Introduction to Algorithms

Prepared by

Dr. Kshyamasagar Mahanta

Asst. prof., CSE,CVRP

Learning Objectives

At the end of this lecture, students will be able to:

1. Define an algorithm.
2. Explain the importance of algorithms in problem-solving.
3. Identify the characteristics of a good algorithm.
4. Distinguish between an algorithm and a computer program.
5. Write simple algorithms for basic problems.

How do you make a cup of tea?

To perform any task, we follow a sequence of steps. Similarly, a computer requires a sequence of instructions to solve a problem. This sequence of instructions is called an **Algorithm**.

What is an Algorithm?

Simple Definition

- An algorithm is a step-by-step procedure for solving a problem.

- **An algorithm is a finite sequence of well-defined instructions used to solve a problem or perform a computation.**

The phrase breaks down into three core concepts:

- **Finite Sequence:** It must always have a clear starting and ending point. It will eventually stop, ensuring the computer does not get stuck in an endless loop.
- **Well-Defined Instructions:** Every single step is completely unambiguous. There is no guesswork involved; whether it is executed by a human or a computer, the instructions produce the exact same results every time .
- **Solve a Problem or Perform a Computation:** The ultimate goal is to process data, sort information , or calculate mathematical equations systematically until the desired task is finished.

Historical Background

Ancient Origins: The word is named after **Muhammad ibn Musa al-Khwarizmi**, a brilliant 9th-century Persian mathematician from modern-day Uzbekistan.

The Latinization: When medieval European scholars translated his math books into Latin, his Arabic name was changed to "Algoritmi" for easier pronunciation.

Original Meaning: Initially, the word "algorism" referred strictly to the step-by-step rules of doing basic arithmetic using Hindu-Arabic numerals.

Language Mix-up: Over centuries, people mistakenly blended the word with the Greek word arithmos (meaning "number"), which eventually created the modern spelling "algorithm".

Al-Khwarizmi also gave us the word "algebra", which comes from al-jabr, a term used in the title of his landmark mathematics book.

Why Do We Need Algorithms?

Algorithms help us:

- Solve problems systematically
- Reduce errors
- Improve efficiency
- Automate tasks
- Develop computer programs

Real-Life Examples

Example 1: Making Tea

1. Take a pan.
2. Pour water.
3. Boil water.
4. Add tea leaves.
5. Add milk.
6. Add sugar.
7. Boil for 2 minutes.
8. Filter and serve.

This sequence forms an algorithm.

Example 2: ATM Cash Withdrawal

1. Insert ATM card.
2. Enter PIN.
3. Select Withdraw.
4. Enter amount.
5. Verify balance.
6. Dispense cash.
7. Print receipt.
8. Return card.

Example 3: Finding the Largest Number

Suppose:

A = 20

B = 35

C = 15

Steps:

1. Compare A and B.
2. Compare larger value with C.
3. Display the largest value.

Characteristics of an Algorithm

A good algorithm must satisfy **five important properties.**

1. Input

- An algorithm may have zero or more inputs.

2. Output

- Every algorithm must produce at least one output.

Algorithm for addition:

Inputs:

A = 10

B = 20

Output:

Sum = 30

3. Finiteness

- An algorithm must terminate after a finite number of steps.

Correct Algorithm

Step 1: Read N

Step 2: Print N

Step 3: Stop

Terminates after 3 steps.

Incorrect Algorithm

Repeat forever

    Print "Hello"

Never stops.

Therefore, it is not a valid algorithm.

4. Definiteness

Every step must be clear and unambiguous.

Ambiguous Instruction

Add some sugar

Question:

How much sugar?

The instruction is unclear.

Add 2 teaspoons of sugar

Now the instruction is clear.

5. Effectiveness

Every instruction must be practical and executable.

Effective Step

Add two numbers

Can be performed.

Ineffective Step

Predict tomorrow's lottery result

Cannot be reliably executed.

Hence not effective.

Writing Simple Algorithms

Example 1: Add Two Numbers

Algorithm ADD

Step 1: Start
Step 2: Read A and B
Step 3: SUM ← A + B
Step 4: Display SUM
Step 5: Stop

Example 2: Calculate Area of Rectangle

Algorithm AREA_RECTANGLE

Step 1: Start
Step 2: Read Length and Width
Step 3: Area ← Length × Width
Step 4: Display Area
Step 5: Stop

Writing Simple Algorithms

Example 3: Find Larger of Two Numbers

Algorithm MAX

Step 1: Start

Step 2: Read A and B

Step 3:

If A > B

    Display A

Else

    Display B

Step 4: Stop

Algorithm vs Program

Algorithm	Program
1. Step-by-step solution	1. Actual code
2. Language independent	2. Language dependent
3. Easier to understand	3. Requires programming language
4. Conceptual	4. Executable
5. Written before coding	5. Written after algorithm design

Example

Algorithm

- Read A and B
- SUM ← A + B
- Display SUM

Program

```
#include<stdio.h>
int main()
{
   int a,b,sum;
   scanf("%d%d",&a,&b);
   sum = a + b;
   printf("%d",sum);
   return 0;
}
```

Questions

1. Write an algorithm to calculate Area of Circle
2. Write an algorithm to calculate Simple Interest
3. Write an algorithm to calculate the average of three numbers entered by the user.
4. Determine Whether a Number is Even or Odd
5. Write an algorithm to calculate the total marks and percentage of a student in five subjects.

Assignments

1. Write an algorithm to find the smallest among three numbers.
2. Write an algorithm to calculate the average of five numbers.
3. Identify the five characteristics of an algorithm with examples.
4. Differentiate between an algorithm and a program with suitable examples.
5. Write the algorithm for withdrawing money from an ATM.

Lecture 2: Pseudocode

Prepared by

Dr. Kshyamasagar Mahanta

Learning Objectives

After completing this lecture, students will be able to:

- Define pseudocode.
- Explain the need for pseudocode in algorithm design.
- Identify common pseudocode constructs.
- Write simple algorithms using pseudocode.
- Convert problem statements into pseudocode solutions.

What is Pseudocode?

Pseudocode is an informal high-level description of an algorithm that uses simple English-like statements to describe the logic of a program.

Simple Definition

- Pseudocode is a way of writing algorithms using plain English mixed with programming-like keywords.
- It is not a programming language.
- It is not executed by a computer.
- It is used for planning solutions before writing actual code.

Meaning of the Word Pseudocode

- "Pseudo" means false or imitation.
- "Code" refers to programming instructions.
- Therefore, **Pseudocode means "false code" or "code-like representation.“**

Why Do We Need Pseudocode?

Consider a problem:

Find the larger of two numbers.

- Before writing a C, Java, or Python program, we first think about the solution.
- Pseudocode helps us organize our thoughts and logic clearly.

Advantages of Pseudocode

1. Easy to Understand

Pseudocode resembles ordinary English.

Example:

```
READ A, B
IF A > B
   PRINT A
ELSE
   PRINT B
ENDIF
```

Even a non-programmer can understand the logic.

2. Language Independent

The same pseudocode can be converted into: C ,C++ ,Java ,Python ,JavaScript

Example:

```
SUM ← A + B
```

The logic remains the same regardless of programming language.

3. Helps Before Coding

- Pseudocode acts as a blueprint.
- Just as engineers create a building plan before construction, programmers create pseudocode before coding.

Benefits:

- Better planning
- Fewer errors
- Easier debugging
- Faster coding

4. Improves Communication

Team members can discuss algorithms without worrying about programming language syntax.

5. Easy to Modify

Changes can be made quickly before actual coding begins.

Rules for Writing Pseudocode

There is no strict standard. However, some common conventions are followed.

Use Simple English

Example: READ Number

Use Meaningful Variable Names

Eample: READ Salary

Use Indentation

Indentation improves readability.

Example:

```
IF Marks >= 40
    PRINT "Pass"
ELSE
    PRINT "Fail"
ENDIF
```

Common Pseudocode Keywords

Keyword	Meaning
START	Beginning of algorithm
STOP	End of algorithm
READ	Accept input
WRITE / PRINT	Display output
IF	Decision making
ELSE	Alternative decision
ENDIF	End of IF block
WHILE	Repetition
FOR	Looping
←	Assignment operator

Example 1: Find Maximum of Two Numbers

Problem Find the larger of two numbers.

Pseudocode

```
Algorithm FindMaximum
READ A, B
IF A > B THEN
   MAX ← A
ELSE
   MAX ← B
ENDIF
PRINT MAX
```

Example 2: Add Two Numbers

Problem Add two numbers.

Pseudocode

```
START
READ A, B
SUM ← A + B
PRINT SUM
STOP
```

Example 3: Calculate Area of Rectangle

```
START
READ Length, Width
Area ← Length × Width
PRINT Area
STOP
```

Example 4: Check Even or Odd Number

```
START
READ N
IF N MOD 2 = 0 THEN
   PRINT "Even"
ELSE
   PRINT "Odd"
ENDIF
STOP
```

Algorithm	Pseudocode
General step-by-step solution	Structured representation
Written in plain steps	Written using programming-style keywords
Less formal	More formal
Easier for beginners	Closer to actual coding

Q. Write pseudocode for the following problem

1. Calculate the square of a number.
2. Calculate Simple Interest.
3. Find the smaller of two numbers.
4. Write pseudocode to calculate the average of three numbers.
5. Write pseudocode to calculate the area of a circle.
6. Write pseudocode to find the largest among three numbers.
7. Calculate the Perimeter of a Rectangle
8. Convert Temperature from Celsius to Fahrenheit
9. Calculate Total Marks and Percentage
10. Check Whether a Number is Positive, Negative, or Zero

.

Algorithm	Program
A step-by-step procedure to solve a problem.	A set of instructions written in a programming language that a computer can execute.
Written in simple language, pseudocode, or flowcharts.	Written in languages like C, Java, Python, etc.
Independent of any programming language.	Depends on a specific programming language.
Focuses on the logic of solving a problem.	Focuses on implementing the algorithm on a computer.
Cannot be executed directly by a computer.	Can be compiled or interpreted and executed by a computer.

.

Algorithm

A step-by-step method for solving a problem.

Can be written in plain English, flowcharts, or other formats.

Focuses on the logic and sequence of steps.

May be more general and descriptive.

Independent of any programming language.

Pseudocode

An informal way of writing an algorithm using programming-like statements.

Written in a structured, code-like format without following the syntax of a specific programming language.

Focuses on expressing the algorithm in a form that is easy to convert into a program.

More detailed and resembles actual programming code.

Also language-independent, but uses programming constructs such as IF, WHILE, and FOR.

Unit II: Algorithmic Complexity

Prepared by

Dr. Kshyamasagar Mahanta

Learning Objectives

At the end of this lecture, students will be able to:

- Understand the need for complexity analysis.
- Explain why different algorithms for the same problem may perform differently.
- Define efficiency in terms of time and memory.
- Compare algorithms based on resource consumption.
- Appreciate the importance of selecting efficient algorithms.

Q. "If there are two routes from Bhubaneswar to Cuttack, which one would you choose?"

Q. "If there are two routes from Bhubaneswar to Cuttack, which one would you choose?“

- Shorter route
- Faster route
- Route with less traffic

Because it saves time and fuel.

Similarly, in computer science:

- Multiple algorithms may solve the same problem, but some are faster and require less memory.
- Therefore, we must analyze algorithms before choosing one.

Why Analyze Algorithms?

Complexity analysis is the process of evaluating the efficiency of an algorithm in terms of:

- Execution Time
- Memory Usage

Problem Statement

Suppose we want to search for a student's roll number in a list of 1000 students.

