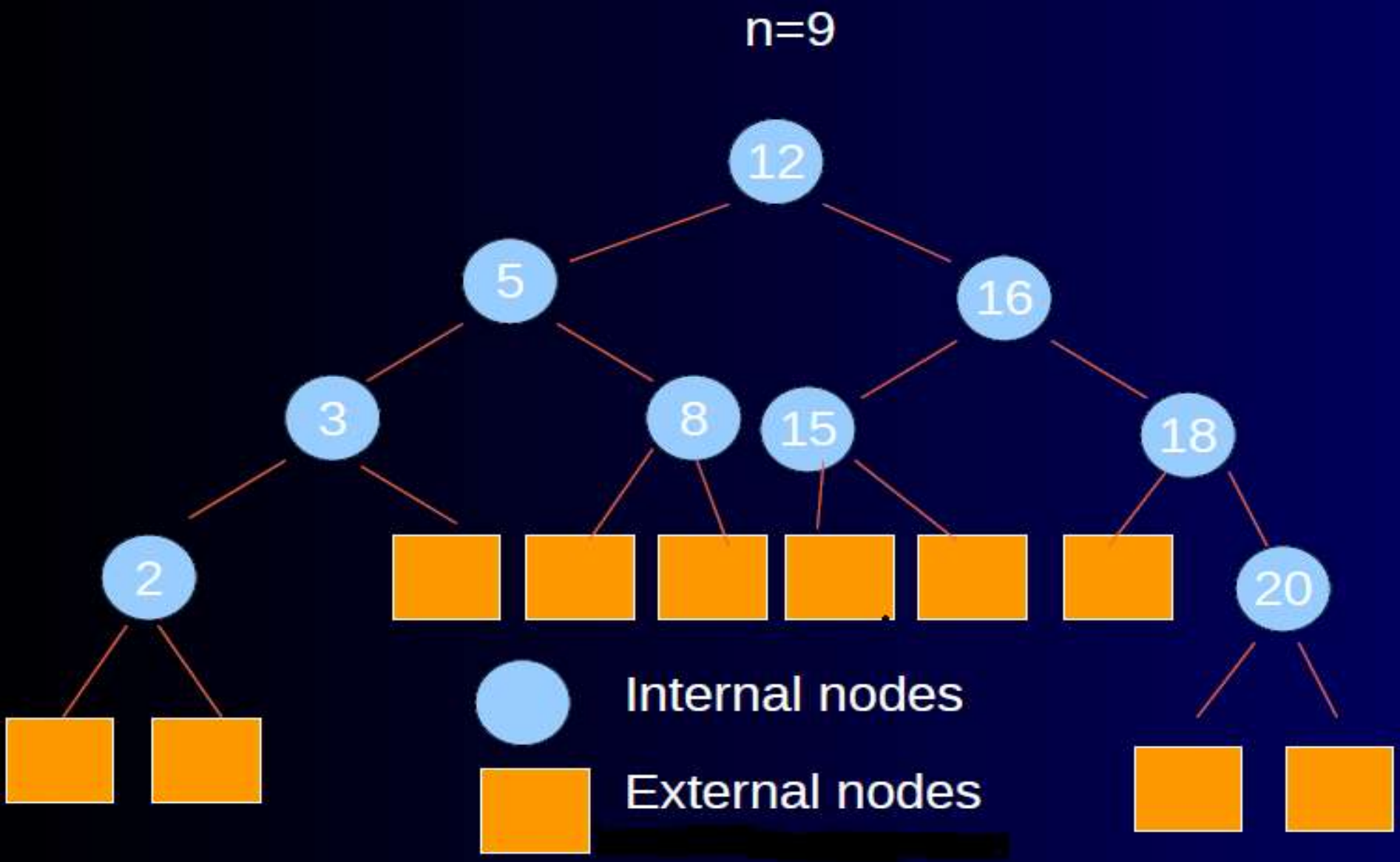
  Repeat till

  - element is found
  - elements not present in the list

Binary Search : Demonstration



Binary Search : Decision Tree



```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ 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 := ⌊(i + l)/2⌋;
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  }
```

Algorithm 3.3 Recursive binary search

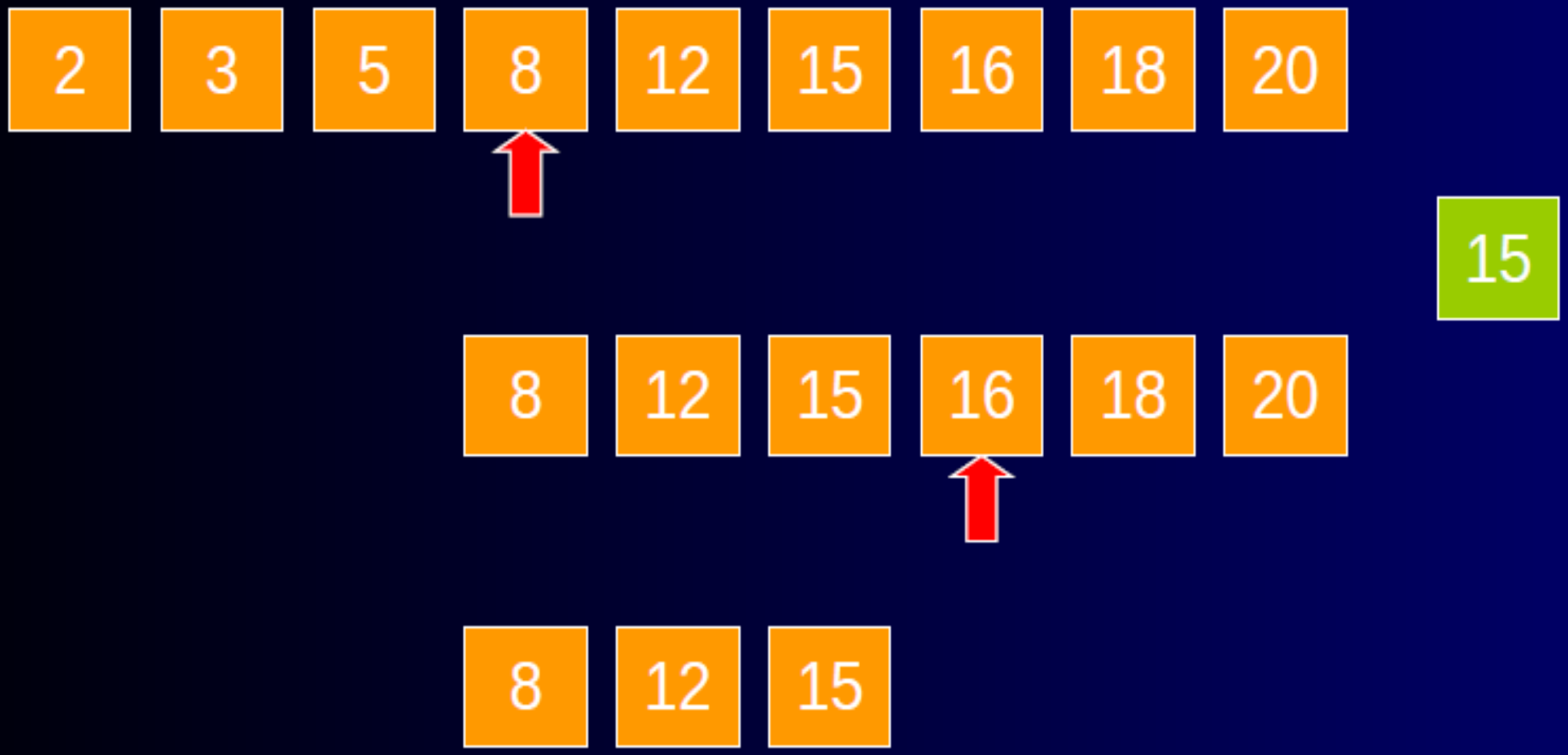
```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
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
- Min=Max=a[1]
  - For (i=2 to n)
  - { if (a[i] < Min) then Min=a[i];
  - if (a[i] > Max) then 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] < Min,
- it can not be >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 ≤ i ≤ j ≤ 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 := ⌊(i + j)/2⌋;
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  }
```

