Introduction to Design Analysis & Algorithms

Odd Sem (2022-23)

By:
Dr. Rimjhim Singh
Asst. Professor (Sr.Gr.)
Department of Computer Science & Engg.
Amrita Vishwa Vidyapeetham

Course Objective

- This course aims
  - To provide the fundamentals of algorithm design and analysis.
  - Specifically in terms of algorithm design techniques
  - Application of these design techniques for real-world problem solving.
  - Analysis of complexity and correctness of algorithms to provide the best solution to complex tasks in terms of simpler tasks.

Course Outcomes

- CO1: Evaluate the correctness and analyze complexity of algorithms.
- CO2: Understand and implement various algorithmic design techniques and solve classical problems
- CO3: Design solutions for real world problems by identifying, applying and implementing appropriate design techniques.
- CO4: Design solutions for real world problem by mapping to classical problems
- CO5: Analyze the impact of various implementation choices on the algorithm complexity

Syllabus

- **Unit 1** Introduction and Review- Algorithms vs. programs. Flow charts and pseudo code, Rate of growth of functions. Review of Asymptotic notation: motivation and types of notations. Recurrence relations and methods to solve them: Recursion tree, substitution, Master Method. Review of Sorting: Bubble –Insertion – Selection – Bucket – Heap, Comparison of sorting algorithms, Applications. Graph Algorithms – Graph Traversal: Applications of BFS: distance, connectivity and connected components and cycles in undirected graphs. Applications of DFS: Topological sort, cycles in directed graphs, Biconnected Components and Strong Connectivity. Path algorithms: Shortest path algorithms (along with analysis) SSSP: Bellman Ford. APSP: Floyd Warshall's. Review of Minimum Spanning Tree (with analysis and applications).

- **Unit 2** Divide and Conquer: Merge sort and Binary search type strategies, Pivot based strategies – Long integer multiplication – Maximum sub array sum - Closest Pair problem etc as examples. Greedy Algorithm - Introduction to the method, Fractional Knapsack problem, Task Scheduling Problem, Huffman coding etc as examples. Dynamic Programming: Introduction to the method, Fibonacci numbers, 0-1 Knapsack problem, Matrix chain multiplication problem, Longest Common Subsequence, and other problems including problems incorporating combinatorics as examples.

- **Unit 3** Backtracking, Branch and Bound 0-1 Knapsack, N- Queen problem, subset sum as some examples. String Matching: Rabin Karp, Boyer Moore, KMP. Network Flow and Matching: Flow Algorithms Maximum Flow – Cuts Maximum Bipartite Matching. Introduction to NP class: Definitions P, NP, NP complete, NP hard, Examples of P and NP.

Textbooks

- TEXTBOOK:

  Michael T Goodrich and Roberto Tamassia, “Algorithm Design Foundations - Analysis and Internet Examples”, John Wiley and Sons, 2007.

- REFERENCES:

  1. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein, “Introduction to Algorithms”, Third Edition, Prentice Hall of India Private Limited, 2009

  2. Dasgupta S, Papadimitriou C and Vazirani U, “Algorithms”, Tata McGraw-Hill, 2009.

  3. Jon Kleinberg, Eva Tardos. Algorithm Design. First Edition, Pearson Education India; 2013.

Evaluation Pattern

- Evaluation: 65:35

- Periodical -1
- Periodical – 2
- Continuous Theory

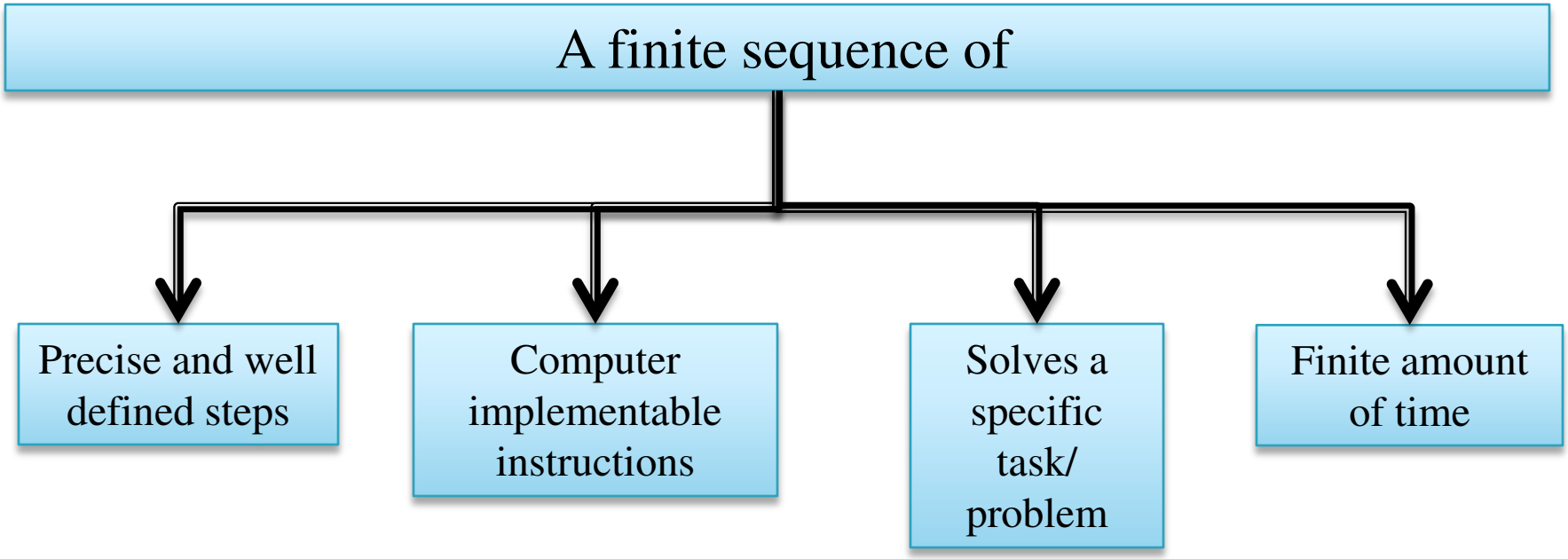
- Lab tests : 2 (Tentative)
- Case Study

- End Semester: 35

Problem Solving and Analysis

- Identify problems in real world solvable by computers
- Understand the problem
  - Understand the inputs
  - Output requirements
  - Constraints under which the problem must operate
- Identify potential solutions
- Select best solution
  - Fastest
  - Most accurate

What is an Algorithm???



Definition:

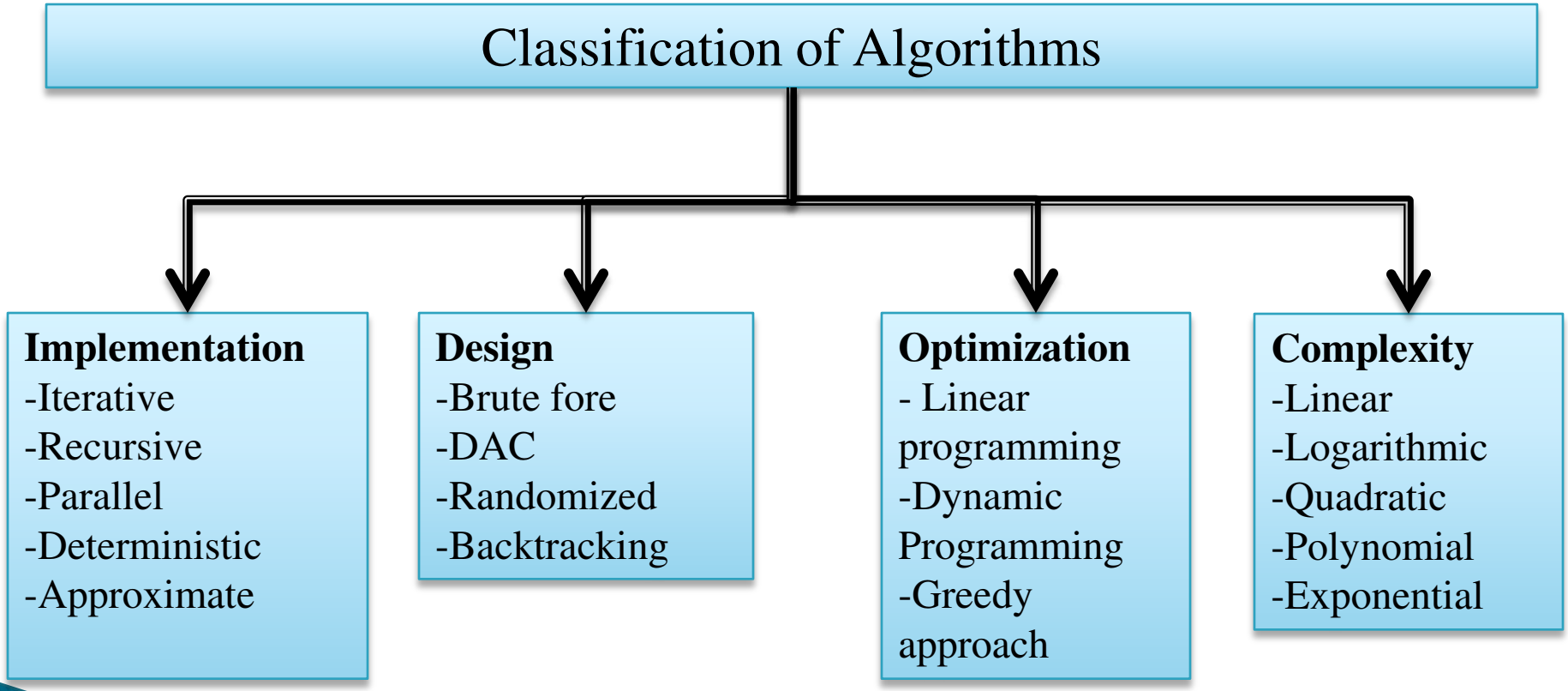
"A *finite* sequence of *unambiguous*, *executable* steps or instructions, which, if followed would ultimately *terminate* and give the solution of the problem".