Algorithm 1

Check each roll number one by one.

Algorithm 2

Use Binary Search on a sorted list.

- Both algorithms produce the same answer.

However:

- Algorithm 1 takes more time.
- Algorithm 2 takes less time.

Therefore: **Correctness alone is not sufficient.** Efficiency is also important.

Main Goal : Choose the most efficient algorithm among available alternatives.

Resources Used by an Algorithm

When an algorithm runs, it consumes computer resources.

The two most important resources are:

1. Time

Time required to complete execution.

Example

- Suppose you need to find a name in a phone directory.

Method 1:

Start from page 1 and check every name.

- Method 2:

Open the middle page and repeatedly divide the search area.

- Method 2 is much faster. Thus: Different algorithms consume different amounts of time.

2. Memory

Memory used while executing.

Example

Suppose two students solve the same mathematical problem.

Student A: Uses one sheet of paper.

Student B: Uses ten sheets of paper.

- Both obtain the same answer.
- Which method is preferable?

Obviously Student A's method.

Similarly, algorithms should use less memory whenever possible.

Why Are Time and Memory Important?

Modern applications handle:

- Millions of users
- Gigabytes of data
- Real-time processing

Inefficient algorithms can cause:

- Slow execution
- System crashes
- High hardware costs
- Poor user experience

Comparing Algorithms

Example 1: Finding Maximum Element

Given : 5, 12, 8, 25, 3

Algorithm A

Compare each element one by one.

Steps:

MAX ← 5

Compare 12 and 5

Compare 8 and 12

Compare 25 and 12

Compare 3 and 25

Total Comparisons = 4

Algorithm B

Sort the entire array first and then take the largest element.

Steps:

Sort all elements

Take last element

This requires many more operations.

Example 2: Searching a Number

10 20 30 40 50 60 70 80

Find 70

Linear Search

Check each element starting from 1st element.

-no of comparision 7.

Binary Search

Check middle element: 40 then 60 then 70

-no of comparision 3

Conclusion

Both produce the same answer.

Binary Search is more efficient.

Real-Life Importance of Complexity Analysis

Example 1: Google Search

When you search something on Google:

- Billions of web pages exist.
- Results appear within seconds.

Why?

- Efficient algorithms.

Example 2: Online Shopping

Websites:

- Amazon
- Flipkart

Need to search millions of products quickly.

Efficient algorithms make this possible.

Example 3: Navigation Apps

Apps such as:

- Google Maps
- Waze

Must find shortest routes quickly.

Complexity analysis helps select the best route-finding algorithms.

Example 4: Social Media

Platforms process:

Millions of posts

- Images
- Videos

Efficient algorithms ensure smooth operation.

What Happens if We Ignore Complexity?

Suppose: Dataset Size = 10

Algorithm works well.

Now:

Dataset Size = 10,000,000

The same algorithm may become:

Extremely slow

- Memory intensive
- Practically unusable

Therefore:

As input size grows, efficiency becomes critical.

Analogy: Bicycle vs Car

Imagine traveling: Distance = 2 km

- Both bicycle and car work fine.

Distance = 500 km. Which is preferable?

- Car.

Similarly:

For small data, many algorithms work.

For large data, only efficient algorithms are practical.

Key Idea

When evaluating an algorithm, we ask:

Question 1

How much time does it take?

Question 2

How much memory does it require?

These two questions form the foundation of complexity analysis.

Q1. Suppose you have a dictionary. Which method is faster?

A. Start from first page and search word by word.

B. Open near the expected alphabet and narrow down.

Q2. You need to find one student in a classroom of 100 students.

Method A: Ask each student individually.

Method B: Use the attendance register arranged by roll number.

Which method is more efficient?

Lecture 5: Time Complexity

Prepared by

Dr. Kshyamasagar Mahanta

Learning Objectives

After completing this lecture, students will be able to:

- Define time complexity.
- Understand how algorithm execution time depends on input size.
- Count basic operations in simple algorithms.
- Express time complexity using Big-O notation.
- Determine the time complexity of simple loops.

Introduction to Time Complexity

Definition

Time Complexity measures how the execution time of an algorithm grows as the size of input increases.

Important Note

Time complexity does NOT measure actual time in seconds.

Because actual execution time depends on:

- Computer speed
- Processor
- Memory
- Programming language

Instead, we count the number of operations performed.

Example Suppose you have to count students in a classroom.

For: 10 students you perform 10 counting operations.

For: 100 students you perform 100 counting operations.

More students ⇒ More operations ⇒ More time.

Input Size and Growth of Execution Time

What is Input Size?

Input size is represented by: n , where n denotes the number of input elements.

Example Consider:

```
for(i=1;i<=n;i++)
{
printf("%d",i);
}
```

If n = 5, o/p: 1 2 3 4 5 loop executes 5 times

if n= 100 loop executes 100 times

If n=1000 loop executes 1000 times

As input grows, execution time also grows.

Counting Operations

Why Count Operations?

Time complexity is determined by counting the dominant operations.

Examples of operations:

- Addition
- Multiplication
- Comparison
- Assignment
- Printing

Example 1

```
a = b + c;
```

Operations:

1 Addition

1 Assignment

Total: 2 Operations

Example 2

```
for(i=1;i<=n;i++)
{
   print(i);
}
```

Loop executes: n times

Print statement executes: n times

Therefore: Number of operations = n

- **Time Complexity: O(n)** Called: **Linear Time Complexity**

Example 3

```
for(i=1;i<=10;i++)
{
   print(i);
}
```

Loop executes: 10 times

Regardless of input size.

Time Complexity: O(1) Called: Constant Time Complexity

Understanding Big-O Notation

Why Use Big-O?

Suppose: Algorithm A performs: 2n + 5 operations

Algorithm B performs: 3n + 20 operations

For large values of n: The constant values become insignificant.

Therefore:

$2n + 5 \approx n$

$3n + 20 \approx n$

Hence both are: O(n)

Big-O focuses on growth rate.

Common Time Complexities

Complexity	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Linearithmic
$O(n^2)$	Quadratic
$O(2^n)$	Exponential

Growth Comparison

For: $n = 1000$

Complexity	Approximate Operations
$O(1)$	1
$O(\log n)$	10
$O(n)$	1000
$O(n^2)$	1,000,000

Observation: Smaller complexity is better.

Examples of Time Complexity

Example 1: Constant Time

```
a = b + c;
```

Operations: Fixed

Time Complexity: O(1)

Example 2: Linear Time

```
for(i=1;i<=n;i++)
{
  printf("%d",i);
}
```

Operations: n

Time Complexity: O(n)

Example 3: Two Independent Loops

```
for(i=1;i<=n;i++)
{
  print(i);
}

for(j=1;j<=n;j++)
{
  print(j);
}
```

Operations: n + n = 2n Time complexity: O(n)

Example 4: Nested Loops

```
for(i=1;i<=n;i++)
{
   for(j=1;j<=n;j++)
   {
      print(i,j);
   }
}
```

Operations: n × n

Time Complexity: $O(n^2)$ Called: Quadratic Time Complexity

Find the time complexity of the following:

1.

```
for(i=1;i<=20;i++)
{
printf("Hello");
}
```

2.

```
for(i=1;i<=n;i++)
{
printf("Hello");
}
```

3.

```
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf("*");
}
}
```

Lecture 6: Space Complexity

Prepared by

Dr. Kshyamasagar Mahanta

Learning Objectives

At the end of this lecture, students will be able to:

- Define space complexity.
- Understand why memory usage is important in algorithm design.
- Identify the components of space complexity.
- Differentiate between input space and auxiliary space.
- Calculate the space complexity of simple algorithms.
- Determine common space complexities such as $O(1)$, $O(n)$, and $O(n^2)$.

Introduction to Space Complexity

Definition : Space Complexity is the amount of memory required by an algorithm during its execution.

Why is Memory Important?

Every program requires memory for:

- Variables
- Arrays
- Functions
- Data structures
- Temporary storage

If memory usage becomes very large:

- Program becomes slow
- System performance decreases
- Memory overflow may occur

Need for Space Complexity Analysis

Suppose we have two algorithms: **Algorithm A** Uses: 1 MB memory , **Algorithm B** Uses:500 MB memory

Both solve the same problem.

Which is better?

Answer: Algorithm A

because it consumes less memory.

Goal

Choose algorithms that:

- Use less memory
- Perform efficiently
- Scale to larger datasets

Components of Space Complexity

Space complexity consists of two major components:

1. Input Space 2. Auxiliary Space

Input Space: Memory required to store the input data.

Ex. int marks[100];

Array stores 100 integers

Memory required for these inputs is called **Input Space**.

Auxiliary Space: Extra memory used by an algorithm excluding input data.

Ex. int sum = 0;

Memory used by **sum** is **Auxiliary Space**.

Space Complexity = Input Space + Auxiliary Space

Constant Space Complexity O(1)

If memory usage remains constant regardless of input size, the algorithm has: O(1) space complexity.

Example 1 : int sum=0;

Uses one memory location.

Input size does not matter.

Space Complexity: O(1)

Example 2 : int a,b,c;

Three variables only.

Memory remains fixed.

Space Complexity: O(1)

Linear Space Complexity O(n)

Example : int arr[n];

Memory required: n elements

If n = 10, Memory for 10 elements.

If n = 1000, Memory for 1000 elements.

Memory increases with input size.

Space Complexity: O(n)

Example: int marks[n];

Space Complexity O(n) because memory grows linearly.

Quadratic Space Complexity $O(n^2)$

Example : int matrix[n][n];

Memory required: n × n elements.

If n = 10, Memory: 100 elements

If n = 100, Memory: 10,000 elements

Space Complexity: $O(n^2)$

Space Complexity	Example
O(1)	int sum=0
O(n)	int arr[n]
$O(n^2)$	int matrix[n][n]

Example 1:

```
int sum=0;
for(i=1;i<=n;i++)
{
   sum=sum+i;
}
```

Variables: sum i

Only two variables.

Space Complexity: O(1)

Even though time complexity is O(n).

Find the space complexity:

Q1. int x;

Q2. int a,b,c,d,e;

Q3. int matrix[10][10];

Q4. int sum = 0; for(int i=1; i<=n; i++) { sum = sum + i; }

Q5. int arr[n]; int sum = 0;

Q6. int arr[n]; int temp[n];

Q7. int arr[n]; int matrix[n][n];

Q8. int cube[n][n][n];

Q9. int x = 10; while(x > 0) { x--;}

Q10. int arr[n]; for(int i=0; i<n; i++) { for(int j=0; j<n; j++) { printf("*"); } }

Q11. int graph[n][n]; int visited[n];

Lecture 7: Best Case, Average Case, and Worst Case Analysis

introduction to Case Analysis

Consider searching for a student named "Rahul" in a classroom attendance register.

Possible situations:

Situation 1

Rahul's name is the first entry.

Search is completed immediately.

Situation 2

Rahul's name is the last entry.

Need to check all names.

Situation 3

Rahul's name is somewhere in the middle.

Need to check several entries.

Observation

The same algorithm behaves differently for different inputs.

Therefore, we analyze:

1. Best Case

Minimum execution time.

2. Average Case

Typical execution time.

3. Worst Case

Maximum execution time.

Linear Search Review

What is Linear Search?

Linear Search checks elements one by one until the desired element is found.

Algorithm

```
LinearSearch(A, n, key)

for i = 0 to n-1
    if A[i] = key
        return i

return Not Found
```

Example Array

1 2 3 4 5

Search Key:

5

Best Case Analysis

Best Case occurs when the desired element is found immediately.