Algorithm 3.6 Recursively finding the maximum and minimum

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

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

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

MIN-MAX

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

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

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

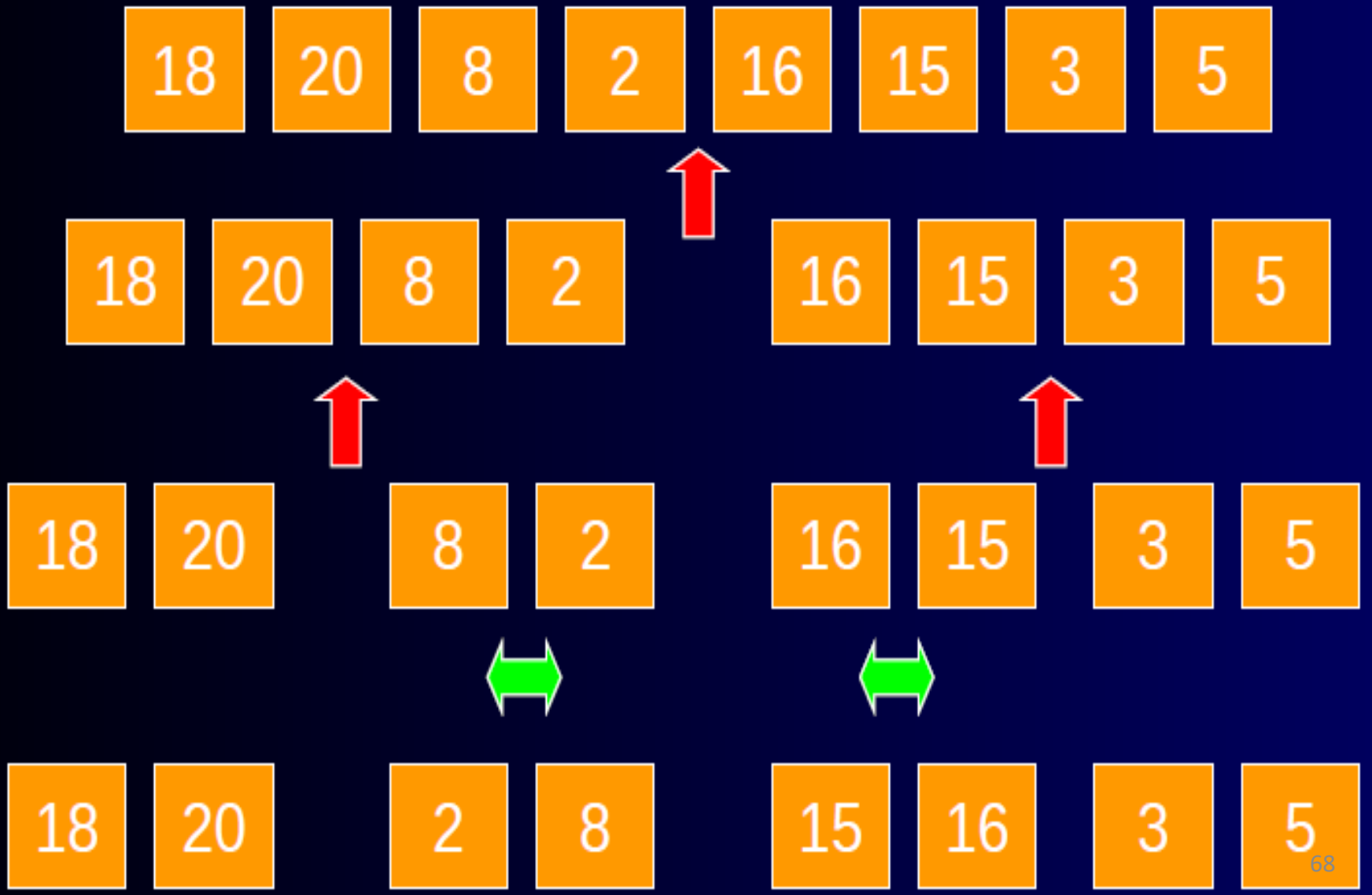
- It saves 25%

Merge Sort

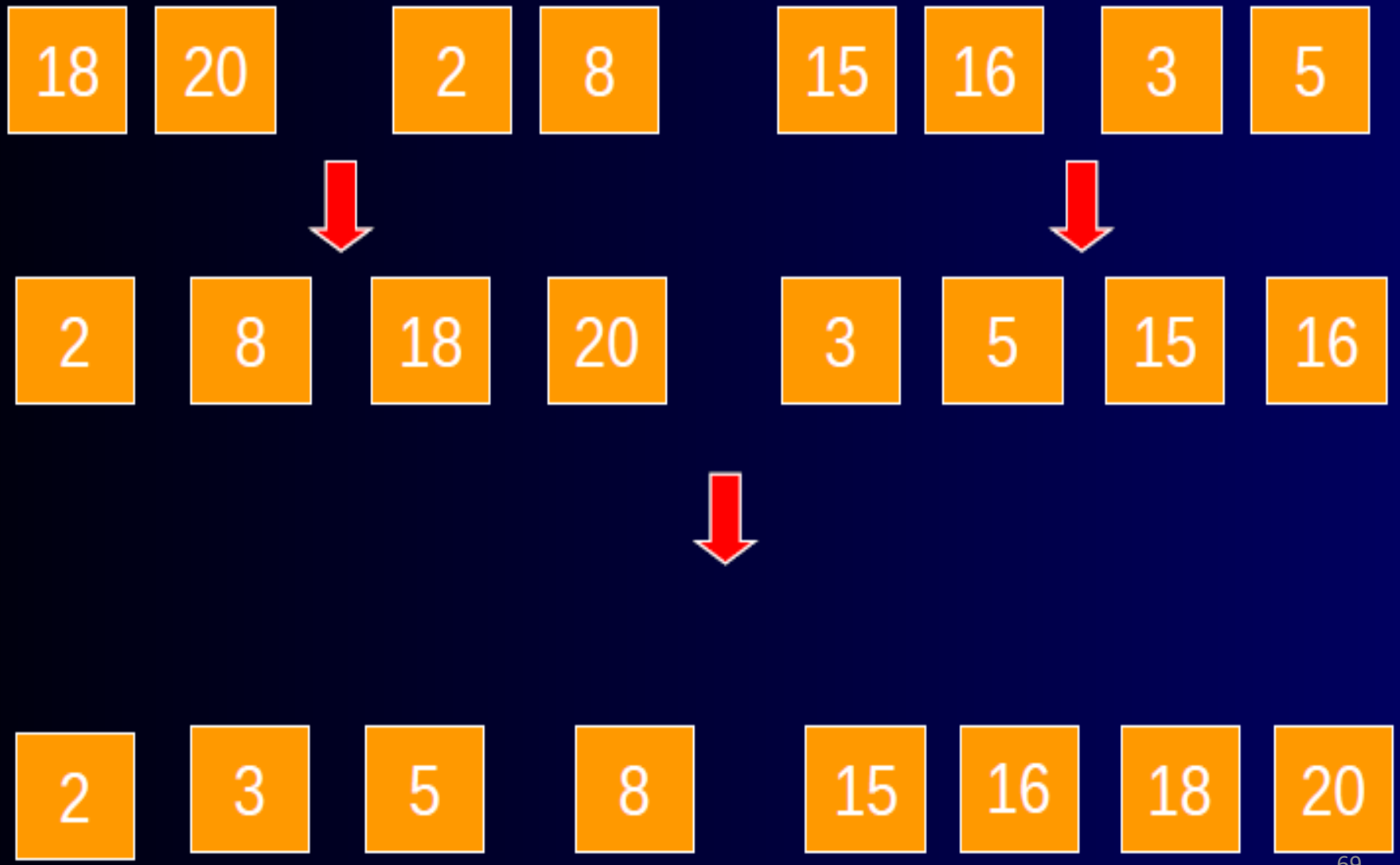
- Given a sequence of elements (keys) a[1],…a[n],