Algorithm vs Program

- **Algorithm**
  - Abstract description of a process
  - Just like a method to solve the problem
  - Does not deal with machine specific details

- **Program**
  - Implementations of algorithms
  - For specific system and platform

Algorithms


Classification of Algorithms
Implementation
-Iterative
-Recursive
-Parallel
-Deterministic
-Approximate
Design
-Brute fore
-DAC
-Randomized
-Backtracking
Optimization
- Linear programming
-Dynamic Programming
-Greedy approach
Complexity
-Linear
-Logarithmic
-Quadratic
-Polynomial
-Exponential

Algorithms : Writing Psuedocodes

- High level description
- Intended for humans
- More structured than prose
- Hides implementation / design details
- Less structured than programs

Expression	· '← ' for assignment and '=' for equality · Mathematical expressions are allowed
Method declaration	· Algorithm name (param1, param2,...) · Defines new method 'name' with arguments
Decision structure	· If condition then Action1 else Action 2 · Use indentations
While/ for/ repeat loops	· While condition do Action-1 · Repeat Action-1 until condition
Array indexing	· A[i] indexes to the ith element of array A
Method calls	· Object.method(param1, param2,...) · Object is optional
Method returns	· Return value

Algorithms vs Pseudocodes

Example: Maximum element in an Array

```
Algorithm arrayMax(A, n)
    Input array A of n integers
    Output maximum element of A

    currentMax ← A[0]
    for i ← 1 to n - 1 do
        if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
```

Pseudocode

Algorithm arrayMax
Input: N number of integers
Output: Maximum number
Step-1: Create a local variable **max** to store the maximum among the list
Step-2: Initialize max with the first element initially, to start the comparison.
Step-3: Then traverse the given array from second element till end, and for each element:
Step-4: Compare the current element with max
Step-5: If the current element is greater than max, then replace the value of max with the current element.
Step-6: At the end, return and print the value of the largest element of array stored in max

Algorithm

Characteristics of an Algorithm

- **Input:** Zero / more quantities are externally supplied.
- **Output:** At least one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.
- **Efficiency:** Every instruction must be very basic and runs in short time.

```
Algorithm arrayMax(A, n)
    Input array A of n integers
    Output maximum element of A

    currentMax ← A[0]
    for i ← 1 to n - 1 do
        if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
```

Analyzing Algorithms

Algorithms can be analyzed based on the following parameters:

- Correctness
- Amount of Work done
- Space used
- Simplicity, clarity
- Optimality

Correctness

Understand what correctness means

- Define the **characteristics of the input** an algorithm is expected to work on
- The **results** that each input must produce
- Prove the statement about the **relationship between input and output**

- Prove Correctness of algorithm

Correctness

- Specify the task to be performed by the algorithm
- Specification consists of:
  - *Name* of algorithm and list of its arguments
  - *Initial condition* (it specifies what is correct input data to the problem)
  - *Final condition* (it specifies what is the desired result of the algorithm)

- An algorithm is called ***totally correct*** for the given specification if and only if for any correct input data it:
  - stops
  - returns correct output

Example

```
var maxNum = -1;
for (var i = 0; i < numbers.length; i++) {
  if (numbers[i] > maxNum) {
    maxNum = numbers[i];
  }
}
```

Numbers= {13, 4 12, 27}
Is it working??????

Will it stop????

Numbers= {-13, -4, -12, -27}
Is it working??????

Is the Result correct??

Example

Change this to maxNum=numbers[0]

```
var maxNum = -1;
for (var i = 0; i < numbers.length; i++) {
  if (numbers[i] > maxNum) {
    maxNum = numbers[i];
  }
}
```

Numbers= {13, 4 12, 27}
Is it working??????

Will it stop????

Numbers= {-13, -4, -12, -27}
Is it working??????

Is the Result correct??

Example

- **What is the largest integer?**

  INPUT: All the integers { … -2, -1, 0, 1, 2, … }

  OUTPUT: The largest integer

  Algorithm:

  - Arrange all the integers in a list in decreasing order;
  - MAX = first number in the list;
  - Print out MAX;

Will it stop????

Is the Result correct??

Analyzing Algorithms

Analysis can be done for :

Resources required.

Memory / space required

Runtime requirements

• An efficient algorithm requires **less** time, space, resources and computations

• Analyzing **runtime requirements** are of utmost importance

• Obvious way to **measure the efficiency** of an algorithm is **to run it** and measure how much **processor time** is needed

• Is it correct ??????

Analyzing Algorithms

Factors:

- Hardware
- Operating System
- Compiler
- Size of input
- Nature of Input

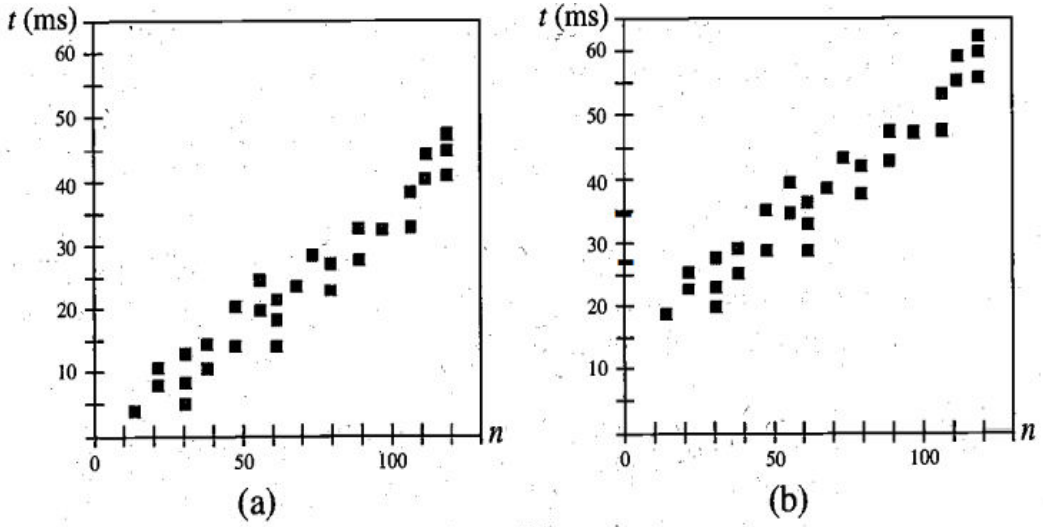


Figure 1.1: Results of an experimental study on the running time of an algorithm. A dot with coordinates (n,t) indicates that on an input of size n, the running time of the algorithm is t milliseconds (ms). (a) The algorithm executed on a fast computer; (b) the algorithm executed on a slow computer.

- Solution

Theoretical approximation is preferred

- It characterizes the execution time of an algorithm independently from the **machine, the language and compiler**.
- **Analyzing variation** in runtime requirements with changing **size of input data**
- Comparisons of different algorithms.

Analyzing Algorithms

- Runtime complexity of algorithm
  - Varies with input size
  - Nature of input data

- Model of Computation:
  - Mathematical model
  - Asymptotic Notations

Analyzing Algorithms: Runtime Complexity

- It is of three types:

(a) Best-case:

- Describes an algorithm's behavior under optimal conditions.
- Searching an element present at first position

(b) Average-case:

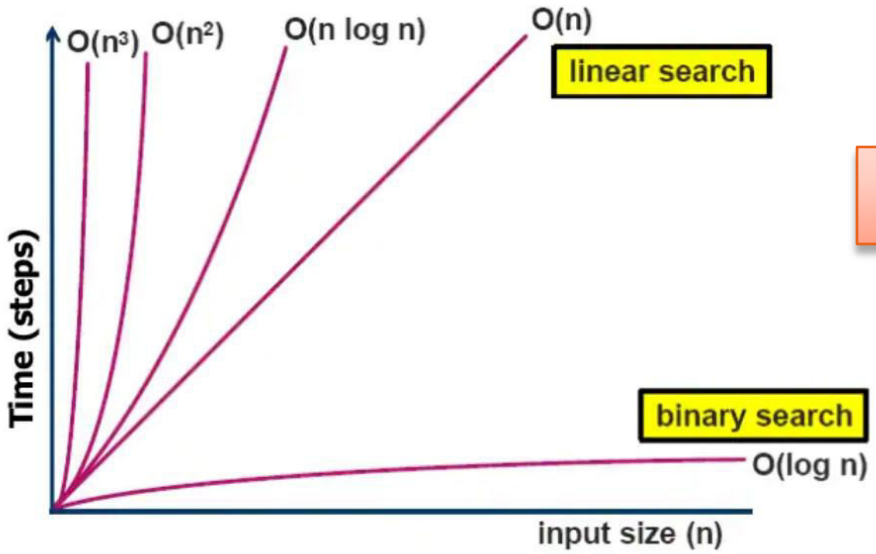
- Provides the cost of algorithm for average case of data.
- Analyzing average data and cost is difficult.

(c) Worst-case:

- Provides upper-bound or maximum cost of algorithm.
- Safe analysis of over-all behaviour.

Mostly worst-case analysis of the algorithms is computed for comparisons.

Analyzing Algorithms: Runtime Complexity



Difference in complexity matters!!!!!

N	$\log_2 N$	5N	$N \log_2 N$	N^2	2^N
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^9$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$

Analyzing Runtime Complexity

- ***Random Access Machine Model:***
- Counts the primitive operations / computations
- Assumes that CPU performs single primitive operation in 1 unit time
- Primitive operations are identified and counted to analyze cost
  - –Assigning a value to a variable
  - –Performing an arithmetic operation
  - –Calling a method
  - –Comparing two numbers
  - –Indexing into an array
  - –Following an object reference
  - –Returning from a method

Analyzing Time Complexity

- ***Counting primitive operations:***

Algorithm FindMax(S, n)

Input : An array S storing n numbers,n>=1

Output: Max Element in S

S1: curMax ← A[0]

S2: i ← 1

S3: while i <= (n-1) do

S4: if curMax < A[i] then

S5: curMax ← A[i]

S6: i ← i+1;

S7: return curmax

Complexity ?????