Minimum number of operations are performed.

Example

Array:

1 2 3 4 5

Search: 1

Found immediately. Comparisons 1

Complexity O(1) (Constant Time)

Key Point : Best Case does not depend on the size of the array.

Whether: n = 10 or n = 1,000,000

If the first element is found immediately: 1 comparison

Best Case remains: O(1)

Worst Case Analysis

Worst Case occurs when the maximum number of operations are performed.

Situation 1

Desired element is the last element.

Array: 1 2 3 4 5

Search: 5

Total Comparisons 5

Situation 2

Element does not exist.

Array: 1 2 3 4 5

Search: 10

Comparisons 5

General Case

For an array of size n: Worst-case comparisons: n

Therefore: Worst Case Complexity O(n) (Linear Time)

Average Case Analysis

Average Case represents the expected performance for a typical input.

Example

Array: 1 2 3 4 5

Search key can be:1,2,3,4,5 with equal probability.

Average Comparisons

$$\frac{1+2+3+4+5}{5}$$

$$= \frac{15}{5}$$

$$= 3$$

For n elements , no of comparisons: n+1/2

Ignoring constants : O(n)

Observation

Average Case is usually closer to Worst Case than Best Case.

Therefore, algorithm designers often focus on:

- Average Case
- Worst Case

Lecture 8: Big-O Notation and Finding the Complexity of an Algorithm

Introduction to Big-O Notation

Big-O notation is a mathematical representation used to describe the upper bound (worst-case growth rate) of an algorithm.

Big-O tells us:

How fast the running time of an algorithm grows as the input size increases.

Why Do We Need Big-O?

Suppose two students write programs to find the largest number in an array.

Algorithm A Needs:100 operations

Algorithm B Needs:1000 operations

For small data, both work.

For large data: Algorithm A is better.

Big-O helps compare algorithms regardless of:

- Computer speed
- Programming language
- Operating system

Rules for Finding Complexity

Rule 1: Ignore Constants

Example:

```
for(i=1;i<=2*n;i++)
{
  printf("%d",i);
}
```

Operations: 2n , Ignoring constant 2: O(n)

Rule 2: Ignore Lower Order Terms

Example:$n^2 + n + 10$

Dominant term:n^2 , Complexity: $O(n^2)$

Expression	**Complexity**
n + 5	O(n)
2n + 100	O(n)
$n^2 + n$	$O(n^2)$
$n^3 + n^2 + n$	$O(n^3)$

Rule 3: Sequential Loops Add

Example

```
for(i=1;i<=n;i++)
{
  printf("A");
}
for(j=1;j<=n;j++)
{
  printf("B");
}
```

Operations: n+n = 2n

Complexity: O(n)

Rule 4: Nested Loops Multiply

Example

```
for(i=1;i<=n;i++)
{
  for(j=1;j<=n;j++)
  {
    printf("*");
  }
}
```

Operations: n × n

Complexity: $O(n^2)$

Q. Find complexity of algorithms

1. int a = 10; int b = 20; int c = a + b; printf("%d", c);
2. for(i=1; i<=n; i++) { printf("%d", i);}
3. for(i=1; i<=100; i++) { printf("Hello"); }
4. for(i=1; i<=2*n; i++) { printf("*"); }
5. for(i=1; i<=n; i++) { printf("*"); } for(j=1; j<=n; j++) { printf("#"); }
6. for(i=1; i<=n; i++){for(j=1; j<=n; j++){printf("*"); }
7. for(i=1; i<=n; i++){for(j=1; j<=m; j++){printf("*");}}
8. for(i=1; i<=n*n; i++) { printf("*"); }
9. for(i=1; i<=n; i++){for(j=1; j<=n; j++){for(k=1; k<=n; k++){ printf("*"); }}
10. for(i=1; i<=n; i++){printf("*");}for(j=1; j<=n*n; j++){printf("#");}
11. for(i=1; i<=n; i=i*2){printf("*");}

Values of i: 1, 2, 4, 8, 16... Loop doubles each iteration. Number of iterations: logn

12. for(i=n; i>=1; i=i/2){printf("*");}

Iterations: logn

13. for(i=1; i<=n; i++){for(j=1; j<=i; j++){printf("*"); }}

Operations: $1+2+3+\cdots+n=n(n+1)/2$

Asymptotic Notations

Definition: Asymptotic notation is a mathematical tool used to describe the efficiency of algorithms in terms of their time or space complexity, focusing on their behavior as the input size grows, in worst or best cases.

There are mainly three asymptotic notations:

- ***Big-O Notation (O-notation)***
- ***Omega Notation (Ω-notation)***
- ***Theta Notation (Θ-notation)***

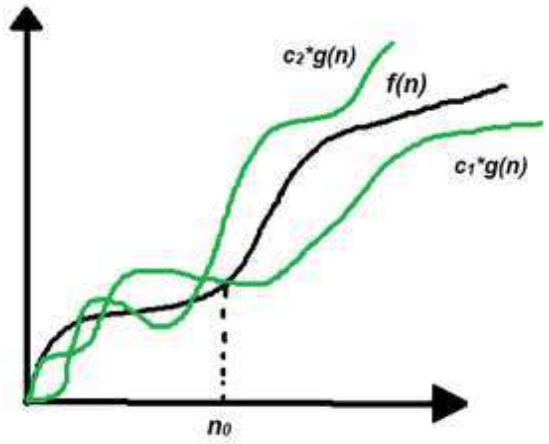
1. Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below.

Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Let g and f be the function from the set of natural numbers . The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n_0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n_0.

Mathematically: *Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n_0 such that 0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n_0}*



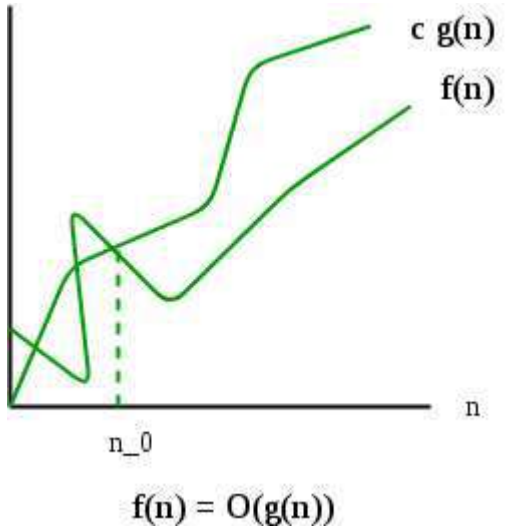
2. Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$

Mathematically :

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

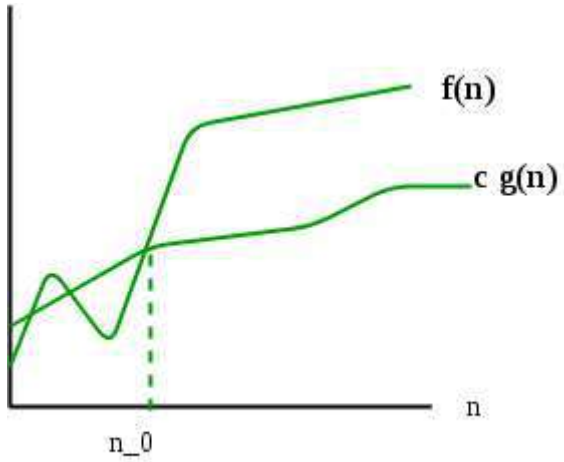


3. Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Def: Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant c > 0 and a natural number n_0 such that c*g(n) ≤ f(n) for all n ≥ n_0

Mathematically: $\Omega(g(n)) = \{ f(n)$: *there exist positive constants c and* n_0 *such that* $0 \leq cg(n) \leq f(n)$ *for all* $n \geq n_0 \}$



f(n) = Omega(g(n))

Lecture 9: Recursion Basics

Definition

Recursion is a programming technique where a function calls itself to solve a smaller version of the same problem.

```
Function()
{
   if(base condition)
      stop;
   Function();
}
```

The function keeps calling itself until a stopping condition is reached.

Every recursive function must have:

1. Base Case

Stops recursion.

2. Recursive Case

Calls itself.

Without a base case, recursion never ends.

Ex:

```
Function Demo(n)
 IF n == 0
Stop
Print n
Demo(n-1)
```

Components of Recursion

Component 1: Base Case

The condition that stops recursion.

Example:

```
if(n==0)
   return;
```

Purpose: Prevent infinite recursion.

Component 2: Recursive Case

The function calls itself.

Example:

```
fun(n-1);
```

Purpose: Reduce the problem size.

What Happens Without Base Case?

Example:

```
void fun()
{
   printf("Hello");
   fun();
}
```

- Never stops. Eventually causes: **Stack Overflow**
- Program crashes.

Ex:

```
void display(int n)
 {
if(n==0)
 return;
printf("%d ",n);
display(n-1);
 }
```

Ex:

```
void fun(int n) { if(n==0) return; fun(n-1); printf("%d ",n); }
```

Printing before recursive call:

Printing after recursive call:

Applications of Recursion

Mathematics

- Factorial
- Fibonacci

Data Structures

- Trees
- Graphs

Algorithms

- Merge Sort
- Quick Sort
- Binary Search

Artificial Intelligence

- Game Trees
- Search Algorithms

Factorial Using Recursion

Introduction to Factorial

Suppose 3 students are standing in a line.

How many different ways can they stand?

Students:A B C

Possible arrangements:

ABC

ACB

BAC

BCA

CAB

CBA

Total arrangements = 6

This is calculated using factorial. 3!=6

Applications of Factorial

Mathematics

Permutations

Combinations

Computer Science

Recursion examples

Algorithm analysis

Probability calculations

Data Science

Arrangement problems

Mathematical Definition of Factorial

The factorial of a positive integer n is the product of all positive integers from 1 to n.

Notation: $n!$

Formula : $n! = n \times (n-1)!$

Special Case: $0! = 1$

Recursive Algorithm for Factorial

Recursive Definition

Base Case $0! = 1$

Recursive Case $n! = n \times (n-1)!$

Factorial(n)

IF n = 0

return 1

ELSE

return n × Factorial(n-1)

```
#include<stdio.h>

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

int main()
{
   int n=5;

   printf("%d",factorial(n));

   return 0;
}
```

Complexity Analysis

Time Complexity

Algorithm:

return n * factorial(n-1);

Each call performs: One multiplication, One recursive call

Time complexity O(n)

Space Complexity

Each recursive call occupies memory.

Call Stack: Stack Depth: n+1

factorial(5)

factorial(4)

factorial(3)

factorial(2)

factorial(1)

factorial(0)

Space Complexity O(n)

Iterative Approach

```
int fact=1;
for(i=1;i<=n;i++)
{
    fact=fact*i;
}
```

Time: O(n)

Space:O(1)

Lecture 11: Fibonacci Series Using Recursion

A sequence in which each number is obtained by adding the previous two numbers.

Series: 0 1 1 2 3 5 8 13 21 34 ...