- split it into two sets a[1]… [$\lfloor n/2 \rfloor$] and a[$\lfloor n/2 \rfloor$ +1]….a[n]

  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 := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

Algorithm 3.7 Merge sort

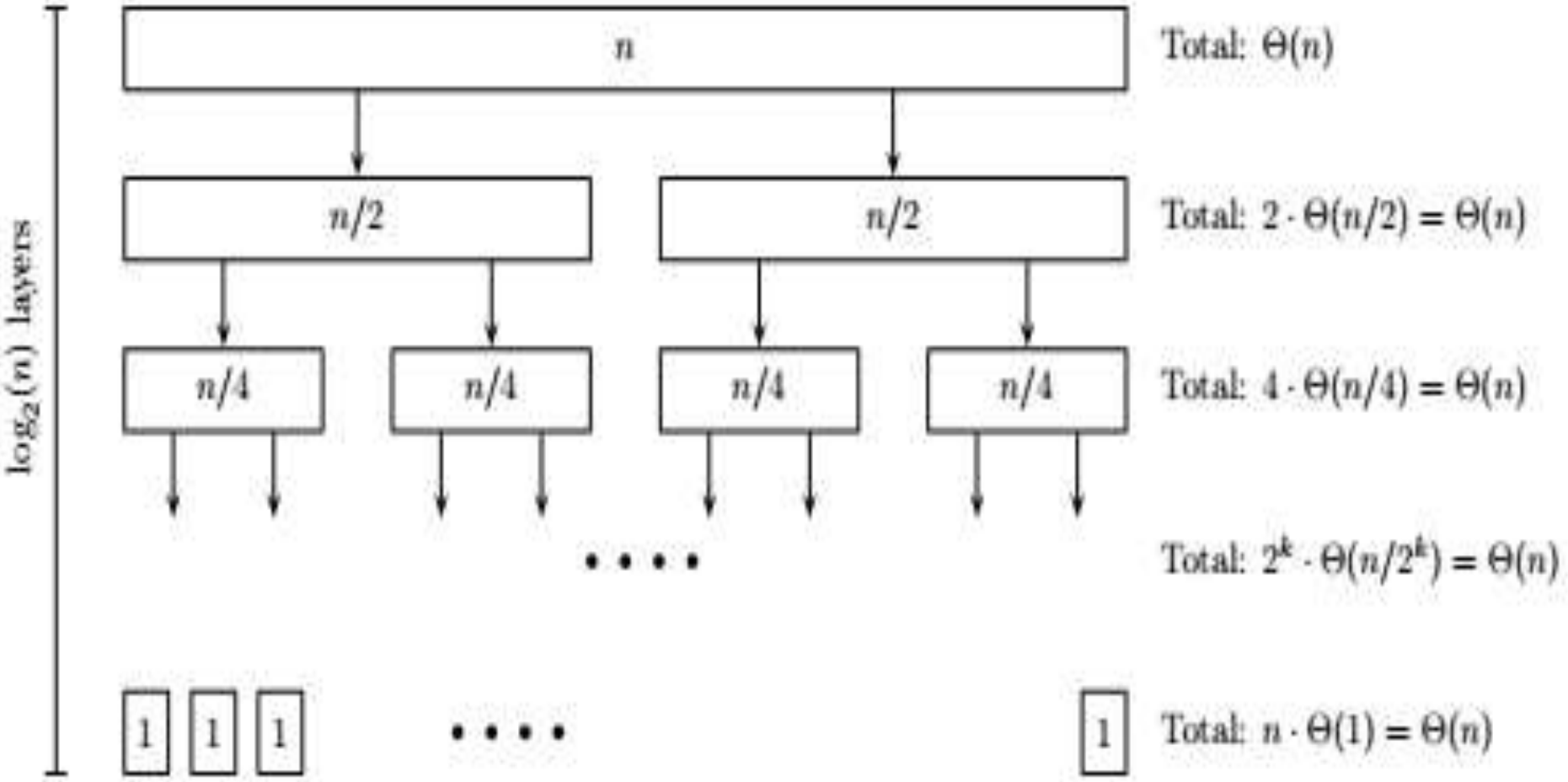
```
1    Algorithm Merge(low, mid, high)
2    // a[low : high] is a global array containing two sorted
3    // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4    // is to merge these two sets into a single set residing
5    // in a[low : high]. b[ ] is an auxiliary global array.
6    {
7        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   }
```


Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

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

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

$= 2\,[\mathbf{2\,T(n/4) + n/2}] + n$

$= 4\,T(n/4) + 2n$

$= 4\,[\mathbf{2\,T(n/8) + n/4}] + 2n$

$= 8\,T(n/8) + 3n$

$= 16\,T(n/16) + 4n$

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

Merge Sort : Analysis

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

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

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

$= 2^{\log_2 n} T(1) + (\log_2 n)\,n$

$= n + n \log_2 n$

$= O(n \log n)$

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

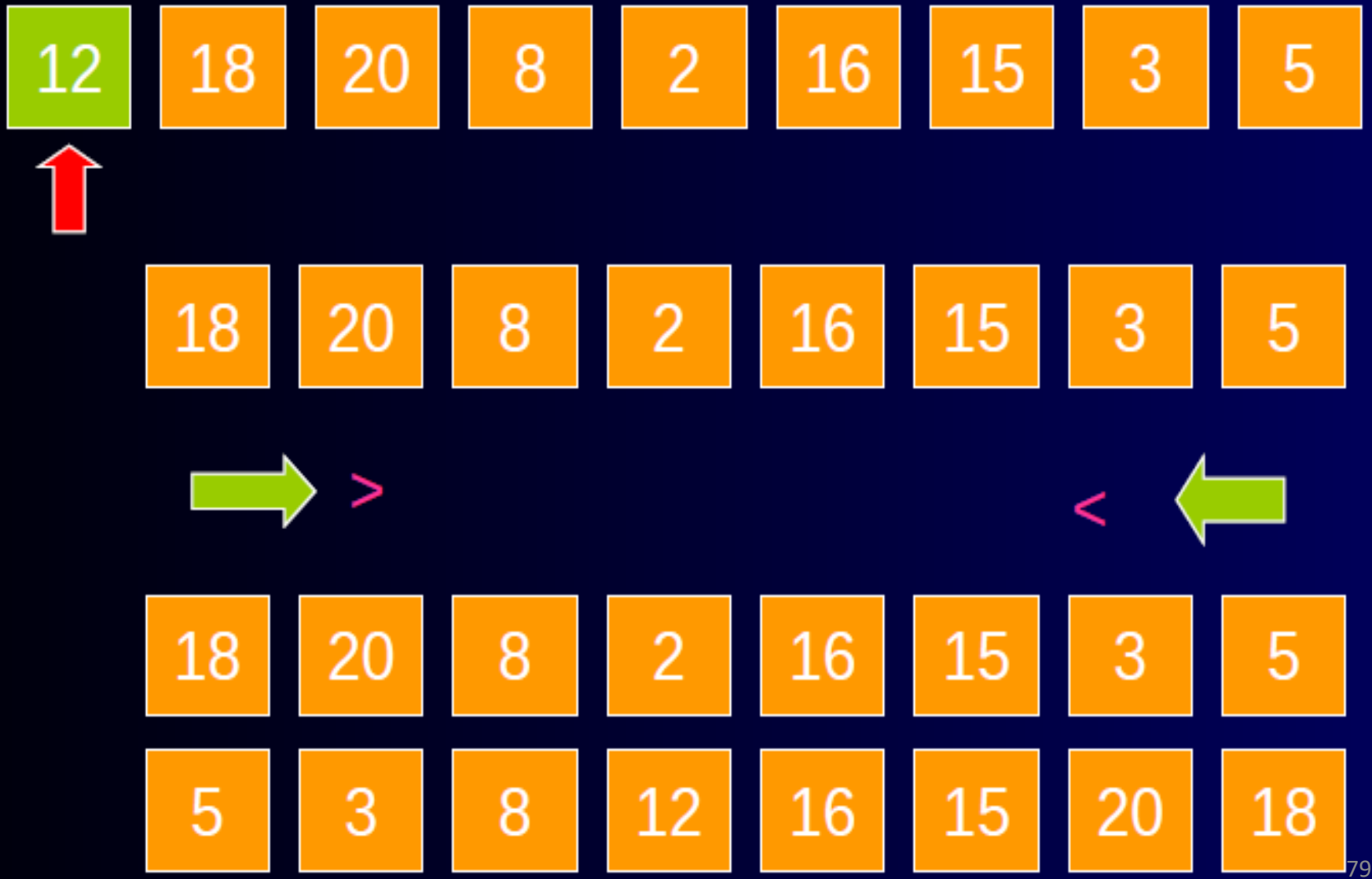
```
1   Algorithm InsertionSort(a, n)
2   // Sort the array a[1 : n] into nondecreasing order, n ≥ 1.
3   {
4       for j := 2 to n do
5       {
6           // a[1 : j − 1] is already sorted.
7           item := a[j]; i := j − 1;
8           while ((i ≥ 1) and (item < a[i])) do
9           {
10     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

Algorithm 3.9 Insertion sort

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] ≤ a[j] for all i between 1 and m
- and all j between m+1 to n
- for some m, 1≤m≤n.
- Thus the elements in a[1:m] and a[m+1:n] can be independently sorted.
- Complexity: O(n log n)

Quick Sort


12 18 20 8 2 16 15 3 5
18 20 8 2 16 15 3 5
>
<
18 20 8 2 16 15 3 5
5 3 8 12 16 15 20 18