Analyzing Time Complexity

- ***Counting primitive operations:***

Algorithm FindMax(S, n)

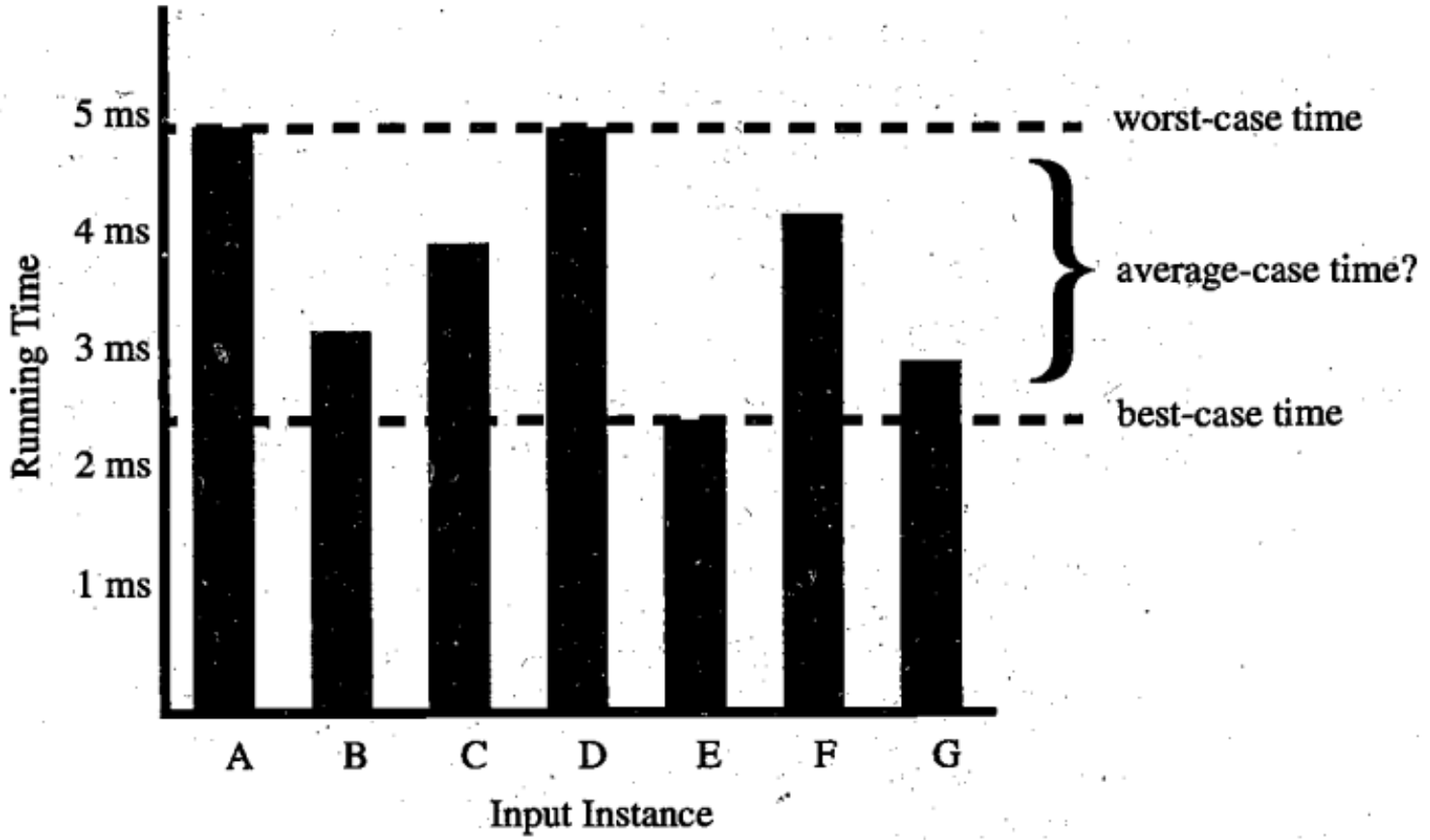
Input : An array S storing n numbers,n>=1

Output: Max Element in S

S1:	curMax ← A[0]	2 op (ind & assign)
S2:	i ← 1	1 op (assign)
S3:	while i<= n-1 do	2n op (n comp+n sub)
S4:	if curMax < A[i] then	2(n-1) op (n-1 [] & n-1 <)
S5:	curMax ← A[i]	0 to 2(n-1) op
S6:	i ← i+1;	2(n-1) op (+ & ->)
S7:	return curmax	1 op

Complexity between 6n and 8n-2

Analyzing Time Complexity



Analyzing Time Complexity

Algorithm recursiveMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

if $n = 1$ **then**

  return $A[0]$

return max{recursiveMax$(A, n-1), A[n-1]$}

Algorithm 1.4: Algorithm recursiveMax.

Think how to analyze ?????

Practice problems

```
def main():
    a=5
    b=6
    c=10
    for i in range(n):
        for j in range(n):
            x = i * i
            y = j * j
            z = i * j
    for k in range(n):
        w = a*k + 45
        v = b*b
    d = 33
main()
```

Practice problems

What is the runtime of g(n)?

```
void g(int n) {
  for (int i = 0; i < n; ++i) f();
}
```

$$\text{Runtime}(g(n)) \approx n \cdot \text{Runtime}(f())$$

```
void g(int n) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) f();
}
```

$$\text{Runtime}(g(n)) \approx n^2 \cdot \text{Runtime}(f())$$

Practice problems

What is the runtime of g(n)?

```
void g(int n) {
  for (int i = 0; i < n; ++i) f();
}
```

Constant factor can be dropped
n

$$\text{Runtime}(g(n)) \approx n \cdot \text{Runtime}(f())$$

```
void g(int n) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) f();
}
```

Constant factor can be dropped
n^2

$$\text{Runtime}(g(n)) \approx n^2 \cdot \text{Runtime}(f())$$

Practice problems

What is the runtime of g(n)?

```
void g(int n) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j <= i; ++j) f();
}
```

$$\begin{aligned}\text{Runtime}(g(n)) &\approx (1+2+3+\cdots+n)\cdot\text{Runtime}(f()) \\ &\approx \frac{n^2+n}{2}\cdot\text{Runtime}(f())\end{aligned}$$

Practice problems

```
int a = 0, b = 0;
for (i = 0; i < N; i++)
{
    a = a + rand();
}
for (j = 0; j < M; j++)
{
    b = b + rand();
}
```

Time complexity=??
Space Complexity=??

Practice problems

```
function isPrime(n) {

for (for i = 2; i <= sqrt(n); ++i)
        {

        if (n % i === 0)
                {
                return false;
                }
        }
return true;
}
```

Time complexity=

Practice problems

```
int a = 0, i = N;
while (i > 0)
    {
    a += i;
    i /= 2;
    }
```

Time complexity=

Practice problems

```
for(var i=0;i<n;i++)
{
    i*=k
}
```

Time complexity=???

Practice problems

```
let a = 0, b = 0;
for (let i = 0; i < n; ++i)
   {
          for (let j = 0; j < n; ++j) {
                  a = a + j;
   }
          }
for (let k = 0; k < n; ++k)
          {
          b = b + k;
          }
```

Time complexity=????

Practice problems

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++)
{
    for (j = 2; j <= n; j = j * 2)
    {
        k = k + n / 2;
    }
}
```

Time complexity=????

Practice problems

```
int count = 0;
 for (int i = N; i > 0; i /= 2)
   {
      for (int j = 0; j < i; j++)
         {
 count += 1;
         }
 }
```

Time complexity=???

Practice problems