Term	Value
F(0)	0
F(1)	1
F(2)	1
F(3)	2
F(4)	3
F(5)	5
F(6)	8

F(2)=F(1)+F(0)=1+0=1

F(3)=F(2)+F(1)=1+1=2

F(4)=F(3)+F(2)=2+1=3

F(5)=F(4)+F(3)=3+2=5

Mathematical Definition

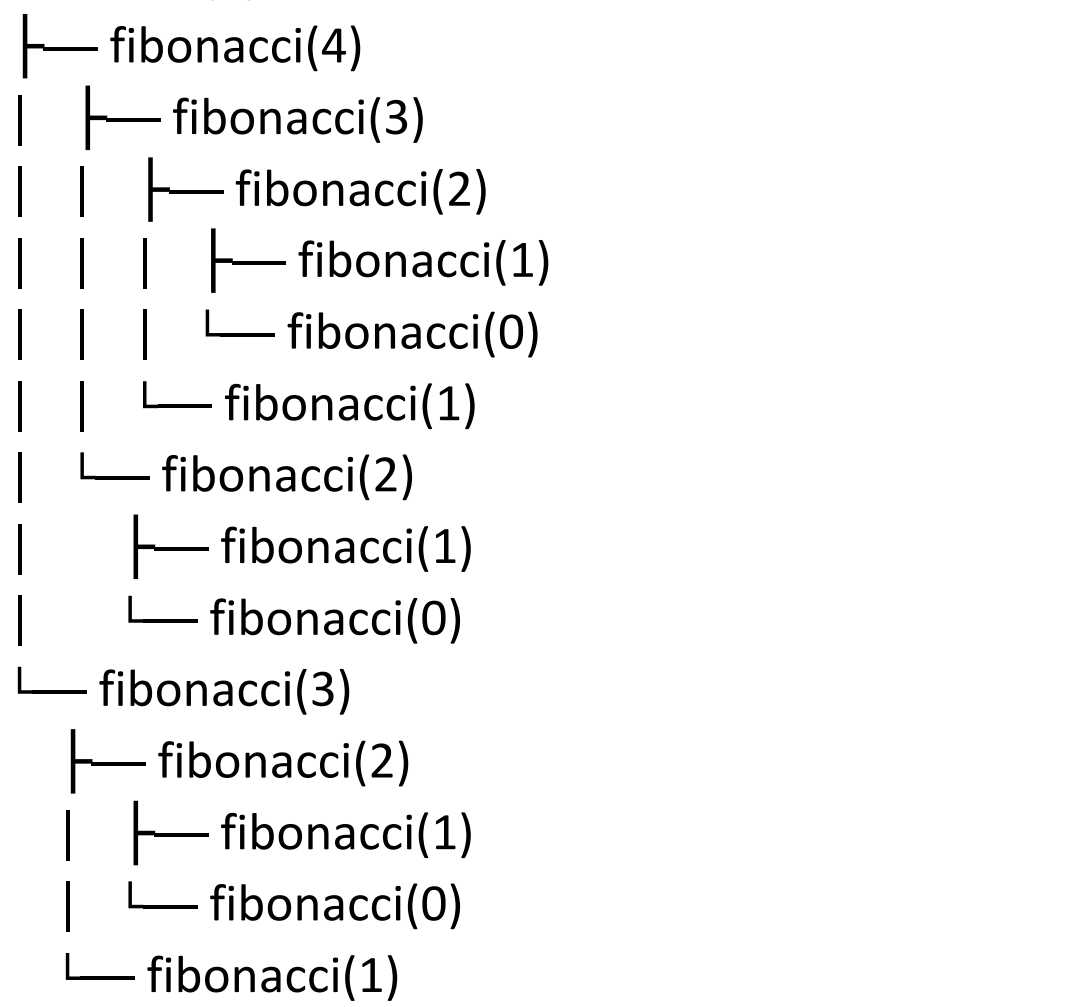
Recursive Formula $F(n)=F(n-1)+F(n-2)$

Base Cases $F(0)=0$ $F(1)=1$

Algorithm

```
Fibonacci(n)
IF n = 0
   return 0
ELSE IF n = 1
   return 1
ELSE
   return Fibonacci(n-1) + Fibonacci(n-2)
```

```c
#include <stdio.h>
// Recursive function to find Fibonacci number
int fibonacci(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
int main()
{
    int n, i;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printf("Fibonacci Series:\n");
    for (i = 0; i < n; i++)
    {
        printf("%d ", fibonacci(i));
    }
    return 0;
}
```



Complexity

$T(n)=T(n-1)+T(n-2)+1$

Time Complexity: O(2^n) (many repeated calculations)

Space Complexity: O(n) (recursive call stack)

Tower of Hanoi

A Puzzle Game

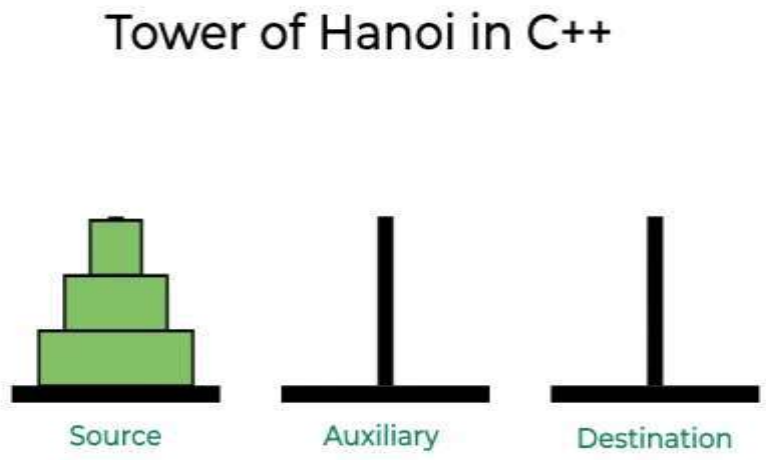
Imagine three rods and several disks of different sizes. Initially, all disks are stacked on the first rod.

Goal: Move all disks from the first rod to the thi rod.

-Smallest disk is on top.

-Largest disk is at the bottom.

- The Tower of Hanoi puzzle was invented by French mathematician:
- **Édouard Lucas**
- in 1883.



Problem Statement

Given: Three rods, n disks of different sizes

Move all disks from: Rod A → Rod C using Rod B as helper.

Rules of Tower of Hanoi

Rule 1

Move only one disk at a time.

Rule 2

Only the top disk can be moved.

Rule 3

A larger disk cannot be placed on a smaller disk.

Understanding the Recursive Idea

Example: 1 Disk

A → C

Moves Required: 1

Example: 2 Disks

Initial

A

1

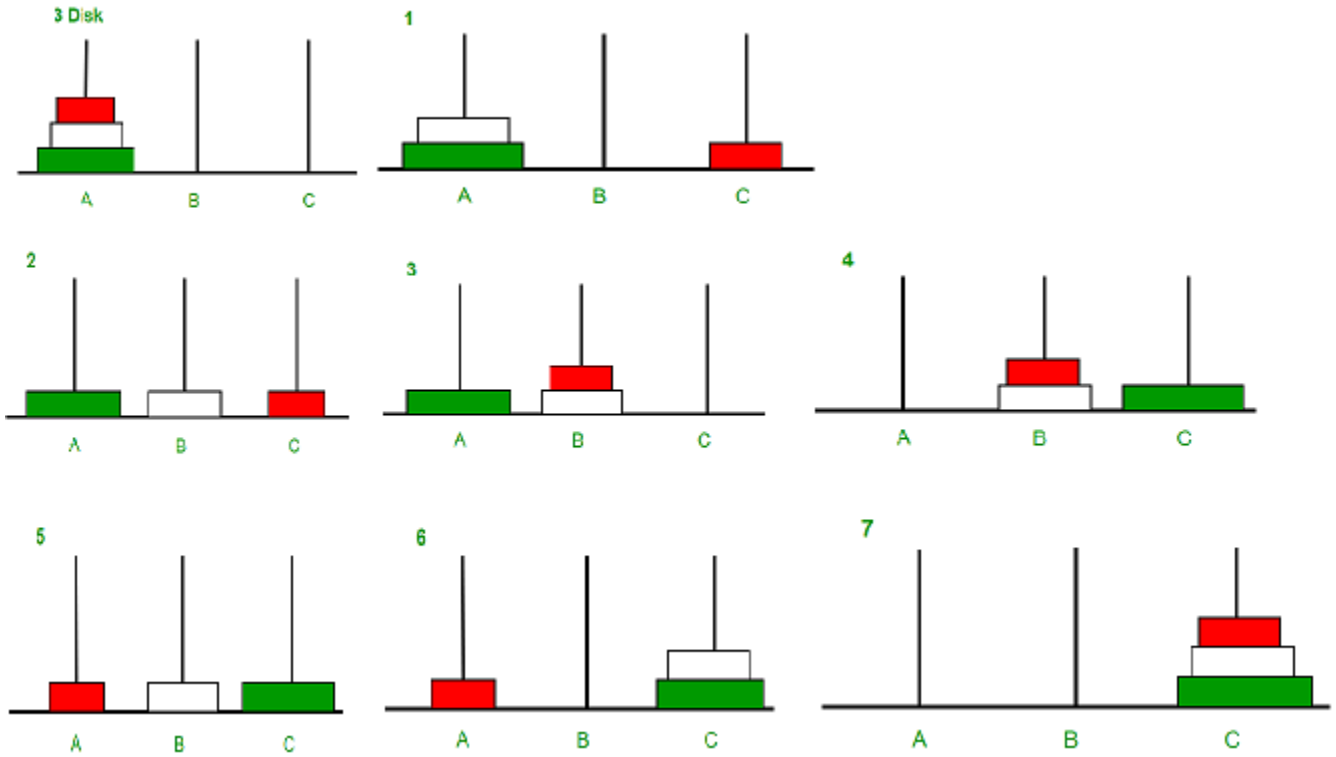
2

Move 1: Move Disk 1 A → B

Move 2: Move Disk 2 A → C

Move 3: Move Disk 1 B → C

Total Moves: 3



To move Disk 3:
Step 1
Move first 2 disks from A to B.
Step 2
Move Disk 3 from A to C.
Step 3
Move 2 disks from B to C.

Notice: To solve 3 disks,

we solve the problem of 2 disks twice.

This is recursion.

Recursive Strategy

To move n disks:

Step 1: Move n−1 disks

Source → Auxiliary

Step 2:Move largest disk

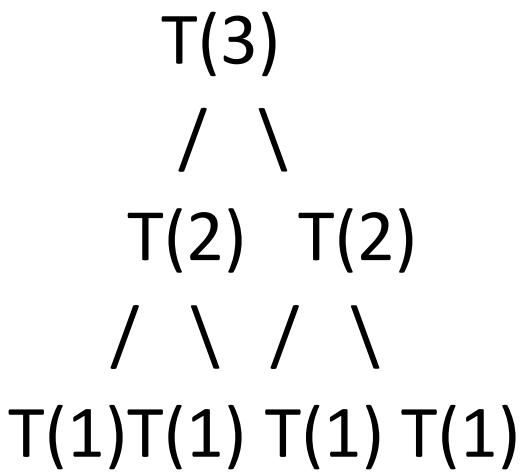
Source → Destination

Step 3: Move n−1 disks

Auxiliary → Destination

```
#include<stdio.h>
void TOH(int n, char source,  char auxiliary, char destination)
{
   if(n==1)
   {
      printf("Move disk 1 from %c to %c\n",source,destination);
      return;
   }
   TOH(n-1, source, destination, auxiliary);
   printf("Move disk %d from %c to %c\n",n,source,destination);
   TOH(n-1, auxiliary, source,destination);
}
int main()
{
   TOH(3,'A','B','C');
   return 0;
}
```

Recursion Tree



$T(n)=2^n-1$

Time complexity: $O(2^n)$

Space complexity: $O(n)$

Lecture 13: Conversion of Recursion to Iteration

Learning Objectives

By the end of this lecture, students will be able to:

- Understand the difference between recursion and iteration.
- Convert simple recursive algorithms into iterative algorithms.
- Explain why iterative solutions often use less memory.
- Compare the performance of recursive and iterative approaches.
- Convert factorial and other simple recursive problems into iterative form.
- Analyze time and space complexity of recursive and iterative algorithms.