```
1    Algorithm Partition(a, m, p)
2    // Within a[m], a[m + 1], ..., 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] ≤ t for m ≤ k < q, and a[k] ≥ t
6    // for q < k < p. q is returned. Set a[p] = ∞.
7    {
8        v := a[m]; i := m; j := p;
9        repeat
10       {
11           repeat
12               i := i + 1;
13           until (a[i] ≥ v);

14           repeat
15               j := j − 1;
16           until (a[j] ≤ v);

17           if (i < j) then Interchange(a, i, j);

18       } until (i > j);

19       a[m] := a[j]; a[j] := v; return j;
20   }

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    }
```

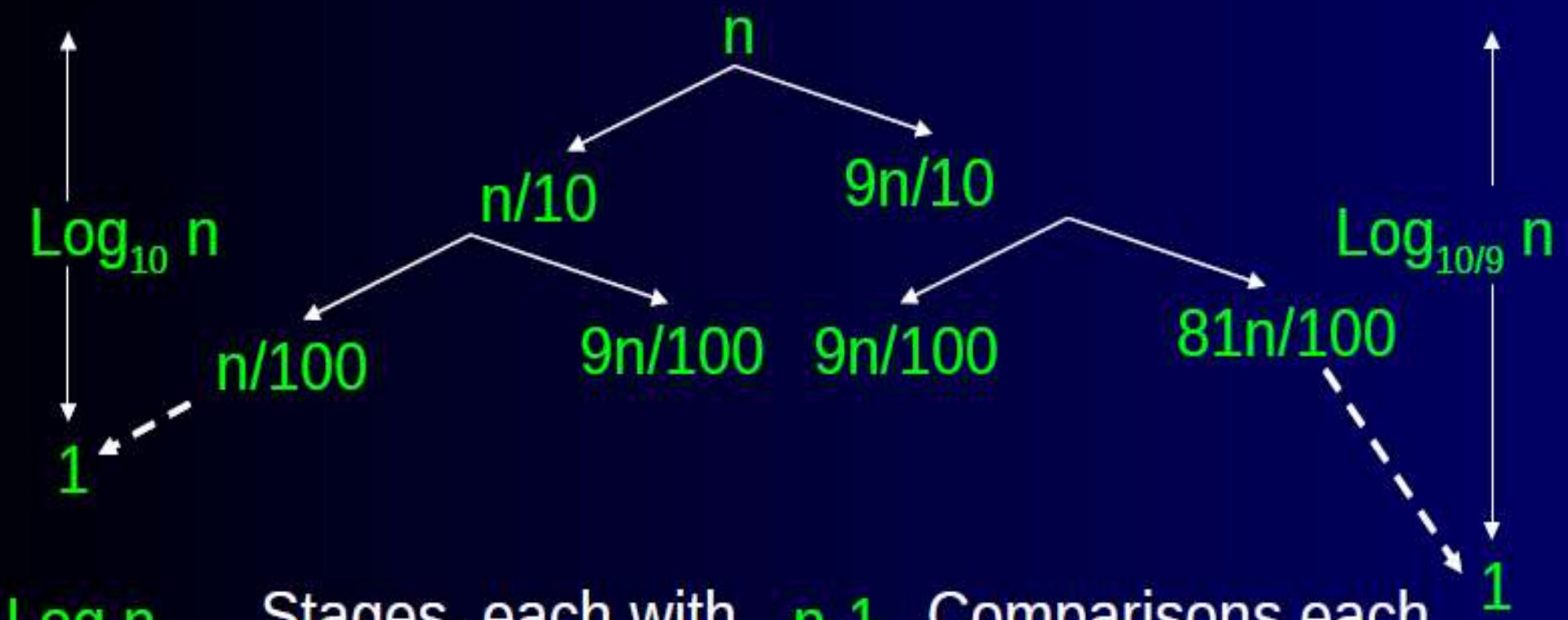
Algorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., 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 ≥ 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 := 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  }
```

Algorithm 3.13 Sorting by partitioning

Quick Sort: Average Case Analysis

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



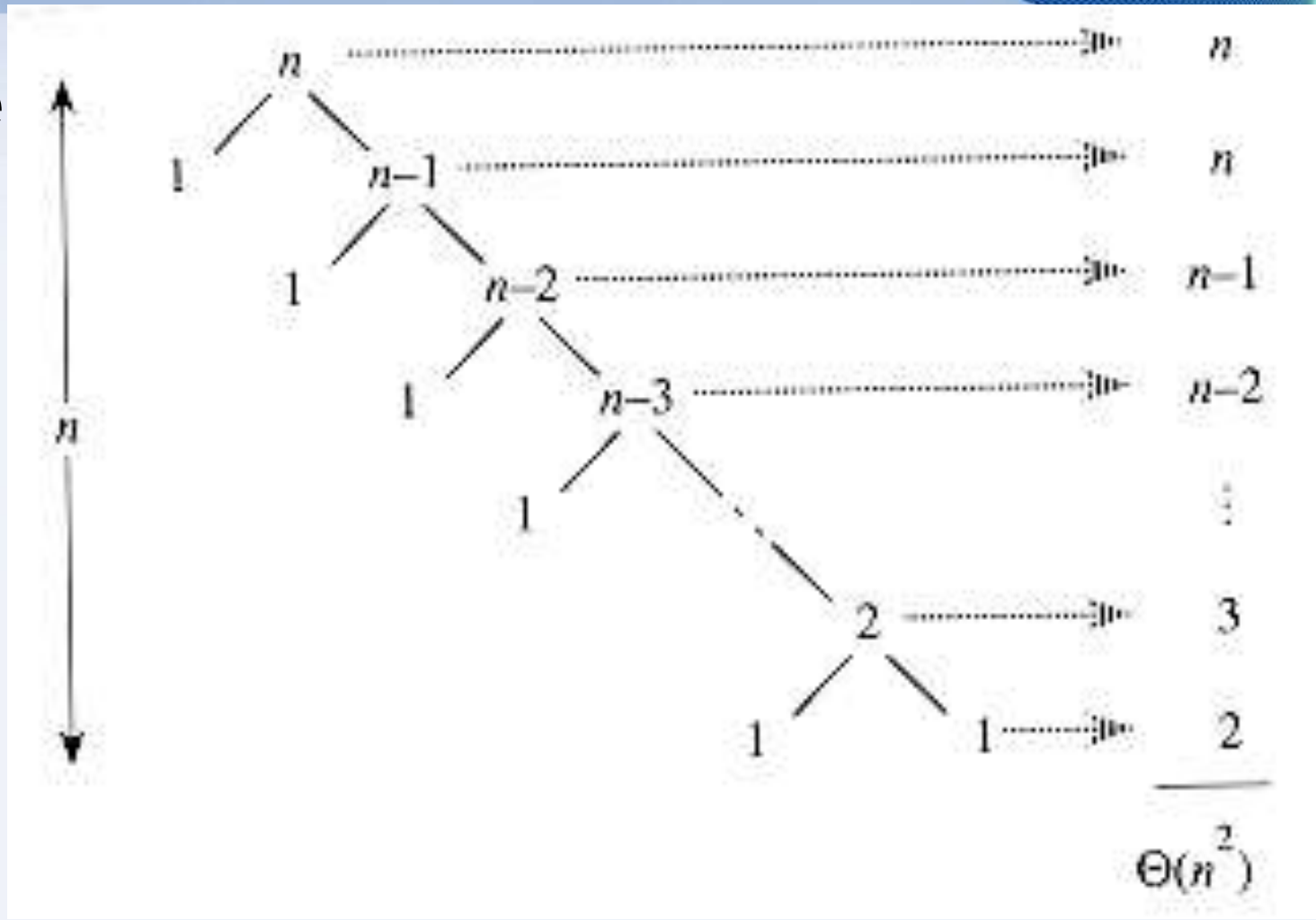
Log n Stages, each with n-1 Comparisons each