```
void counter(int n)
{
    for(int i = 1 ; i < n ; i++)
    {
        for(int j = 1 ; j<n ; j += i )
            {
               cout<<i<<" "<<j;
            }
        cout<<endl;
 }
}
```

Time complexity=????

Analyzing Time Complexity

- Important factor to be considered when estimating complexity
- When experimental setup (hardware/software) changes
  - Running time/memory is affected by a constant factor
  - 2n or 3n or 100n is still linear
  - Growth rate of the running time/memory is not affected
- Growth rates of functions
  - Linear
  - Quadratic
  - Exponential
  - Logarithmic
  - Polynomial

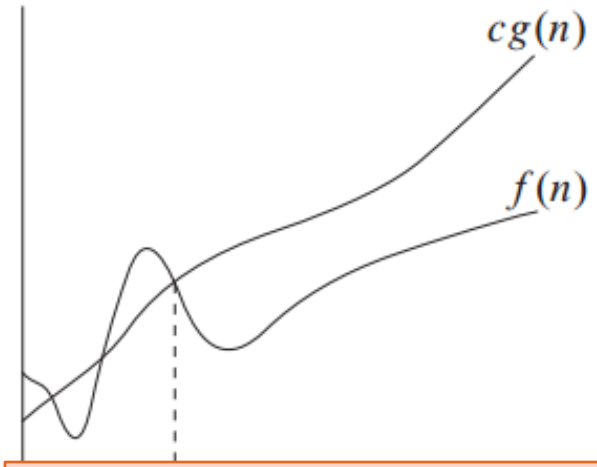
Asymptotic Complexity Analysis: Big O

- A method to characterize the execution time of an algorithm
- Provides the upper bound or worst-case complexity analysis
- Adding two square matrices is Quadratic
- Searching in an array (linear search) is Linear
- Searching in an array (binary search) is Logarithmic
- Multiplying two square matrices is Cubic

- *The O notation only uses the dominating terms of the execution time. Constants are disregarded.*

Asymptotic Complexity Analysis: Big O

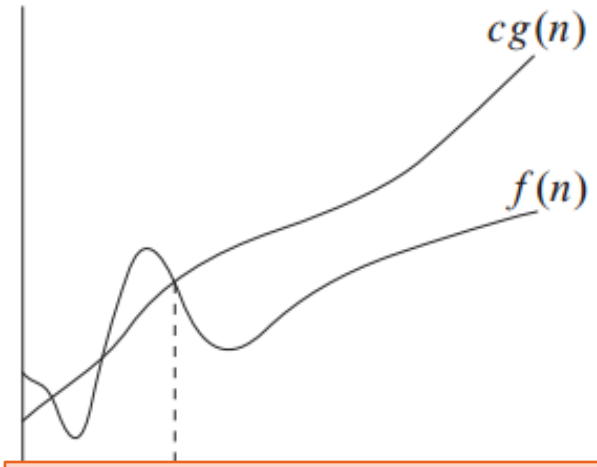
- Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$.
- This definition is often pronounced as "$f(n)$ is *big-Oh* of $g(n)$" or "$f(n)$ is *order* $g(n)$."
- Example: 7n − 2 is O(n).



Mostly analysis is done for larger 'n'

Asymptotic Complexity Analysis: Big O

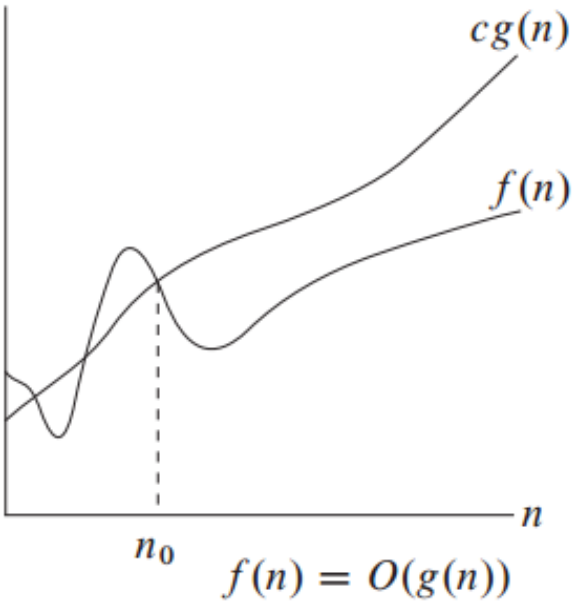
- Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$.
- This definition is often pronounced as "$f(n)$ is *big-Oh* of $g(n)$" or "$f(n)$ is *order* $g(n)$."
- Example: 7n − 2 is O(n).



Mostly analysis is done for larger 'n'

Asymptotic Complexity Analysis: Big O

- Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$.
- This definition is often pronounced as "$f(n)$ is *big-Oh* of $g(n)$" or "$f(n)$ is *order* $g(n)$."
- Example: 7n − 2 is O(n).



n	$\log n$	$\log^2 n$	$\sqrt{n}$	$n \log n$	n^2	n^3	2^n
4	2	4	2	8	16	64	16
16	4	16	4	64	256	4,096	65,536
64	6	36	8	384	4,096	262,144	1.84×10^{19}
256	8	64	16	2,048	65,536	16,777,216	1.15×10^{77}
1,024	10	100	32	10,240	1,048,576	1.07×10^{9}	1.79×10^{308}
4,096	12	144	64	49,152	16,777,216	6.87×10^{10}	10^{1233}
16,384	14	196	128	229,376	268,435,456	4.4×10^{12}	10^{4932}
65,536	16	256	256	1,048,576	4.29×10^{9}	2.81×10^{14}	10^{19728}
262,144	18	324	512	4,718,592	6.87×10^{10}	1.8×10^{16}	10^{78913}

Asymptotic Complexity Analysis: Big O

Example

•Show 7n-2 is O(n)

f(n) => O(g(n)) if

f(n) <= c•g(n) when n> n_0

Here f(n)= 7n-2

g(n)=n

–need c > 0 and n_0 >= 1 such that

7n-2 <= cn for n >= n_0

–this is true for c = 7 and n_0 = 1

Multiple values of C and n_0 are possible

Asymptotic Complexity Analysis: Big O