Most recursive algorithms can be converted into iterative algorithms using loops.

Recursion

A function calls itself.

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

Iteration

Uses loops.

```
fact=1;


for(i=1;i<=n;i++)
{
    fact=fact*i;
}
```

Feature	Recursion	Iteration
Uses	Function Calls	Loops
Memory	More	Less
Speed	Slower	Faster
Stack Usage	Yes	No
Code Length	Short	Slightly Longer
Risk of Stack Overflow	Yes	No

Why Convert Recursion to Iteration?

Problem with Recursion

Consider: factorial(1000)

This creates 1000 recursive calls.

Each call occupies stack memory.

Consequences

More memory consumption

Function call overhead

Slower execution

Stack overflow for very large inputs

Advantages of Iteration

- Less Memory :Uses only loop variables.
- Faster Execution: No repeated function calls.
- Better Performance: Suitable for large inputs.

Converting Factorial Recursion to Iteration

Recursive Factorial

Mathematical Formula $n!=n\times(n-1)!$

Base Case: $0!=1$

Recursive Algorithm

Factorial(n)

IF n=0

return 1

ELSE

return n × Factorial(n−1)

Recursive C Program

```
int factorial(int n)
{
   if(n==0)
      return 1;

   return n * factorial(n-1);
}
```

4 × factorial(3)

3 × factorial(2)

2 × factorial(1)

1 × factorial(0)

1

Converting to Iteration

Observe: $4! = 1 \times 2 \times 3 \times 4$

Instead of recursive calls, use a loop.

Iterative Algorithm

```
fact = 1
FOR i = 1 to n
    fact = fact × i
RETURN fact
```

Program

```
int factorial(int n)
 { int fact=1;
for(int i=1;i<=n;i++)
{ fact=fact*i; }
return fact;
}
```

Method	Time	Space
Recursive	O(n)	O(n)
Iterative	O(n)	O(1)

Why Space Improves?
Recursive:
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)

5 stack frames.
Iterative:

fact
i
Only two variables.

More Examples of Conversion

Example 1: Print Numbers

Recursive

```
void print(int n)
 { if(n==0)
return;
print(n-1);
printf("%d ",n);
}
```

O/P:

Iterative

```
for(i=1;i<=n;i++)
{
   printf("%d ",i);
}
```

Example 2: Sum of First n Numbers

Recursive

```
sum(n) {
if(n==0)
return 0;
return n+sum(n-1); }
```

Iterative

```
sum=0;
for(i=1;i<=n;i++)
{
    sum=sum+i;
}
```

Example 3: Fibonacci

Recursive

fib(n)=fib(n-1)+fib(n-2)

Complexity: O(2^n)

Iterative

```
a=0;
b=1;
for(i=2;i<=n;i++)
{
   c=a+b;
   a=b;
   b=c;
}
```

Complexity: O(n)

Important Observation

Not all recursive algorithms are easy to convert.

Examples:

- Tree Traversal
- Tower of Hanoi
- Graph Traversal

often require an explicit stack.

Lecture 14: Greedy Method

Learning Outcomes

By the end of this lecture, students will be able to:

- Define an algorithm paradigm.
- Explain the Greedy Method.
- Understand the greedy choice property.
- Identify problems suitable for greedy algorithms.
- Solve the Coin Change Problem using the greedy approach.
- Analyze the time complexity of greedy algorithms.
- Compare greedy algorithms with brute-force and dynamic programming approaches.
- Apply the greedy strategy to real-life problems.

Introduction to Algorithm Paradigms

An **algorithm paradigm** is a general strategy or approach used to design algorithms for solving problems.

- Think of it as a **problem-solving technique**.
- Instead of creating every algorithm from scratch, we follow a proven strategy.

Everyday Analogy

Suppose you want to travel from Bhubaneswar to Cuttack.

Possible strategies:

Major Algorithm Paradigms

Paradigm	Main Idea
Greedy	Choose the best option at the current step
Divide and Conquer	Divide the problem into smaller parts
Dynamic Programming	Store previously computed results
Backtracking	Try all possibilities and backtrack if needed
Branch and Bound	Eliminate non-promising solutions

Introduction to Greedy Method

Definition: A **Greedy Algorithm** is an algorithm that makes the **best possible choice at the current step** without considering future consequences.

Simple Definition: "Choose the best available option now.“

Why is it Called Greedy?

Imagine you have several chocolates.

A greedy child always takes the biggest chocolate first without thinking about what remains.

Similarly, a greedy algorithm always chooses the locally best option.

Local Optimum vs Global Optimum

Local Optimum: The best choice at the current step.

Global Optimum : The best overall solution.

Important Principle

- Greedy algorithms **never go back** to change previous decisions.
- Once a choice is made, it is permanent.

Characteristics of Greedy Algorithms

A greedy algorithm works well when two important properties are satisfied.

1. Greedy Choice Property

A globally optimal solution can be obtained by making the best local choice at every step.

Example: Coin Change (Indian denominations)

2. Optimal Substructure

The solution to a problem contains optimal solutions to its subproblems.

General Greedy Algorithm

Greedy Algorithm

Start

While problem not solved

Select the best available choice

Include it in solution

Reduce the remaining problem

Stop

Example of Greedy Thinking

Suppose you have to collect fruits.

Available: Apple, Orange, Mango, Banana

Rule: Always choose the largest fruit first.

This is a greedy strategy.

Coin Change Problem

Given an amount of ***n*** rupees and an unlimited supply of coins or notes of denominations {1, 2, 5, 10}. we have to find the minimum number of coins required to make up the given amount.

Problem Statement

Available Coins: 10, 5, 2, 1

Amount: 18

Find the minimum number of coins.

Greedy Strategy Always select the largest possible coin.

Step 1 Remaining Amount: 18

Largest coin: 10

Take: 10

Remaining:8

Step 2 Largest coin ≤ 8 is 5

Take: 5

Remaining: 3

Step 3 Largest coin ≤ 3 is 2

Take: 2

Remaining: 1

Step 4 Largest coin ≤ 1 is 1

Remaining 0

Final Solution

10 + 5 + 2 + 1

Total Coins: 4

Greedy Algorithm for Coin Change

```
CoinChange(amount)
Sort coins in descending order
FOR each coin
  WHILE coin ≤ amount
        Print coin
        amount = amount - coin
```

```
#include<stdio.h>
int main()
{
   int coins[] = {10,5,2,1};
   int amount = 18;
   int i;
   printf("Coins Used:\n");
   for(i=0;i<4;i++)
   {
      while(amount >= coins[i])
      {
         printf("%d ", coins[i]);
         amount = amount - coins[i];
      }
   }
   return 0;
}
```

Q. Amount 27

Q. Amount 39

Q. Amount 43

Q. Amount 57

Time Complexity Analysis

- *Sorting the coins initially takes **O(n log n)** time, where n is the number of coin denominations.*
- *The main loop iterates over each coin denomination once, resulting in a linear time complexity of **O(n)**.*
- *Overall, the time complexity of the greedy algorithm for the coin change problem is **O(n log n)** due to the sorting step.*

Space complexity: O(1)

Advantages of Greedy Algorithms

✓ Easy to understand

✓ Easy to implement

✓ Fast execution

✓ Low memory usage

✓ Often produces optimal solutions

✓ Efficient for many optimization problems

Limitations of Greedy Algorithms

Greedy algorithms do not always produce the optimal solution.

They work correctly only when the greedy-choice property holds.

Counter Example

Available coins: 4 3 1

Amount: 6

Greedy Solution

Choose: 4 + 1 + 1

Coins used: 3

Better Solution

3 + 3

Coins used: 2

Observation: The greedy approach fails for this coin system.

Real-Life Applications

1. ATM Cash Dispensing

An ATM usually dispenses the highest denomination notes first.

2. Giving Change in Shops

Cashiers typically return the largest denomination first.

3. Network Routing

Routers often make locally optimal forwarding decisions.

4. Data Compression

The Huffman Coding algorithm is based on the greedy method.

5. Minimum Spanning Tree

Algorithms such as Prim's Algorithm and Kruskal's Algorithm use greedy strategies.

6. Job scheduling algorithm

7. Fractional Knapsack problem

8. Shortest path in Weighted Graph [Dijkstra’s]

9. Priority Queues

10. Sorting: Selection sort, Topological sort

Short Questions

1. Define the Greedy Method.
2. What is a locally optimal choice?
3. What is the greedy-choice property?
4. State two applications of greedy algorithms.
5. What is the time complexity of the greedy coin change algorithm?

Long Questions

1. Explain the Greedy Method with an example.
2. Solve the Coin Change Problem using the greedy approach.
3. Explain the advantages and limitations of greedy algorithms.
4. Write the greedy algorithm and C program for the Coin Change Problem.

Quiz

Q1. A greedy algorithm always chooses:

A) A random choice

B) The locally optimal choice

C) The smallest choice

D) The last choice

Q2. Which of the following is a greedy algorithm application?

A) Huffman Coding

B) Tower of Hanoi

C) Bubble Sort

D) Linear Search

Q3. The greedy method always guarantees the optimal solution.

A) True

B) False

Huffman Coding (Greedy Algorithm)

Introduction to Data Compression

Motivation

Suppose you want to send a file over the Internet.

Original file size: 100 MB After compression: 35 MB

Advantages:

Faster transmission

Less storage

Lower bandwidth usage

What is Data Compression?

Data compression is the process of reducing the number of bits required to represent data.

Types of Compression

1. Lossless Compression

Original data can be recovered exactly.

No information is lost.

Examples:

ZIP files

PNG images

PDF documents

Huffman Coding

2. Lossy Compression

Some information is discarded to reduce file size.

Examples:

JPEG images

MP3 audio

MPEG video

Huffman Coding

Huffman Coding is a **lossless data compression algorithm** that assigns variable-length binary codes to characters based on their frequencies.

- Frequently occurring characters receive **shorter codes**, while infrequent characters receive **longer codes**.
- Huffman Coding was proposed by David A. Huffman in 1952.

Consider the text: AAAAABBC

Frequency table:

Character	Frequency
A	5
B	2
C	1

A occurs most frequently.

Therefore, A should receive the shortest binary code.

Greedy Idea

At every step:

Combine the two nodes having the smallest frequencies.

This locally optimal decision ultimately produces an optimal Huffman Tree.

Therefore, Huffman Coding is a Greedy Algorithm.

Prefix Codes

A prefix code is a coding scheme where no code is the prefix of another code.

This property ensures that encoded data can be decoded unambiguously.

Valid Example

Character	Code
A	0
B	10
C	110
D	111

No code begins with another complete code.

Invalid Example

Character	Code
A	0
B	01

Here, 0 is a prefix of 01.

Ex: Encoded data: 0110111

The decoder cannot uniquely determine where one code ends and another begins.