$T(n)=T(n/10)+T(9n/10)+\Theta(n)$

$= \Theta(n \log n)$

Quick Sort : Analysis

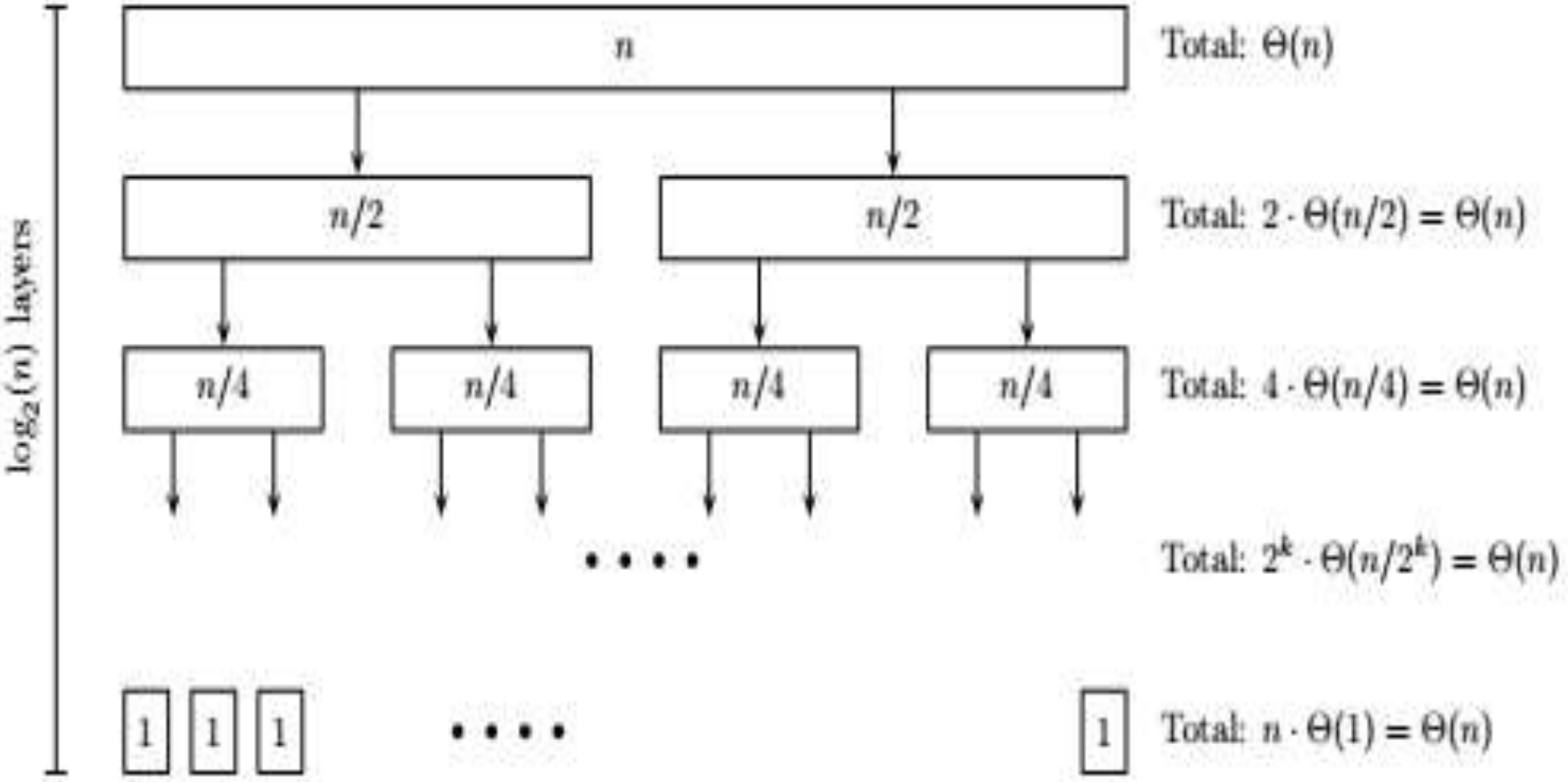
- Worst Case
-



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

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≤k≤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 kth-smallest element in a[1 : n] and places it
3   // in the kth position of a[ ]. The remaining elements are
4   // rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
5   // a[m] ≥ a[k] for k < m ≤ n.
6   {
7       low := 1; up := n
8
9       repeat
10      {
11          // Each time the loop is entered,
12          // 1 ≤ low ≤ k ≤ up ≤ n
13          j := Partition(a, low, up);
14                  // j is such that a[j] is the jth-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 kth-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 Pє 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 Θ(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 X1, with P1P2

- Determine point P of X1, that belong to convex hull of X1 and use it for partitioning
- P is obtained by computing area formed by P1PP2 for each P of X1 and picking the largest area
- Let the point be P3, the problem now can be subdivided by considering
  - All points of X1 that are to the left of P1P3
  - All points of X1 that are to the left of P3P2
  - Rest all the points are interior to the triangle P1P3P2 so they are not considered
- Complexity: Worst case O(n2), average O(nlog n)