```
Algorithm FindMax(S, n)
    Input : An array S storing n numbers,n>=1
    Output: Max Element in S
S1:     curMax ← A[0]
S2:     i ← 1
S3:     while i<= n-1 do
S4:             if curMax < A[i] then
S5:                     curMax ← A[i]
S6:             i ← i+1;
S7:     return curmax
```

Complexity between 6n and 8n-2

Prove T(n) = 8n-2 is O(n)

show T(n) = 8n-2 is O(n)
c=8 n0=1

Asymptotic Complexity Analysis: Big O

f(n) => O(g(n)) if

f(n) <= c•g(n) when n> n_0

Example

• Show $3n^3 + 20n^2 + 5$ is $O(n^3)$

Sol

–need c > 0 and n_0 >= 1 such that

$3n^3 + 20n^2 + 5 <= cn^3$ for n >= n_0

–this is true for c = 4 and n_0 = 21

Another way to prove

$3n^3 < (3n^3 + 20n^2 + 5) <= (28)\ n^3$

C=28 ; Constant term.
Hence proved

Asymptotic Complexity Analysis: Big O

Example

- Prove n^2 is not O(n)

Example

- Show 3 log n + log log n is O(log n)

Asymptotic Complexity Analysis: Big O

- The big-Oh notation gives an **upper bound** on the growth rate of a function

- The statement "f(n) is O(g(n))" means that the **growth rate** of f(n) is no more than the growth rate of g(n)

- We are guaranteeing that f(n) grows at a **rate no faster** than g(n)
- Both can grow at the same rate
  - Though 1000n is larger than n^2 , n^2 grows at a faster rate
  - n^2 will be larger function after n = 1000
  - Hence 1000n = O(n)

- The big-Oh notation can be used to **rank functions** according to their growth rate

Asymptotic Complexity Analysis: Big O

Growth rate analysis of different functions

n	$\log n$	$\log^2 n$	$\sqrt{n}$	$n \log n$	n^2	n^3	2^n
4	2	4	2	8	16	64	16
16	4	16	4	64	256	4,096	65,536
64	6	36	8	384	4,096	262,144	1.84×10^{19}
256	8	64	16	2,048	65,536	16,777,216	1.15×10^{77}
1,024	10	100	32	10,240	1,048,576	1.07×10^{9}	1.79×10^{308}
4,096	12	144	64	49,152	16,777,216	6.87×10^{10}	10^{1233}
16,384	14	196	128	229,376	268,435,456	4.4×10^{12}	10^{4932}
65,536	16	256	256	1,048,576	4.29×10^{9}	2.81×10^{14}	10^{19728}
262,144	18	324	512	4,718,592	6.87×10^{10}	1.8×10^{16}	10^{78913}

Asymptotic Complexity Analysis: Big O

4. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?

Options:

a. X will always be a better choice for small inputs

b. X will always be a better choice for large inputs

c. Y will always be a better choice for small inputs

d. X will always be a better choice for all inputs

Asymptotic Complexity Analysis: Big Oh

- If is f(n) a polynomial of degree d, then f(n) is $O(n^d)$, i.e.,

- Drop lower-order terms
- Drop constant factors

- Use the smallest possible class of functions to represent in big Oh

  - "2n is O(n)"
  - Use the simplest expression of the class
  - "3n+ 5 is O(n)" instead of "3n + 5 is O(3n)"

Asymptotic Complexity Analysis: Big Oh

Algorithm **A** and **B** have a worst-case running time complexity, respectively. Therefore, algorithm B always runs faster than the algorithm A.

TRUE / FALSE

Problem solving

- Show that 8n+5 is O(n)
- Show that $20n^3 + 10n^2 + 5$ is $O(n^3)$
- Show that 3logn+2 is O(logn).

Arrange the following functions in order of their complexities.

$5n^2$, $\log_3 n$, 20n, $\log_2 n$, $n^{2/3}$, 2n, 2^{100}, n^{1000}

Plot graph for the fucntions and verify your results.

Practice problems

Calculate problems for the following the following

$T(n)$
$5n^3 + 200n^2 + 15$
$3n^2 + 2^{300}$
$5\log_2 n + 15\ln n$
$2\log n^3$
$4n + \log n$
2^{64}
$\log n^{10} + 2\sqrt{n}$
$2^n + n^{1000}$

Properties of BIG(O)

- Let $d(n)$, $e(n)$, $f(n)$, and $g(n)$ be functions mapping nonnegative integers to non-negative reals.

1. If $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.
2. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)+e(n)$ is $O(f(n)+g(n))$.
3. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)e(n)$ is $O(f(n)g(n))$.
4. If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.
5. If $f(n)$ is a polynomial of degree d (that is, $f(n) = a_0 + a_1 n + \cdots + a_d n^d$), then $f(n)$ is $O(n^d)$.
6. n^x is $O(a^n)$ for any fixed $x > 0$ and $a > 1$.
7. $\log n^x$ is $O(\log n)$ for any fixed $x > 0$.
8. $\log^x n$ is $O(n^y)$ for any fixed constants $x > 0$ and $y > 0$.

Properties of BIG(O): Example

Show $2n^3 + 4n^2\log n$ is $O(n^3)$

$\log n$ is $O(n)$ (rule 8)

$4n^2\log n$ is $O(4n^3)$ (rule 3)

$2n^3 + 4n^2\log n$ is $O(2n^3 + 4n^3)$ (rule 2)

$2n^3 + 4n^3$ is $O(n^3)$

$2n^3 + 4n^2\log n$ is $O(n^3)$ (rule 4)

Problem Solving

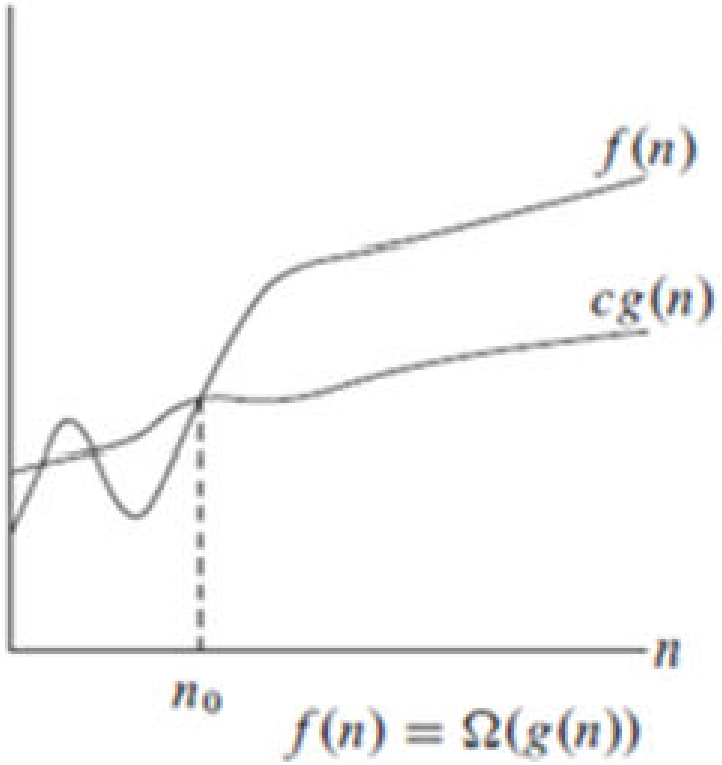
$n^2 2^{3 \log n} =$????

$2^{2 \log n} =$????

Own understanding:
Analyze complexity of bubble sort and merge sort by counting primitive functions.

Asymptotic Complexity Analysis: Ω

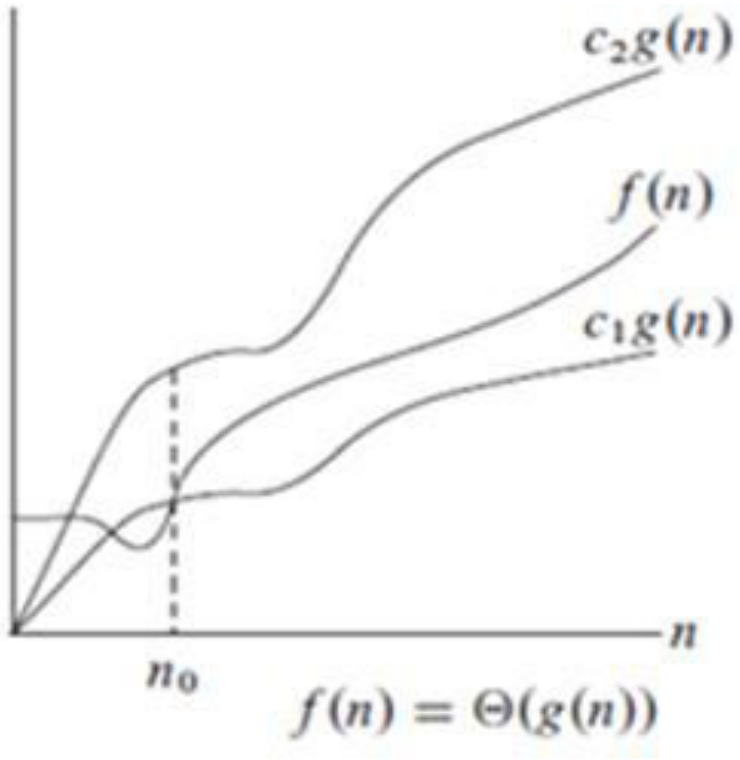
- Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $\Omega(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\boldsymbol{f(n) \geq cg(n)}$ for every integer $n \geq n_0$.
- This definition is often pronounced as "$f(n)$ is *big-Omega* of $g(n)$"
- Example: 7n − 2 is Ω (n).



Asymptotic Complexity Analysis: θ

Likewise, we say that $f(n)$ is $\Theta(g(n))$ (pronounced "$f(n)$ is big-Theta of $g(n)$")

if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$; that is, there are real constants $c_1 > 0$ and $c_2 > 0$, and an integer constant $n_0 \geq 1$ such that $\boldsymbol{c_1 g(n) \leq f(n) \leq c_2 g(n)}$, for $n \geq n_0$.



NOTE:
g(n) here is same on both the sides.

Left side represents Ω while right side represents θ

Inference:

Θ exists only when Ω(g(n) ∩ O(g(n) ≠ φ

Problem

Problem-1
Let $T(n)=3n^3 + 20n^2 + 5$ for an algorithm. Express in terms of Big(O), Big(Ω) and Big (θ)

Problem-2
Let T(n) be running time for an algorithm and its lower bound is $2n^2 + 1$ and its upper bound is $20n^2 + 5n+8$.
Express in terms of Big(O), Big(Ω) and Big (θ)

Problem-3
Express complexity of Bubble sort in terms of Big(O), Big(Ω) and Big (θ)

Problem Solving: homework

Comment on complexity of T(n)=1+2+3+.....+n in terms of θ, Ω and O

For each of the following pairs of functions, either f(n) is in O(g(n)), f(n) is in Ω(g(n)), or f(n) = Θ (g(n)). Determine which relationship is correct and briefly explain why.

- $f(n) = \log n^2$; $g(n) = \log n + 5$
- $f(n) = (n^2 - n)/2$, $g(n) = 6n$

Little-Oh and Little-Omega

- Let $f(n)$ and $g(n)$ be functions mapping integers to real numbers. We say that $f(n)$ is $o(g(n))$ (pronounced "$f(n)$ is little-oh of $g(n)$") if, for any constant $c > 0$, there is a constant $n_0 \geq 1$ such that $f(n) < cg(n)$ for $n \geq n_0$. **Eg: $7n + 8 \in o(n^2)$**

- Likewise, we say that $f(n)$ is $\omega(g(n))$ (pronounced "$f(n)$ is little-omega of $g(n)$") if $g(n)$ is $o(f(n))$, that is, if, for any constant $c > 0$, there is a constant $n_0 \geq 1$ such that $f(n) > cg(n)$ for $n \geq n_0$. **Eg: $4n + 6 \in \omega(1)$**

Note: Little-o, (o()) notation is used to describe an upper-bound that cannot be tight.

Little ω, notation is used to denote a lower bound that is not asymptotically tight.

Examples

- $x^2 \in O(x^2)$
- $x^2 \notin o(x^2)$
- $x^2 \in o(x^3)$

The following are true for Big-O, but would not be true if you used little-o:

- $x^2 \in O(x^2)$
- $x^2 \in O(x^2 + x)$
- $x^2 \in O(200 * x^2)$

The following are true for little-o:

- $x^2 \in o(x^3)$
- $x^2 \in o(x!)$
- $\ln(x) \in o(x)$

For example, the function $f(n) = 3n$ is:
•in $O(n^2)$, $o(n^2)$, and $O(n)$
•not in $O(\lg n)$, $o(\lg n)$, or $o(n)$

Limit Theorem

Notation	Common name	Limit test (note limit may not exist)
$f(n) \in O(g(n))$	Asymptotic upper bound	$\lim_{x \to \infty} \left\| \frac{f(x)}{g(x)} \right\| < \infty$
$f(n) \in o(g(n))$	Asymptotically negligible	$\lim_{x \to \infty} \left\| \frac{f(x)}{g(x)} \right\| = 0$
$f(n) \in \Omega(g(n))$	Asymptotic lower bound	$\lim_{x \to \infty} \left\| \frac{f(x)}{g(x)} \right\| > 0$
$f(n) \in \omega(g(n))$	Asymptotically dominant	$\lim_{x \to \infty} \left\| \frac{f(x)}{g(x)} \right\| = \infty$
$f(n) \in \Theta(g(n))$	Asymptotically tight bound	$0 < \lim_{x \to \infty} \left\| \frac{f(x)}{g(x)} \right\| < \infty$

Prove using limit theorem

∞ = 1/0

- Example 1: $3x^2+2x$ is $O(x^2)$

Prove using limit theorem

- Example 2: $3n^2+2n$ is $o(n^2)$
- Example 3: $3n^2+2n$ is $o(n^3)$

Prove using limit theorem

- Example 4: $3n^2+2n$ is $\Omega(n)$
- Example 5: $3n^2+2n$ is $\omega(n)$

Asymptotically tight notation

- Theta, commonly written as **Θ**, is an Asymptotic Notation to denote the ***asymptotically tight bound*** on the growth rate of runtime of an algorithm.
- For example, if f(n) is O(g(n)), and f(n) is O(h(n)) and if g(n) is O(h(n)) then g(n) is called the asymptotically tight notation.

Asymptotically tight notation

- So for log(n) we could say:
  log(n) is O(n^n) since log(n) grows asymptotically no faster than n^n
  log(n) is O(n) since log(n) grows asymptotically no faster than n
  log(n) is O(log(n)) since log(n) grows asymptotically no faster than log(n)
  We went from **loose upper bounds to a tight upper bound**

- log(n) is Ω(1) since log(n) grows asymptotically no slower than 1
  log(n) is Ω(log(log(n))) since log(n) grows asymptotically no slower than log(log(n))
  log(n) is Ω(log(n)) since log(n) grows asymptotically no slower than log(n)
  We went from **loose lower bounds to a tight lower bound**
  Since we have log(n) is O(log(n)) and Ω(log(n)) we can say that log(n) is Θ(log(n)).

For the functions, n^k and c^n, what is the asymptotic relationship between these functions?

Assume that k >= 1 and c > 1 are constants.

Choose all answers that apply:

- (A) n^k is $O(c^n)$
- (B) n^k is $\Omega(c^n)$ ✓
- (C) n^k is $\Theta(c^n)$

For the functions, $\log_2 n$ and $\log_8 n$, what is the asymptotic relationship between these functions?

Choose all answers that apply:

(A) $\log_2 n$ is $O(\log_8 n)$ ✓

(B) $\log_2 n$ is $\Omega(\log_8 n)$ ✓

(C) $\log_2 n$ is $\Theta(\log_8 n)$ ✓

For the functions, 8^n and 4^n, what is the asymptotic relationship between these functions?

Choose all answers that apply:

(A) 8^n is $O(4^n)$

(B) 8^n is $\Omega(4^n)$ ✓

(C) 8^n is $\Theta(4^n)$

Logarithmic and Exponent properties

- Let a, b, and c be positive real numbers. We have

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_c a)/\log_c b$
5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a/b^c = b^{a-c}$.

Try yourself!!

- Order the following growth rates in ascending order

$$6n\log n \quad 2^{100} \quad \log\log n \quad \log^2 n \quad 2^{\log n}$$
$$2^{2^n} \quad \lceil\sqrt{n}\rceil \quad n^{0.01} \quad 1/n \quad 4n^{3/2}$$

- Order these asymptotic notations by increasing growth rate.

$$n^{1/logn}, \quad e^n, \quad (\tfrac{3}{2})^n, \quad n^{loglogn}, \quad (\log n)^{\log n}$$

Some other observations we can make are:

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$
$$f(n) = \mathbf{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$
$$f(n) = \mathbf{o}(g(n)) \Leftrightarrow g(n) = \omega(f(n))$$

Big-Θ as an Equivalence Relation

If we look at the first relationship, we notice that $f(n) = \Theta(g(n))$ seems to describe an equivalence relation:

1. $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
2. $f(n) = \Theta(f(n))$
3. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, it follows that $f(n) = \Theta(h(n))$

Consequently, we can group all functions into equivalence classes, where all functions within one class are big-theta Θ of each other

Examples:

$$\mathbf{O}(n) + \mathbf{O}(n^2) + \mathbf{O}(n^4) = \mathbf{O}(n + n^2 + n^4) = \mathbf{O}(n^4)$$
$$\mathbf{O}(n) + \mathbf{\Theta}(n^2) = \mathbf{\Theta}(n^2)$$
$$\mathbf{O}(n^2) + \mathbf{\Theta}(n) = \mathbf{O}(n^2)$$
$$\mathbf{O}(n^2) + \mathbf{\Theta}(n^2) = \mathbf{\Theta}(n^2)$$

BASIC SORTING TECHNIQUES

REVISION

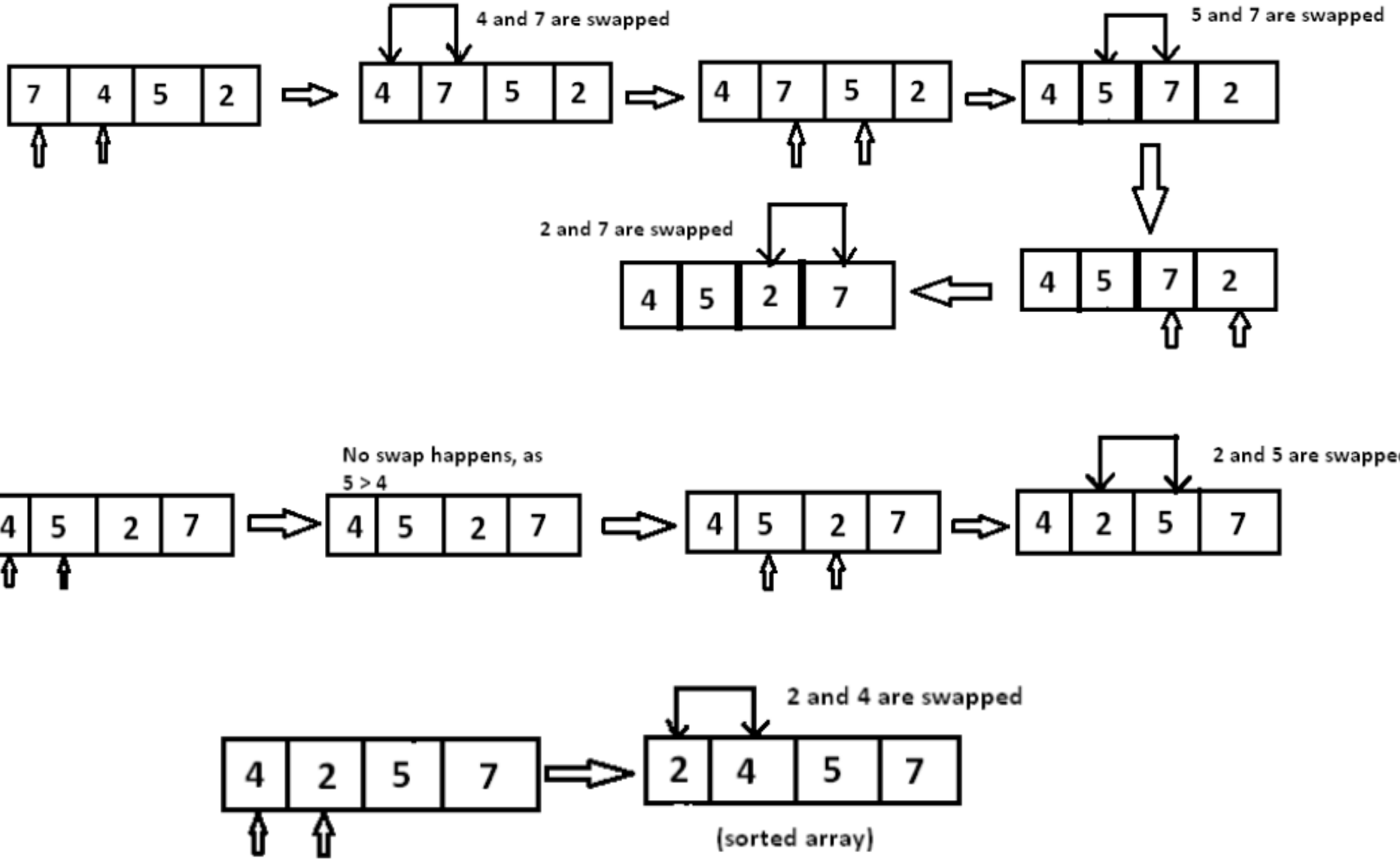
Bubble Sort

- Approach:
  - Compares the adjacent elements
- Swaps the elements if they are in wrong order.
- procedure bubbleSort(list : array of items)