Constructing a Huffman Tree

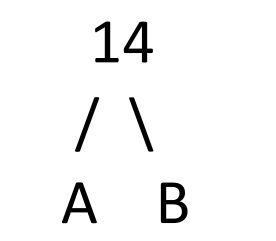
Example

Character:Frequency A:5, B:9, C:12, D:13, E:16,F:45

Step 1: Arrange frequencies in ascending order: 5 9 12 13 16 45

Step 2 : Combine the two smallest frequencies. 5 + 9 = 14

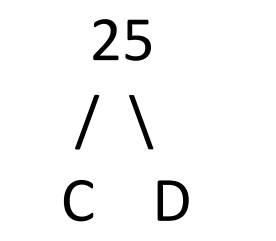
Tree:



Remaining: 12 13 14 16 45

Step 3: Combine: 12 + 13 = 25

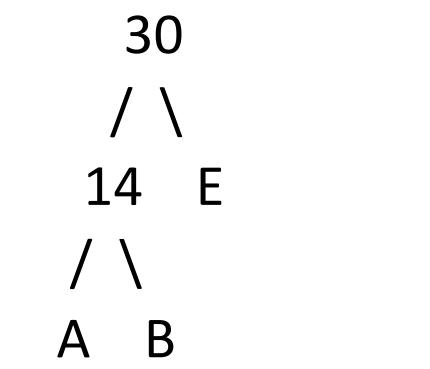
Tree:



Remaining: 14 16 25 45

Step 4: Combine: 14 + 16 = 30

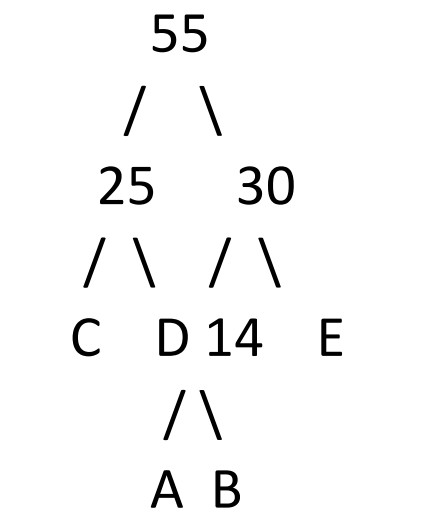
Tree:



Remaining: 25 30 45

Step 5: Combine: 25 + 30 = 55

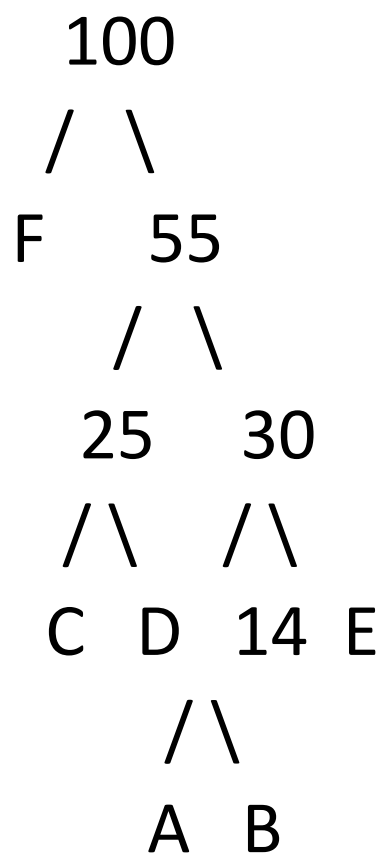
Tree:



Remaining: 45 55

Step 6 Combine: 45 + 55 = 100

Final Huffman Tree:



Assign Binary Codes

Rule: Left edge = 0 ,Right edge = 1

Character	Code
F	0
C	100
D	101
A	1100
B	1101
E	111

Observation

Character F has the highest frequency.Therefore,F receives the shortest code.This minimizes the total number of bits required.

Encoding Suppose the message is: FACE

Using the generated codes:

Character	Code
F	0
A	1100
C	100
E	111

Encoded message: 01100100111

Q. Constrcut Huffman tree

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

Huffman Algorithm

Algorithm

1. Create a leaf node for every character.
2. Insert all nodes into a min-priority queue.
3. While more than one node exists:
    Remove two nodes with minimum frequency.
    Create a new internal node.
    Frequency = Sum of both frequencies.
    Insert the new node into the queue.
4. Remaining node becomes the root.
5. Traverse the tree.
6. Assign:
    Left edge = 0
    Right edge = 1

Why is Huffman Greedy?

At every iteration, the algorithm combines the two least frequent nodes.

This is the locally optimal choice.

Repeated local optimal choices produce the globally optimal Huffman Tree.

Complexity Analysis

Assume: n = number of unique characters.

Using a min-priority queue (min-heap):

Building the Tree

Each insertion and deletion takes: O(logn)

Performed approximately n times.

Overall complexity: O(nlogn)

Space Complexity

Tree and priority queue require: O(n)

Advantages

✓ Lossless compression

✓ Optimal prefix codes

✓ Efficient storage

✓ Faster transmission

✓ Widely used in compression software

Quiz

Q1 Huffman Coding is:

A. Divide and Conquer

B. Dynamic Programming

C. Greedy Algorithm

D. Backtracking

Q2 In Huffman Coding, the most frequent character gets:

A. Longest code

B. Random code

C. Shortest code

D. Decimal code

Q3 Huffman Coding is:

A. Lossy

B. Lossless

C. Randomized

D. Recursive

Q4 What data structure is commonly used to efficiently construct a Huffman Tree?

A. Stack

B. Queue

C. Min-Heap (Priority Queue)

D. Linked List

Q5 The time complexity of Huffman Tree construction using a min-heap is:

A. O(n)

B. O(log n)

C. $O(n^2)$

D. O(n log n)

Q. i)Construct a Huffman Tree for the following frequencies:

Character	Frequency
A	7
B	5
C	2
D	4
E	9

ii)Generate Huffman codes for each character.

iii)Encode the word BEAD using your generated codes.

Q. A communication system contains the following symbols.

Symbol	P	Q	R	S	T
Frequency	5	9	12	13	16

(a) Construct the Huffman Tree.

(b) Find the Huffman code for each symbol.

Q. Consider the following frequency table.

Character	A	B	C	D
Frequency	40	30	20	10

(a) Construct the Huffman Tree.

(b) Find the Huffman codes.

(c) Encode the word ABCD.

Lecture 16: Fractional Knapsack Problem using Greedy Algorithm

Introduction to the Knapsack Problem

Real-Life Scenario

Suppose you are a traveler carrying a backpack.

The backpack can hold only 50 kg.

You have several valuable items, but you cannot carry everything.

Your objective is:

Carry items so that the total profit is maximum without exceeding the weight limit.

This is known as the Knapsack Problem.

What is a Knapsack?

A knapsack is simply a bag with a limited carrying capacity.

Example:

Bag Capacity = 50 kg

Items available = Gold, Silver, Laptop, Camera

Question:

Which items should you carry to maximize your profit?

Mathematical Representation

Suppose
Capacity = W
Number of items = n

Each item has:
Weight = wi
Profit = pi

Objective:

Maximize total profit such that

$\sum wi \leq W$

Types of Knapsack Problems

Fractional Knapsack	**0/1 Knapsack**
Items can be divided	Items cannot be divided
Greedy Algorithm	Dynamic Programming
Faster	Slower
Gives optimal solution using Greedy	Greedy does not always work

Example

Suppose you have:

Item	Weight	Profit
Gold	10 kg	₹600
Silver	20 kg	₹1000
Diamond	30 kg	₹1200

Bag Capacity: 50 kg

Fractional Knapsack Concept

The Fractional Knapsack Problem allows us to take a fraction of an item if the entire item cannot fit into the knapsack.

Main Idea

Choose items based on Profit/weight called the Profit-to-Weight Ratio (P/W Ratio).

Always select the item with the highest ratio first.

Greedy Principle

At every step:

Choose the item having the maximum Profit/Weight ratio.

This is a locally optimal decision that leads to the optimal solution.

Algorithm Steps

1. Calculate Profit/Weight ratio.
2. Sort items in descending order of ratio.
3. Take the entire item if it fits.
4. If it doesn't fit, take only the required fraction.
5. Stop when the bag becomes full.

Example 1

Capacity = 50 kg

Item	Weight	Profit
A	10	60
B	20	100
C	30	120

Step 1: Calculate Ratio

Item	Weight	Profit	Ratio
A	10	60	6
B	20	100	5
C	30	120	4

Step 2: Sort

Item	Ratio
A	6
B	5
C	4

Step 3: Fill Knapsack

Capacity = 50

Take A completely. Profit: 60

Remaining capacity: 40 kg

Take B completely. Profit: 160

Remaining: 20 kg

Item C weighs 30 kg

Only 20 kg space is available.

Take 20/30 of Item C.

Profit obtained: $160+120\times(20/30)=240$

Final Answer

Item	**Fraction Taken**	**Profit**
A	1	60
B	1	160
C	20/30	240

Total Profit 60+100+80=240

Example 1

Capacity = 60 kg

Item	Weight	Profit
A	10	100
B	20	120
C	30	180
D	40	160

Solution

Step 1: Ratio

Item	Ratio
A	10
B	6
C	6
D	4

Take A. Profit = 100

Remaining = 50

Take B. Profit = 120

Remaining = 30

Take C. Profit = 180

Remaining = 0

Maximum Profit: 100+ 120+180=400

Example 2

Capacity = 15 kg

Item	Weight	Profit
P	5	50
Q	10	60
R	15	45

Solution

Ratio

Item	Ratio
P	10
Q	6
R	3

Take P. Profit = 50

Remaining = 10

Take Q. Profit = 110

Remaining = 0

Maximum Profit: 50+60=110

Example 3

Capacity = 25 kg

Item	**Weight**	**Profit**
A	**15**	**90**
B	**10**	**50**
C	**20**	**100**

Example 4

Capacity = 30 kg

Item	**Weight**	**Profit**
A	10	100
B	20	120
C	30	150

Greedy Algorithm (Pseudocode)

```
FractionalKnapsack(items, capacity)
Calculate Profit/Weight ratio
Sort items in descending order of ratio
profit = 0
FOR each item
    IF item weight <= capacity
        Take complete item
        capacity = capacity - weight
        profit = profit + item profit
    ELSE
        fraction = capacity / weight
        profit = profit + (fraction × item profit)
        BREAK
Return profit
```

Complexity Analysis

The main operations are:

Calculate Ratios : O(n)

Sort Items :O(nlogn)

Select Items :O(n)

Total Time Complexity: O(nlogn)

Space Complexity

Constant Space: O(1)

You sort the given items directly inside the original array.

You only use a few simple variables to track the current weight and total value.

No extra lists or arrays are created.

Linear Space: O(n)

You build a new structure to combine the weight, value, and ratio of each item.

Language built-in sorting tools (like standard libraries in some programming languages) create temporary arrays during execution.

Why Greedy Works?

At every step:

Choose the item giving the highest profit per kilogram.

Since items can be divided, the locally optimal decision always contributes to the globally optimal solution.

Applications

- Cargo loading
- Budget allocation
- Investment planning
- Resource allocation in cloud computing
- Bandwidth allocation
- Job scheduling
- Warehouse management

Question 1

Capacity = 40 kg

Item	Weight	Profit
A	10	80
B	20	100
C	30	120

Find the maximum profit.

Quiz

Q1. Fractional Knapsack uses:

A) Divide and Conquer B) Greedy Algorithm C) Backtracking D) Dynamic Programming

Q2. Items are selected based on:

A) Weight B) Profit C) Profit/Weight Ratio D) Random Choice

Q3. The time complexity after sorting is:

A) $O(n^2)$ B) O(log n) C) O(n log n) D) $O(n^3)$

Q4. Can an item be divided in the Fractional Knapsack Problem?

Answer: Yes.

Q5. Is the greedy algorithm guaranteed to produce the optimal solution for the Fractional Knapsack Problem?

Answer: Yes, because items can be divided and selecting the highest profit-per-unit-weight first always leads to an optimal solution.