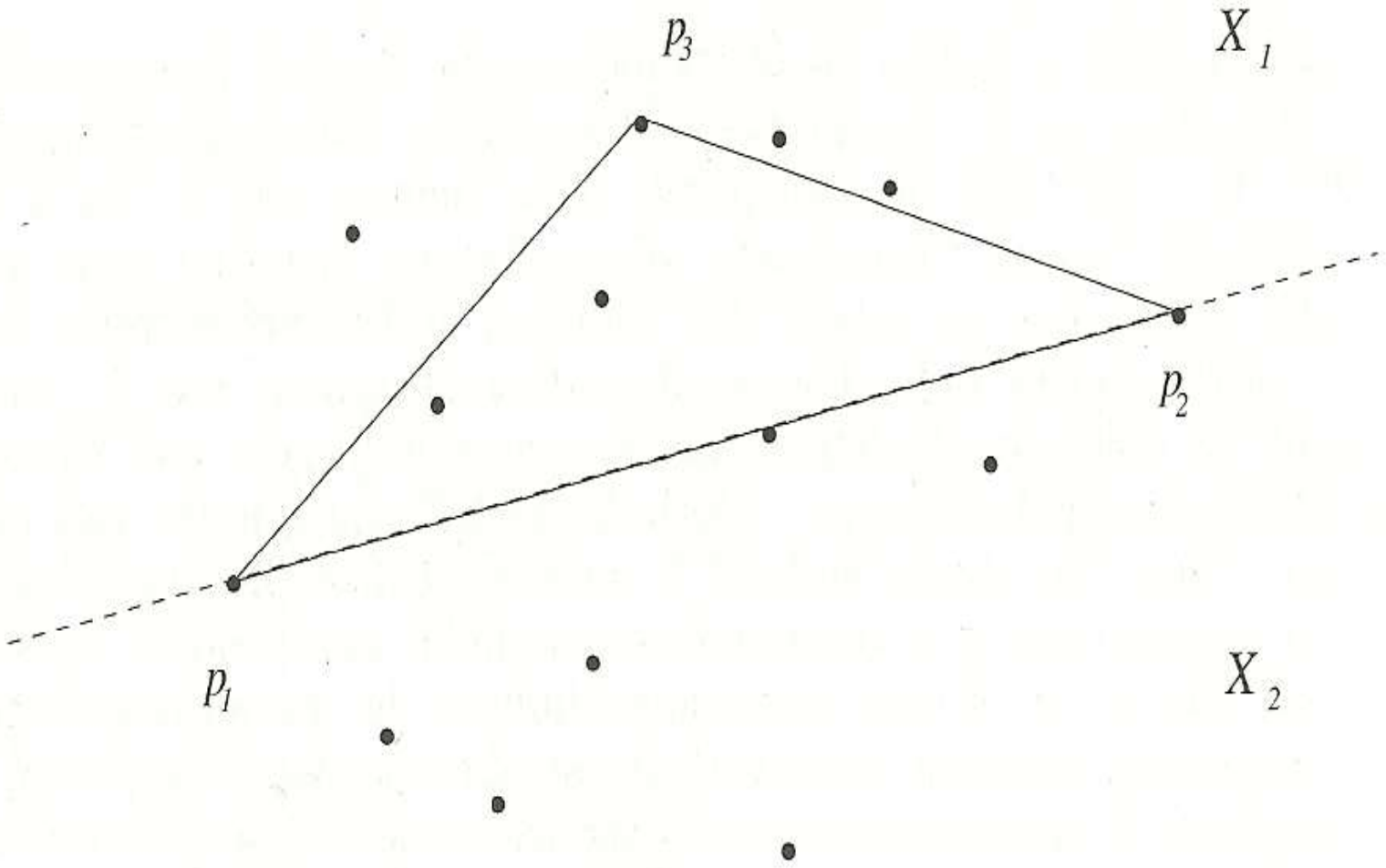
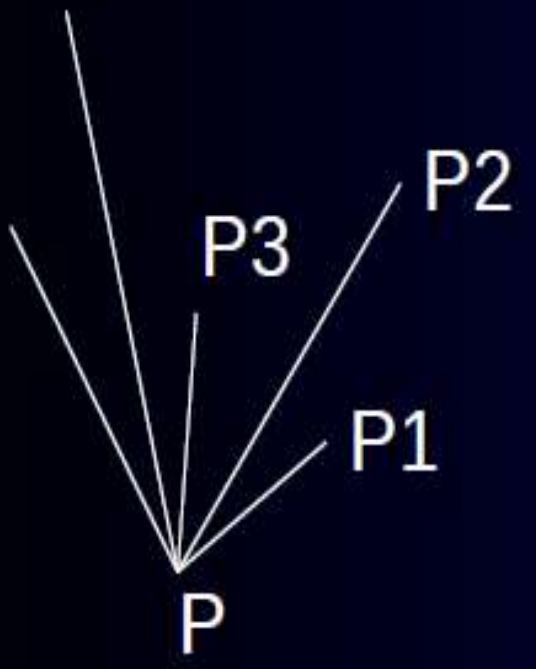


Figure 3.10 Identifying a point on the convex hull of X_1

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 P1,P2,P3…
- Take three successive points P1,P2,P3,
  - if it is left turn include the points in convex hull
  - If it is right turn, exclude middle point P2