```
loop = list.count;
for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
        if list[j] > list[j+1] then /* swap them */
                swap( list[j], list[j+1] )
                swapped = true
        end if
    end for
    if(not swapped) then
      break
    end if
end for
end procedure return list
```

Bubble Sort

Sort array a= [7, 4, 5, 2] in increasing order



Bubble Sort

- Sort array a1= [60, 50, 40, 30, 20, 10]

- Sort array a2=[10, 20, 30, 40, 50, 60]

Selection Sort

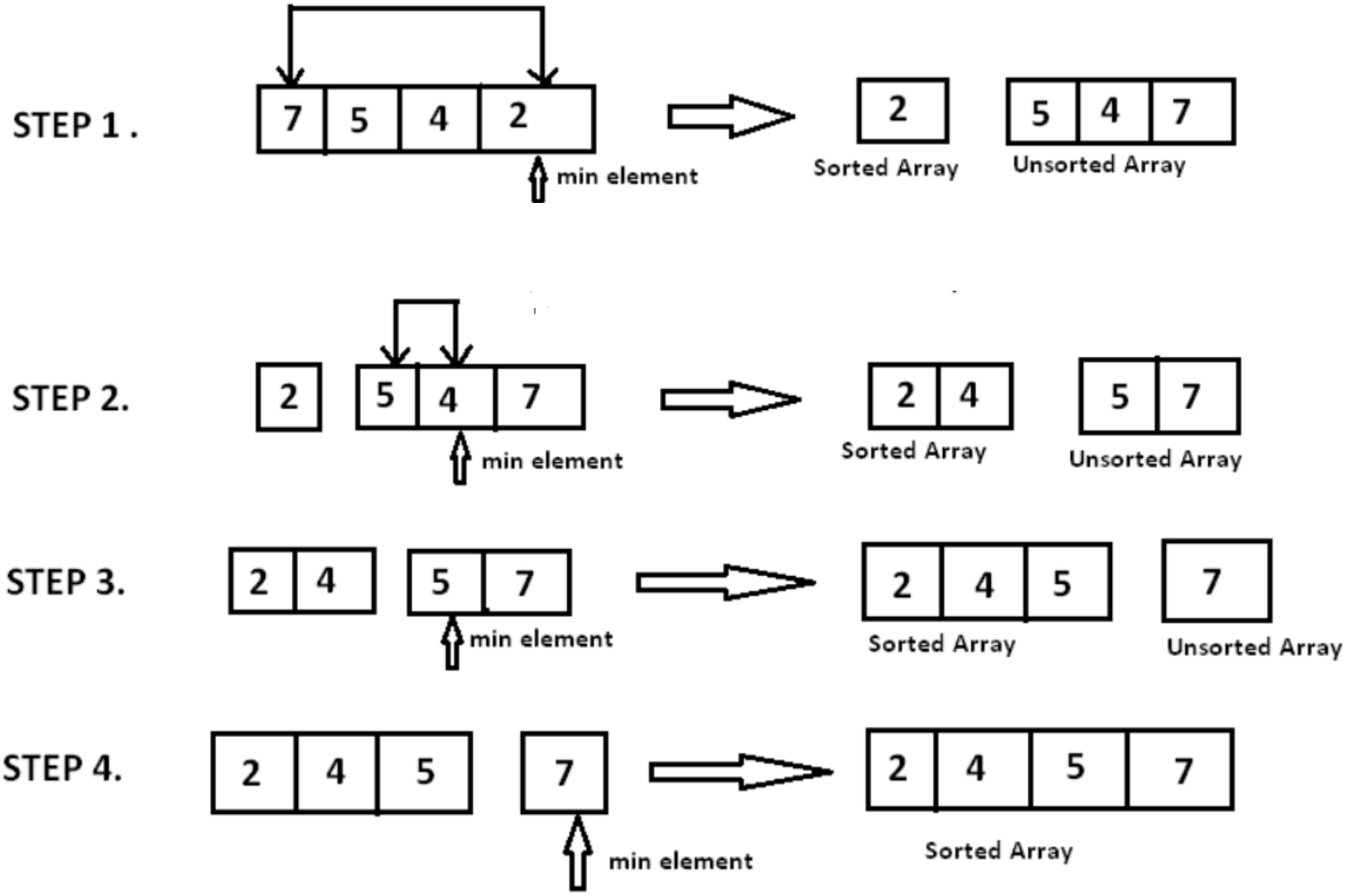
- Approach
  - Find the minimum or maximum element in an unsorted array
  - Put it in its correct position in a sorted array.

Selection Sort : Pseudocode