Minimum Platform Algorithm

- Given arrival **arr[]** and departure **dep[]** times of trains on the same day, find the minimum number of platforms needed so that no train waits. A platform cannot serve two trains at the same time; if a train arrives before another departs, an extra platform is needed.
- **Note:** Time intervals are in the 24-hour format (HHMM) , where the first two characters represent hour (between 00 to 23) and the last two characters represent minutes

Example: The timetable is as given below, the answer is 3. Otherwise, the railway station will not be able to accommodate all the trains.

Rail	Arrival	Departure
Rail A	0900 hrs	0930 hrs
Rail B	0915 hrs	1300 hrs
Rail C	1030 hrs	1100 hrs
Rail D	1045 hrs	1145 hrs

Solution: Let's take the same example as described above. Calculating the number of platforms is done by determining the maximum number of trains at the railway station at any time.

First, sort all the arrival(*A*) and departure(*D*) times in an array. Then, save the corresponding arrivals anddepartures in the array also. After sorting, our array will look like this:

0900	0915	0930	1030	1045	1100	1145	1300
A	A	D	A	A	D	D	D

Now modify the array by placing 1 for *A* and -1 for *D*. The new array will look like this:

1	1	-1	1	1	-1	-1	-1

Finally make a cumulative array out of this:

1	2	1	2	3	2	1	0

Our solution will be the maximum value in this array. Here it is 3.

Note: If we have a train arriving and another departing at the same time, then put the departure time first in the sorted array.

Example 2:

Input: N=6,

arr[] = {9:00, 9:45, 9:55, 11:00, 15:00, 18:00}

dep[] = {9:20, 12:00, 11:30, 11:50, 19:00, 20:00}

Algorithm

1. **Sort** both arrival and departure arrays independently in non-decreasing order.
2. Use **two pointers**:
    - `i` points to the next arrival to process.
    - `j` points to the next departure to process.
3. Maintain a running count of platforms currently needed:
    - If the next **arrival** time is **less than or equal to** the next **departure** time, a new train arrives before the earliest train departs. Increase the count and advance `i`.
    - Otherwise, the earliest train departs before the next arrival. Decrease the count and advance `j`.
4. Track the **maximum** value of the count at any step. That maximum is the answer.

Time and Space Complexity

- **Precise**
  - Sorting arrivals: **O(n log n)**
  - Sorting departures: **O(n log n)**
  - Single linear sweep with two pointers: **O(n + n)**
  - Total time: **O(n log n + n + n)** which simplifies to **O(n log n)**
- **Simplified**
  - **O(n log n)** time
- **Space Complexity**
  - Sorting in place plus a few counters and pointers
  - **O(1)** extra space

Q1. Using the Greedy Method, find the minimum number of coins required to make an amount of ₹87 using the denominations ₹50, ₹20, ₹10, ₹5, ₹2 and ₹1. Show all the steps and explain why the greedy approach works for these denominations.

Q2. Construct a Huffman Tree for the following characters and frequencies:

Character	A	B	C	D	E
Frequency	5	9	12	13	16

Determine:

The Huffman tree.

The Huffman code for each character.

The total number of bits required to encode the message based on these frequencies.

Q3. Solve the following Fractional Knapsack Problem using the Greedy Method. The capacity of the knapsack is 50 kg.

Item	Weight (kg)	Profit
1	10	60
2	20	100
3	30	120
4	40	160

Calculate:

Profit/weight ratio of each item.

Order in which items are selected.

Fraction of any item taken.

Maximum profit obtained.

Q4. The arrival and departure times of trains at a railway station are given below:

Arrival: 9:00, 9:40, 9:50, 11:00, 15:00, 18:00

Departure: 9:10, 12:00, 11:20, 11:30, 19:00, 20:00

Using the Greedy Method, determine the minimum number of platforms required so that no train has to wait. Show the sorting/merging process and explain the algorithm.

Divide and Conquer

Introduction to Divide and Conquer

Divide and Conquer is an algorithm design paradigm in which a large problem is divided into smaller subproblems, each subproblem is solved, and the solutions are combined to obtain the final solution.

Basic Idea

Break a large problem into smaller problems, solve the smaller problems, and combine their results.

Simple Real-Life Example

Suppose 100 students are standing in a line and you need to find the student with the highest marks.

Instead of checking all 100 students at once:

Divide students into two groups.

Find the highest mark in each group.

Compare the two winners.

The larger value is the final answer.

This follows the Divide and Conquer idea.

Three Fundamental Steps

Divide and Conquer has three major steps:

Step 1: DIVIDE

Break the original problem into smaller subproblems.

Example:

[8 3 7 4 2 6 5 1]

Divide into:

[8 3 7 4] [2 6 5 1]

Then divide again:

[8 3] [7 4] [2 6] [5 1]

Step 2: CONQUER

Solve each smaller subproblem.

Usually, recursion is used.

For very small problems, we use a base case.

Example: [8]

A single element is already sorted.

Step 3: COMBINE

Combine the solutions of the smaller problems to obtain the solution to the original problem.

This step is particularly important in **Merge Sort**.

General Structure

DivideAndConquer(problem)

IF problem is small enough

    Solve directly

ELSE

    Divide problem into subproblems

    Solve each subproblem recursively

    Combine the solutions

RETURN solution

Why Use Divide and Conquer?

Suppose we have a problem of size: n=1,000,000

Solving everything directly may be expensive.

Instead, divide it into smaller problems: 500,000+500,000

Then: 250,000+250,000+250,000+250,000

The problem becomes easier to manage.

Important Characteristics

Divide and Conquer generally involves:

- Recursion
- Smaller subproblems
- Independent subproblems in many cases
- Combining results

Common Examples

1. Merge Sort : O(nlogn)
2. Quick Sort

Average: O(nlogn)

Worst:O(n2)

3. Binary Search: O(logn)

Merge Sort

Merge Sort is a sorting algorithm based on the Divide and Conquer paradigm.

It divides an array into smaller arrays until each subarray contains one element.

Then it merges the subarrays in sorted order.

Example

Consider: 8 3 7 4 2 6 5 1

Divide Phase

First: 8 3 7 4 | 2 6 5 1

Again: 8 3 | 7 4 | 2 6 | 5 1

Again: 8 | 3 | 7 | 4 | 2 | 6 | 5 | 1

Now every subarray has one element.

Conquer and Combine Phase

Merge: 8 | 3 Sorted: 3 8

Merge: 7 | 4 Sorted: 4 7

Merge: 2 | 6 Sorted: 2 6

Merge: 5 | 1 Sorted: 1 5

Next Level

Merge: 3 8 and 4 7 Result: 3 4 7 8

Merge: 2 6 and 1 5 Result: 1 2 5 6

Final Merge 3 4 7 8 and 1 2 5 6

Final: 1 2 3 4 5 6 7 8

Merge Sort Algorithm

```
MERGE-SORT(A,p,r)
1       if p <r
2               q ← (r + p)/2
3               MERGE-SORT(A, p, q )
4               MERGE-SORT(A,q+1,r)
5               MERGE(A, p, q, r)
```

Merge Operation

The Merge() operation combines two sorted arrays.

Example: A = [3, 8] B = [4, 7]

Compare: 3 < 4 Take 3.

Then: 4 < 8 Take 4.

Then: 7 < 8 Take 7.

Finally: 8

Result: 3 4 7 8

- MERGE(A, p,q, r)

```
1  n1←q-p+1
2  n2←r-q
3  create arrays L[1...N1+1] and R[1...N2+1]
4  for i←1 to N1
5          do L[i] ← A[p+i-1]
6  for j ← 1 to n2
7          do R[j] ← A[q+j]
8  L[N1+1] ← ∞
9  R[N2+1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13          do if L[i] ≤ R[j]
14                    A[k] ← L[i]
15                     i ← i+1
16              else A[k] ← R[j]
17                     j ← j+1
```

Merge Sort Complexity

At every level, the total work for merging is approximately: O(n)

Number of levels: $\log_2 n$

Therefore: O(nlogn)

Merge Sort Complexity Table

Case	Time
Best	O(n log n)
Average	O(n log n)
Worst	O(n log n)

Auxiliary space: O(n)

Note: *Merge Sort guarantees O(n log n) time complexity even in the worst case.*

Binary Search

Binary Search is a searching algorithm used on a sorted array.

Instead of checking every element, it repeatedly divides the search interval into two halves.

Therefore, Binary Search is a Divide and Conquer algorithm.

Example

Sorted array: 10 20 30 40 50 60 70 80 90

Search: 70

Step 1

Middle element: 50

Since: 70>50

Ignore the left half.

Search: 60 70 80 90

Step 2

Middle: 70

Found!

Binary Search Steps

1. Find middle element.
2. Compare target with middle.
3. If target == middle:

    Found.
4. If target < middle:

    Search left half.
5. If target > middle:

    Search right half.
6. Repeat.

Recursive Binary Search

```
BinarySearch(A, low, high, key)
IF low > high
    return NOT FOUND
mid = (low + high) / 2
IF A[mid] == key
    return FOUND
ELSE IF key < A[mid]
    return BinarySearch(A, low, mid-1, key)
ELSE
    return BinarySearch(A, mid+1, high, key)
```

Binary Search Example

Array: 5 10 15 20 25 30 35 40

Search: 10

Middle: 20

Since: 10<20

Search left: 5 10 15

Middle: 10

Found.

Binary Search Complexity

Every step eliminates approximately half the elements.

Therefore: $n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow ...$

After k steps: $n/2^k=1$

Therefore: $2^k=n$

$k=\log_2 n$

Hence: O(logn)

Binary Search Complexity

Case	**Complexity**
Best	O(1)
Average	O(log n)
Worst	O(log n)

Recursive implementation auxiliary space: O(logn)

Iterative implementation: O(1)

Quick Sort

Quick Sort is another Divide and Conquer sorting algorithm.

It works by selecting a pivot and partitioning the array around the pivot.

Main Steps

- Select a pivot.
- Partition the array.
- Elements smaller than pivot go to the left.
- Elements greater than pivot go to the right.
- Recursively sort the left part.
- Recursively sort the right part.

Example

Consider: 8 3 7 4 2 6 5 1

Choose: Pivot = 5

Partition: 3 4 2 1 | 5 | 8 7 6

Now recursively sort: 3 4 2 1 and 8 7 6

Left Side

Choose 1 as pivot, for example:

| 1 | 3 4 2

Then continue recursively.

Right Side

Similarly: 8 7 6

can be partitioned around a selected pivot.

Eventually: 1 2 3 4 5 6 7 8

Quick Sort Algorithm

```
QuickSort(A, low, high)
IF low < high
   pivot = Partition(A, low, high)
   QuickSort(A, low, pivot-1)
   QuickSort(A, pivot+1, high)
```

Partition Concept

Suppose: 10 7 8 9 1 5

Choose: Pivot = 5

After partitioning:

1 | 5 | 10 7 8 9

All elements on the left are smaller than the pivot.

All elements on the right are greater than the pivot.

Quick Sort Complexity

Best Case

Pivot divides the array approximately equally.