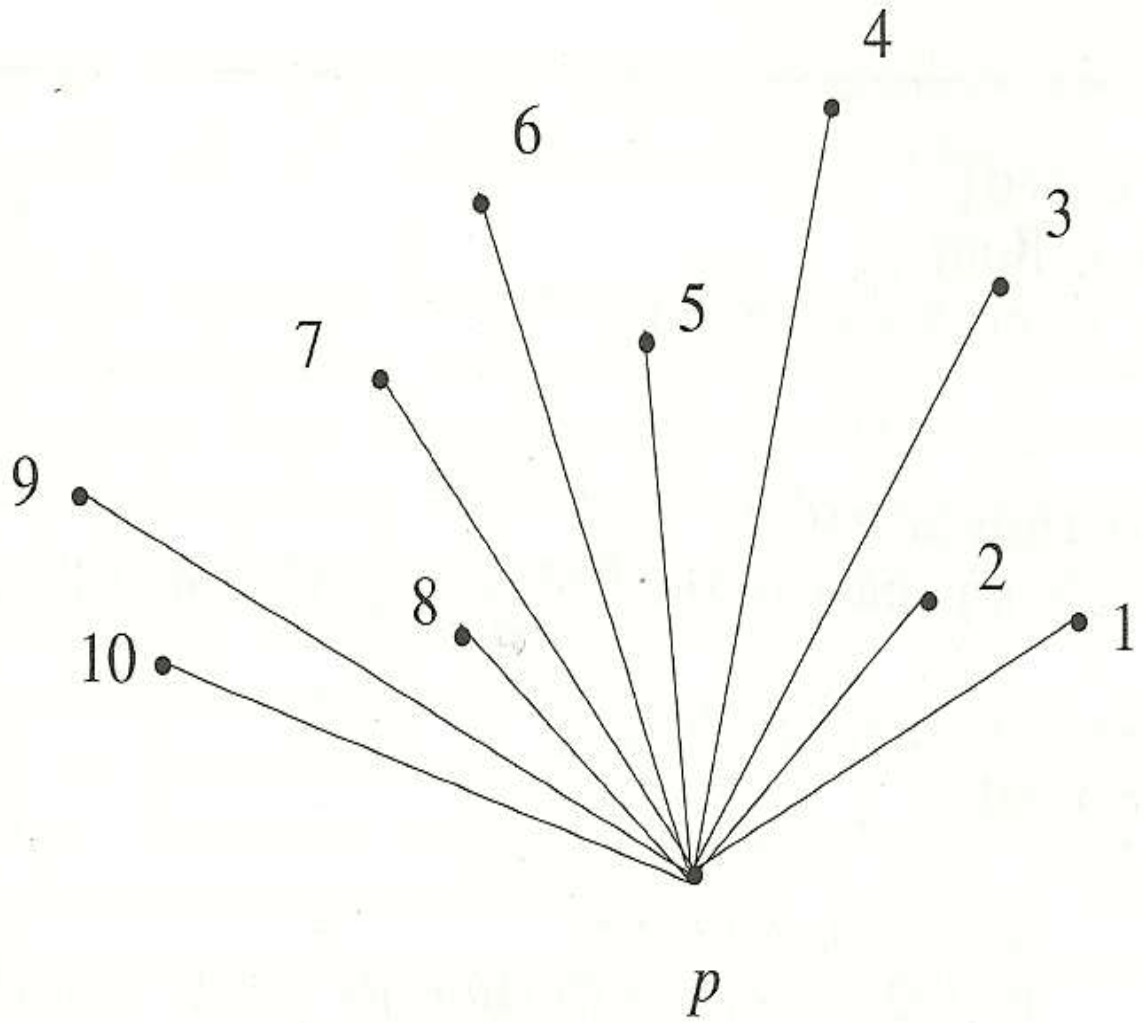


Figure 3.11 Graham's scan algorithm sorts the points first

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)

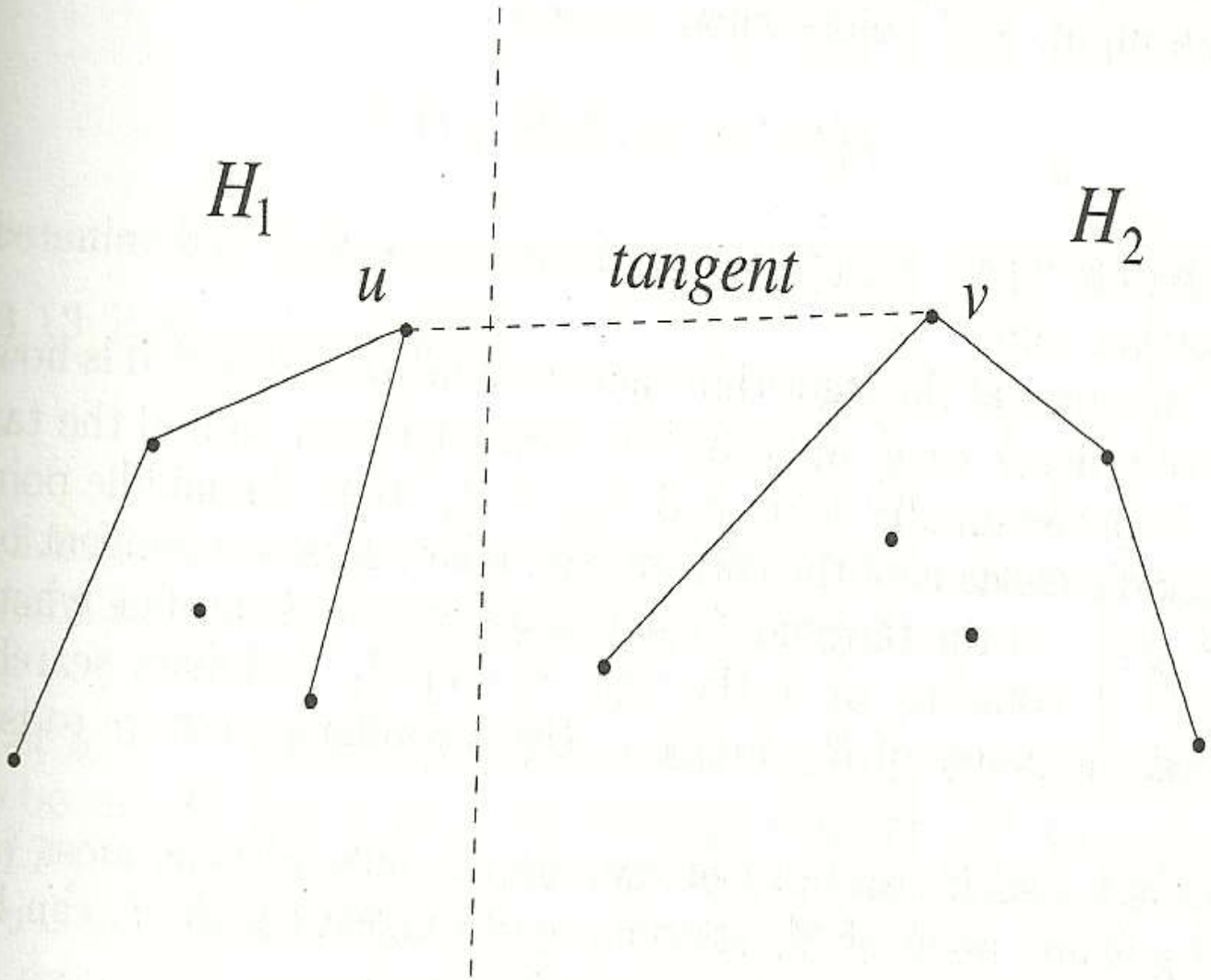


Figure 3.12 Divide and conquer to compute the convex hull

Thank You