```
procedure selection sort list : array of items n : size of list
    for i = 1 to n - 1
        min = i          /* set current element as minimum*/
        for j = i+1 to n
            if list[j] < list[min] then /* check the element to be min*/
                min = j;
            end if
        end for /* swap the minimum element with the current element*/
        if indexMin != i then
            swap list[min] and list[i]
        end if
    end for
end procedure
```

Selection Sort

- Approach



Selection Sort

- Sort a=[60, 50, 40, 30, 20, 10]

- Sort a=[10, 20, 30, 40, 50, 60]

Insertion Sort

- Approach:
  - It considers part of a list sorted and rest of the list as unsorted.
  - Then continuously picks elements from the unsorted part and puts them at appropriate position in the sorted list by shifting the elements to right.

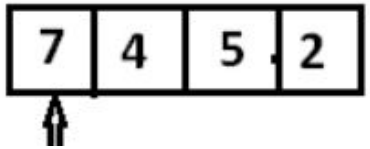
Insertion Sort

- Approach:
  - INSERTION-SORT(A)

```
for i = 0 to n-1
    temp ← A [i]
    j ← i
    while j >  0 and A[j-1] >temp
            A[j] ← A[j-1]
            j ← j – 1
    End while
    A[j] ← temp
End for
```

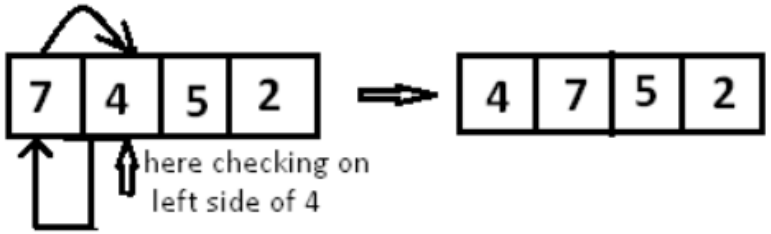
Insertion Sort

STEP 1.



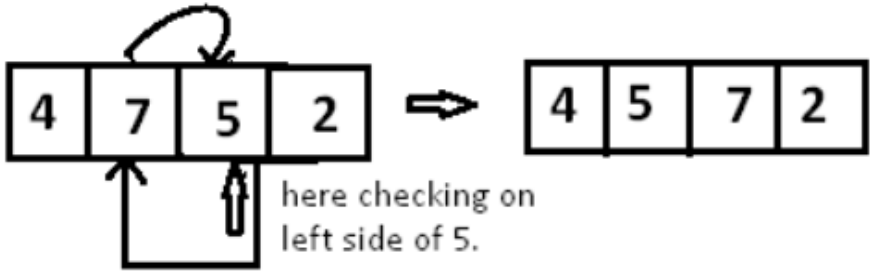
No element on left side of 7,so no change in its position .

STEP 2.



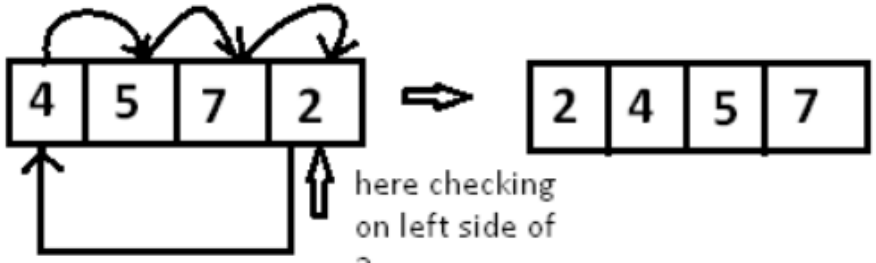
As 7>4 ,therfore 7 will be moved forward and 4 will be moved to 7's position.

STEP 3.



As 7>5,7 will be moved forward,but 4 <5,so no change in position of 4. And 5 will be moved to position of 7.

STEP 4.



As all the element on left side of 2 are greater than 2,so all the elements will be moved forward and 2 will be shifted to position of 4

Insertion Sort

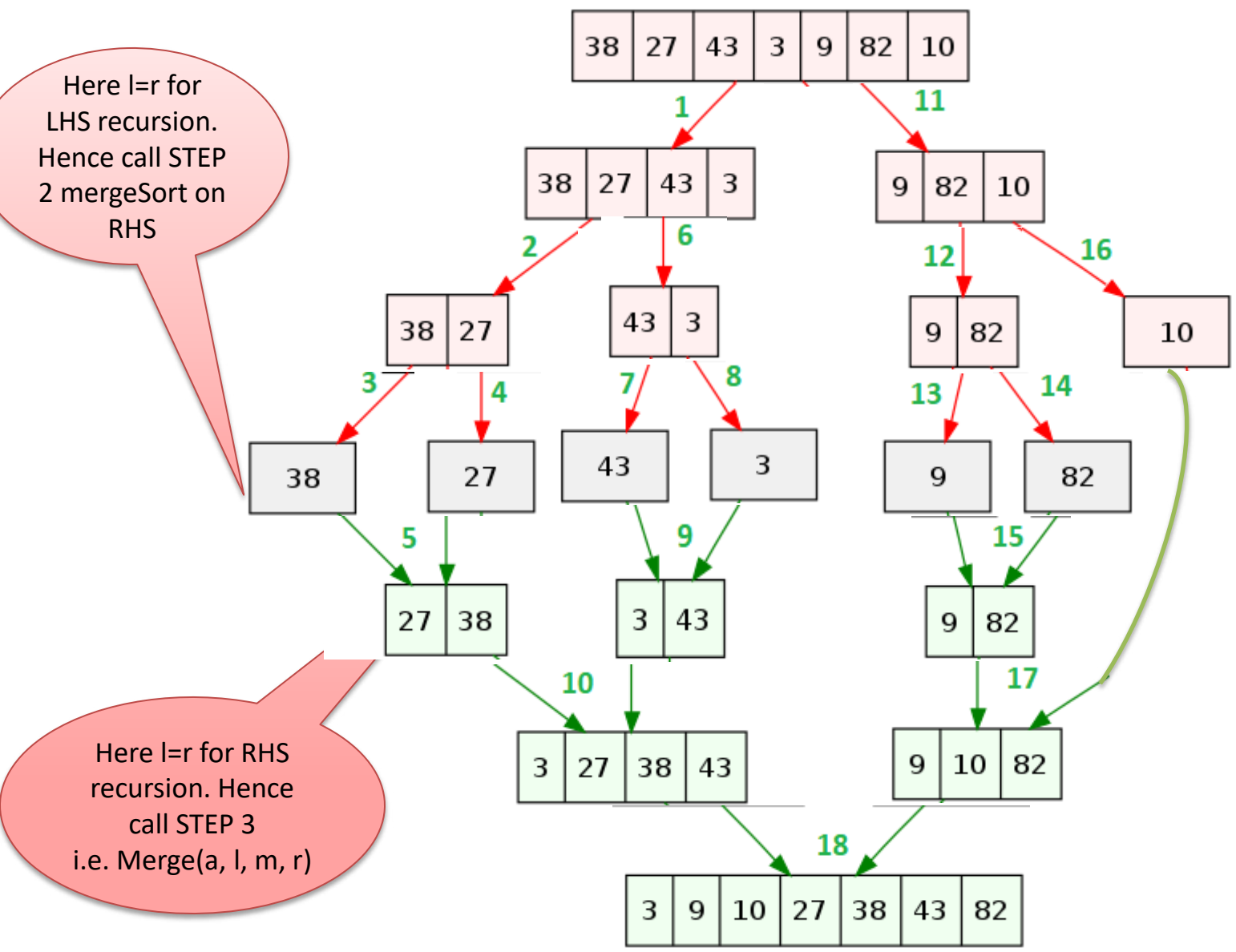
- Sort a=[60, 50, 40, 30, 20, 10]

- Sort a=[10, 20, 30, 40, 50, 60]

Merge sort approach

- Merge Sort is a Divide and Conquer algorithm.
  - First divides the problem into sub-problems
  - Then solve each sub-problem

- Merge sort consists of two main steps:
  - Divides the input array into two sub-arrays, calls itself for the two halves recursively.
  - Then merges the two sorted halves.


Here l=r for LHS recursion. Hence call STEP 2 mergeSort on RHS
Here l=r for RHS recursion. Hence call STEP 3 i.e. Merge(a, l, m, r)
38 27 43 3 9 82 10
1
11
38 27 43 3
9 82 10
2
6
12
16
38 27
43 3
9 82
10
3
4
7
8
13
14
38
27
43
3
9
82
5
9
15
27 38
3 43
9 82
10
17
3 27 38 43
9 10 82
18
3 9 10 27 38 43 82

Merge Sort

- Sort a=[60, 50, 40, 30, 20, 10]

- Sort a=[10, 20, 30, 40, 50, 60]

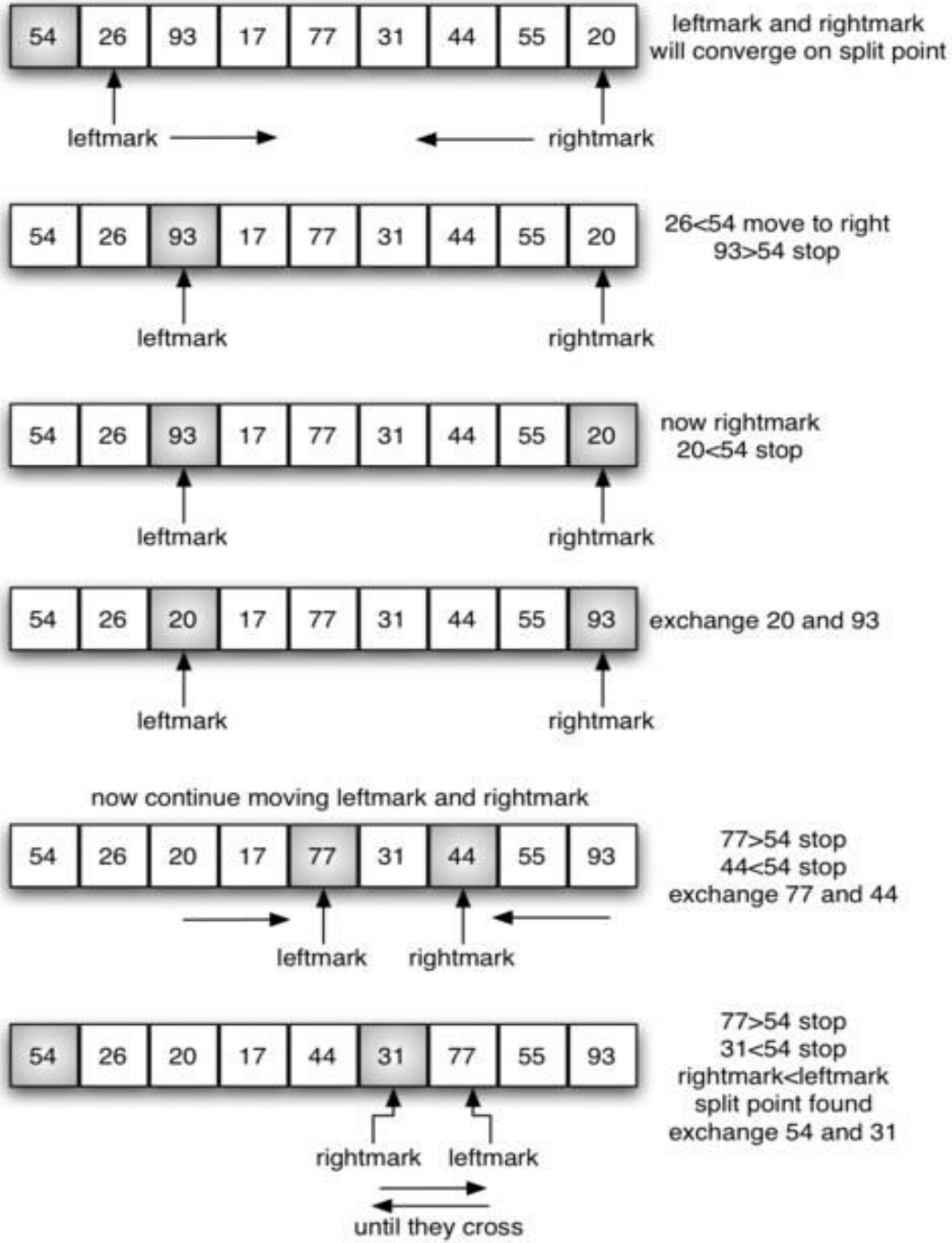
QUICK SORT

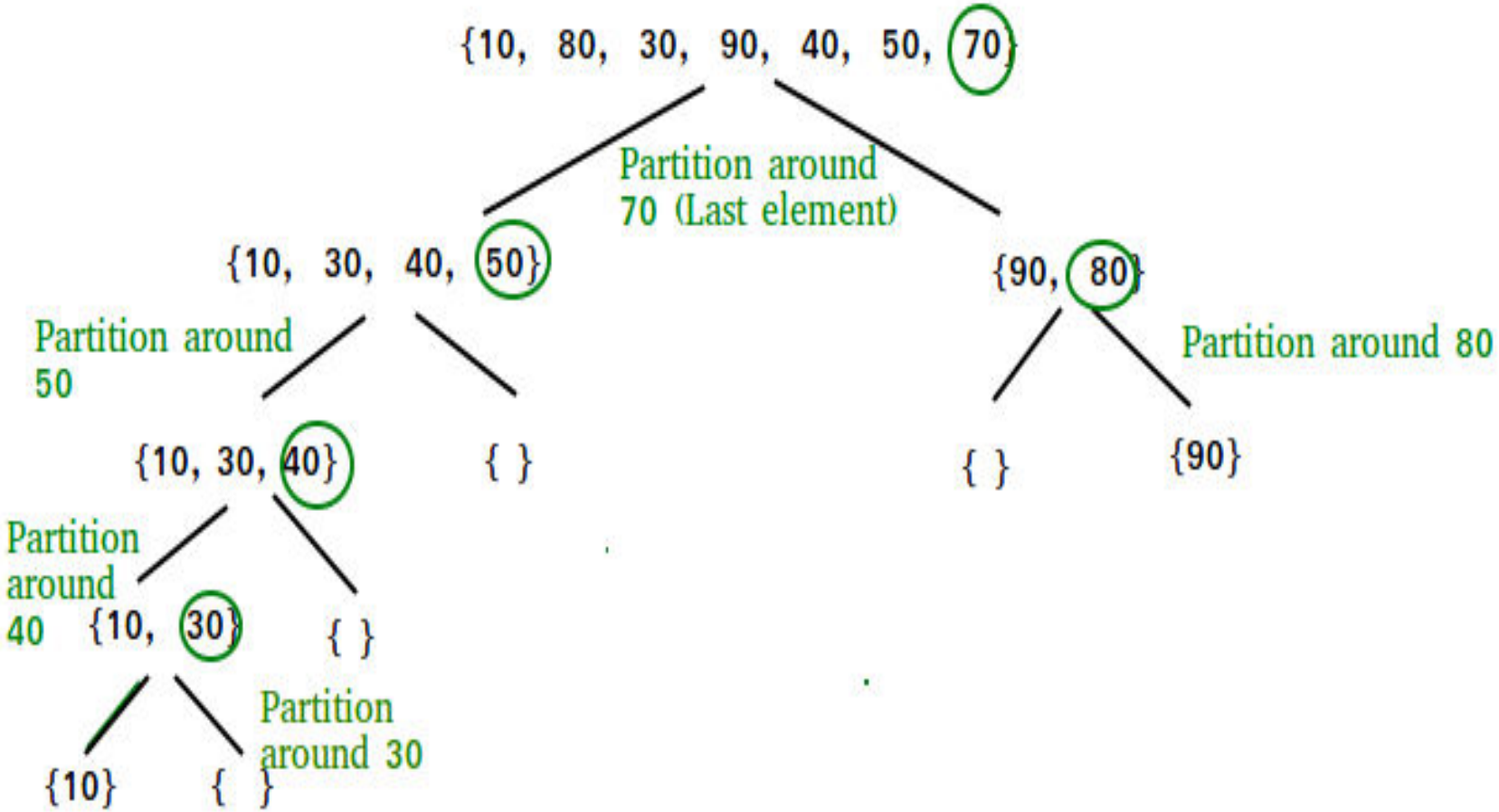
- Divide and Conquer approach
- It picks an element as pivot
- Partitions the given array around the picked pivot.
  - Put all smaller elements (smaller than ‘pivot’) before it, and put all greater elements (greater than pivot) after it

QUICK SORT

– Divide and Conquer approach

- In Quick sort algorithm, partitioning of the list is performed using following steps...
- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] < pivot then stop.
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i > j.
- **Step 7 -** Exchange the pivot element with list[j] element.


54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
54 26 93 17 77 31 44 55 20
26<54 move to right
93>54 stop
leftmark
rightmark
54 26 93 17 77 31 44 55 20
now rightmark
20<54 stop
leftmark
rightmark
54 26 20 17 77 31 44 55 93
exchange 20 and 93
leftmark
rightmark
now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
77>54 stop
44<54 stop
exchange 77 and 44
leftmark
rightmark
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
rightmark
leftmark
until they cross


{10, 80, 30, 90, 40, 50, 70}
Partition around
70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around
50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition
around
40
{10, 30}
{ }
Partition
around 30
{10}
{ }

Recurrence relations

By:
Dr. Rimjhim Singh
Asst. Professor (Sr. Gr)
CSE, Coimbatore

Recurrence relations

- ***A recurrence equation*** defines mathematical statements that the running time of a recursive algorithm must satisfy
- ***Recurrence relation*** is a mathematical model that captures the underlying time-complexity of an algorithm
- Writing recurrence relations
- Solving recurrence relations
  - Iterative substitution method
  - Recurrence tree method
  - Master's theorem

Writing recurrences: Ex-1

Decreasing Function

```
Void fun1(int n)
{
    if (n>0)
    {
        Print(n)
        fun1 (n-1)
    }
    else
        return
}
```

Writing recurrences : Ex-2

```
Void fun2 (int n)
{
    if (n>0)
    {
        for( i=0; i<n; i++)
        {
            Print(n)
        }
        fun2 (n-1)
    }
    else
        return
}
```

Writing recurrences : Ex-3

```
Void fun3 (int n)
{
    if (n>0)
    {
        for( i=1; i<n; i=i*2)
        {
            Print(n)
        }
        fun3 (n-1)
    }
    else
        return
}
```

Writing recurrences: Ex-4

```
Void fun4 (int n)
{
    if (n>0)
    {
        Print(n)
        fun4 (n/2)
    }
    else
        return
}
```

Writing recurrences: Ex – 5

```
Void fun5 (int n)
{
    if (n>0)
    {
        Print(n)
        fun5 (n-1)
        fun5 (n-1)
    }
    else
        return
}
```

Iterative substitution method/ Induction method

- Also known as the “plug-and-chug” method.
- Assumes that the problem size ***n is fairly large***
- ***Substitutes the general form*** of the recurrence for each occurrence of the function T on the right-hand side.

Recurrence tree method

- It is a *visual approach*.
- *Draw a tree R* where each node represents a different substitution of the recurrence equation.
- Thus, each node in R has a value of the argument ‘n’ of the function T(n) associated with it.
- In addition, *associate the overhead* with each node v in R,
  - **Overhead** is defined as the **value of the non-recursive** part of the recurrence equation for v.

Solving recurrences: Ex-1

```
Void fun1(int n)
{
    if (n>0)
    {
        Print(n)
        fun1 (n-1)
    }
    else
        return
}
```

Solving recurrences : Ex-2

```
Void fun2 (int n)
{
    if (n>0)
    {
        for( i=0; i<n; i++)
        {
            Print(n)
        }
        fun2 (n-1)
    }
    else
        return
}
```

Solving recurrences : Ex-3

```
Void fun3 (int n)
{
    if (n>0)
    {
        for( i=1; i<n; i=i*2)
        {
            Print(n)
        }
        fun3 (n-1)
    }
    else
        return
}
```

Solving recurrences: Ex-4

```
Void fun4 (int n)
{
    if (n>1)
    {
        Print(n)
        fun4 (n/2)
    }
    else
        return
}
```

Solving recurrences: Ex – 5

```
Void fun5 (int n)
{
    if (n>0)
    {
        Print(n)
        fun5 (n-1)
        fun5 (n-1)
    }
    else
        return
}
```