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

Therefore: $O(n\log n)$

Average Case $O(n\log n)$

Worst Case

If the pivot is repeatedly the smallest or largest element:

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

Therefore: $O(n^2)$

Typical recursive stack space:

Average: $O(\log n)$

Worst: $O(n)$

Comparing the Three Algorithms

Feature	Merge Sort	Quick Sort	Binary Search
Purpose	Sorting	Sorting	Searching
Divide	Array into halves	Partition around pivot	Search interval into halves
Conquer	Sort subarrays	Sort partitions	Search one half
Combine	Merge arrays	Usually no explicit merge	No major combine step
Best	O(n log n)	O(n log n)	O(1)
Average	O(n log n)	O(n log n)	O(log n)
Worst	O(n log n)	$O(n^2)$	O(log n)

Divide and Conquer vs Greedy

Divide & Conquer	Greedy
Divides problem into subproblems	Makes local choices
Usually uses recursion	May or may not use recursion
Solves subproblems	Does not generally reconsider choices
Combines subproblem results in many algorithms	Builds solution incrementally
Merge Sort, Quick Sort, Binary Search	Huffman, Fractional Knapsack

Question Apply Binary Search to find 70 in: 10 20 30 40 50 60 70 80 90

Solution

Initial: 10 20 30 40 50 60 70 80 90

Middle = 50

Since: 70>50

Search right half: 60 70 80 90

Middle = 70

Therefore: 70

FOUND

Number of comparisons depends on the exact midpoint convention; here it takes 2 midpoint comparisons.

Q1. Divide and Conquer consists of:

A. Divide, Sort, Search

B. Divide, Conquer, Combine

C. Search, Divide, Merge

D. Select, Compare, Swap

Q2. Which algorithm uses Divide and Conquer?

A. Merge Sort

B. Bubble Sort

C. Slection Sort

D. Linear Search

Q3. Worst-case complexity of Quick Sort is:

A. O(log n)

B. O(n)

C. O(n log n)

D. $O(n^2)$

Q4. Binary Search requires:

A. Unsorted data

B. Sorted data

C. Random data

D. Linked data only

Q5. Worst-case complexity of Merge Sort is:

A. O(n)

B. O(log n)

C. O(n log n)

D. $O(n^2)$

Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique that solves a complex problem by breaking it into smaller overlapping subproblems, solving each subproblem only once, and storing its result for future use.

Dynamic Programming = Solve + Store + Reuse

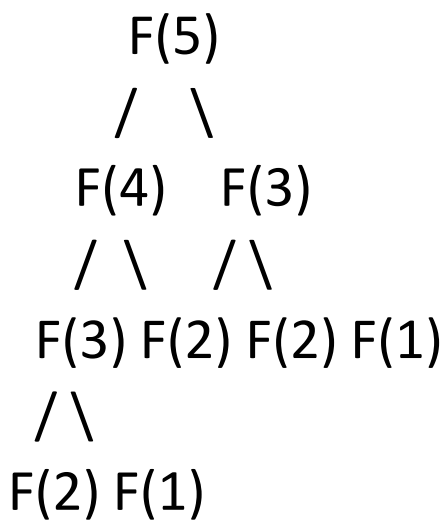
Instead of solving the same problem repeatedly:Solve → Store → Reuse

"We have already studied recursion. If recursion can solve a problem, why do we need Dynamic Programming?"

Consider Fibonacci: F(n)=F(n-1)+F(n-2)

For example: F(5)

The recursive calculation is:



Notice that: F(3) F(2) are calculated multiple times.

This is repeated work.

Dynamic Programming attempts to avoid this repeated computation.

When Do We Use Dynamic Programming?

Dynamic Programming is especially useful when a problem has two properties:

1. Overlapping Subproblems

The same smaller problems are solved repeatedly.

2. Optimal Substructure

An optimal solution to the whole problem can be constructed from optimal solutions to its subproblems.

Recursion vs Dynamic Programming

Recursive Fibonacci

```
Fib(n)
IF n <= 1
    return n
return Fib(n-1) + Fib(n-2)
```

The same values are repeatedly calculated.

Time complexity:$O(2^n)$

Dynamic Programming Fibonacci

Store previously calculated values.

```
fib[0] = 0
fib[1] = 1
FOR i = 2 TO n
    fib[i] = fib[i-1] + fib[i-2]
```

Time complexity: O(n)

Space complexity: O(n)

Thus: $\boxed{O(2^n) \rightarrow O(n)}$

This is the main motivation behind Dynamic Programming.

Optimal Substructure

A problem has optimal substructure if an optimal solution to the complete problem contains optimal solutions to its subproblems.

Example

Suppose we want to find the shortest route from: A → D

and the shortest route is: A → B → C → D

Then the path: B → C → D must also be an optimal path from B to D under the same problem conditions.

This demonstrates optimal substructure.

A problem suitable for Dynamic Programming generally has:

Overlapping Subproblems

+

Optimal Substructure

↓

Dynamic Programming

Two Approaches to Dynamic Programming

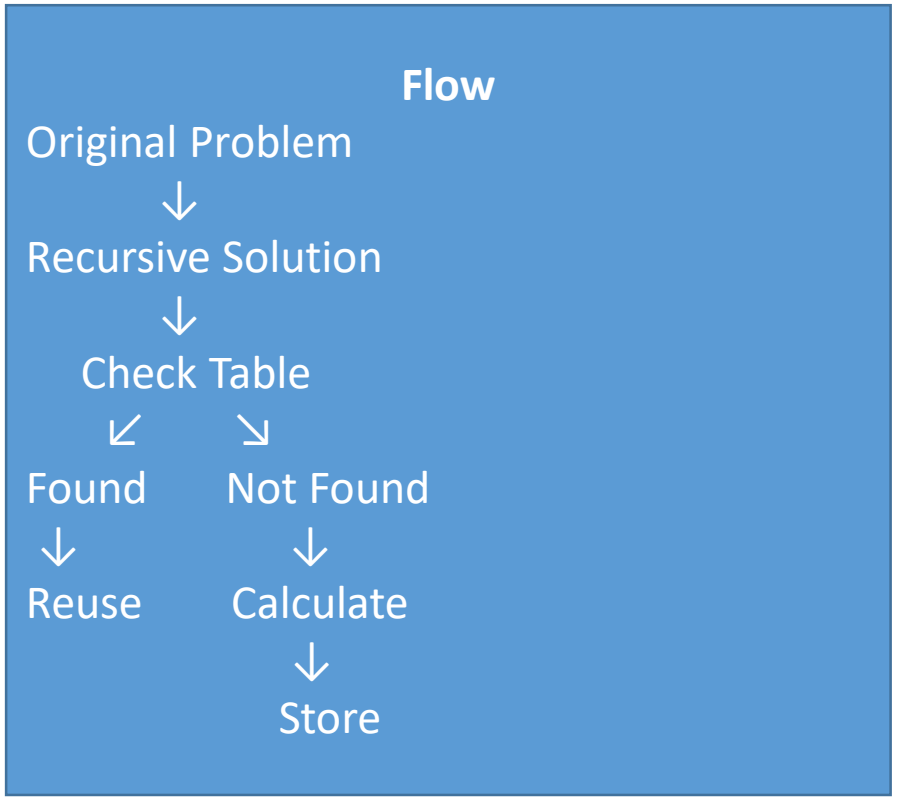
There are two common approaches: **1. Memoization 2. Tabulation**

1. Memoization is a top-down Dynamic Programming approach. It starts with the original problem and recursively solves subproblems.

Whenever a result is calculated, it is stored.

Example: Fibonacci

```
Fib(n)
IF n <= 1
    return n
IF fib[n] is already calculated
    return fib[n]
fib[n] = Fib(n-1) + Fib(n-2)
return fib[n]
```



2. Tabulation

Tabulation is a bottom-up Dynamic Programming approach.

We start from the smallest subproblems and gradually build the solution.

Fibonacci

Initialize:

fib[0] = 0

fib[1] = 1

Then:

fib[2] = fib[1] + fib[0]

fib[3] = fib[2] + fib[1]

fib[4] = fib[3] + fib[2]

and so on.

Example for n = 7

i	fib[i]
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13

Therefore:

F(7)=13

Memoization vs Tabulation

Memoization	Tabulation
Top-down	Bottom-up
Usually recursive	Usually iterative
Computes required subproblems	Builds table systematically
Uses recursion stack	Usually avoids recursion
Easy to derive from recursion	Often efficient in practice

0/1 Knapsack Using Dynamic Programming

A thief has a bag with limited capacity. There are several items.

Each item has: Weight & Profit

The thief can either:

- Take the complete item
- Do not take the item

The item cannot be divided.

Therefore, this is called the: 0/1 Knapsack Problem

Example

Capacity: W=7

Items:

Item	Weight	Profit
1	1	1
2	3	4
3	4	5
4	5	7

Greedy selection based only on profit/weight ratio does not always produce the optimal solution.

Therefore, Dynamic Programming is commonly used.

$$K(i,C) = \max\left[(P_i + K(i-1, C-w_i)), K(i-1, C)\right]$$

$= 0$; $i=0$ or $C=0$

$= K(i-1, C)$; $w_i > C$.

$C = 6$

items	01	02	03
P_i	10	12	28
w_i	1	2	4

Capacity (C) ⟶

items ↓

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	40

$K(1,1) = \max(10 + K(0, 1-1), K(1-1, 1))$

$= \max(10+0, 0) = 10$

$K(1,2) = \max(10 + K(0, 2-1), K(0,2))$

$= \max(10, 0) = 10$

Wednesday

0/1 Knapsack problem

Week 11th • Day (070 ...

$m = 8$ $\quad p = \{1, 2, 5, 6\}$

$n = 4$ $\quad w = \{2, 3, 4, 5\}$

Tabulation Method

P	W	i \ W	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

$$V[i, w] = \max\{V[i-1, w], V[i-1, w - w[i]] + P[i]\}$$

$$V[4, 5] = \max\{V[3, 5], V[3, 5-5] + 6\}$$

$$= 5, 0+6 = (6)$$

$$V[4, 6] = \max\{V[3, 6], V[3, 6-5] + 6\}$$

$$6, 0+6 = (6)$$

Q. Capacity (W) = 7

items	Weight	Profit
1	1	1
2	3	4
3	4	5
4	5	7

$$K(i,c) = \max\left[(P_i + K(i-1, c-\omega_i)),\ K(i-1,c)\right]$$

$= 0$; $i=0$ or $c=0$

$= K(i-1, c)$; $\omega_i > c$

items i \ Capacity c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4	5	6	6	9
4	0	1	1	4	5	7	8	9

$K(1,1) = \max[(1+0), 0] = 1$

Matrix Chain Multiplication

If we have three matrices A, B and C, does the order in which we multiply them affect the amount of computation?

Mathematically: A*(B*C)=(A*B)*C

- The final result is the same. But the number of scalar multiplications can be very different.
- This is the central idea behind Matrix Chain Multiplication.
- **Matrix Chain Multiplication** is a Dynamic Programming problem in which we determine the most efficient way to parenthesize a sequence of matrices so that the total number of scalar multiplications is minimized.

Important:

We are not changing the order of matrices.

We are only deciding: Which matrices should be multiplied first?

Matrix Multiplication Basics

Suppose: $A_{m \times n}$ and $B_{n \times p}$

Then: A x B is possible ,because: $(m \times n) \times (n \times p)$

The resulting matrix has dimensions: $m \times p$

Example

Suppose:

Matrix **A** has dimention=10x20 and

Matrix **B** has dimention=20x30

Then: dimention of AxB produces: 10×30

Number of Scalar Multiplications

For: $A_{m \times n} \times B_{n \times p}$ the number of scalar multiplications required is **$m \times n \times p$**

Example

A: 10x20

B: 20x30

Cost:10x20x30=6000

Therefore 6000 scalar multiplications are required.

Consider three matrices: A_{10x20} B_{20x30} C_{30x40}

There are two possible parenthesizations:

Option 1 (A B)C

Option 2 A(BC)

Both produce the same final matrix: 10x40 but their computational costs are different.

Calculate Cost of (AB)C

First calculate: AB

Cost: 10x20x30=6000

Resulting matrix: 10x30

Now multiply with C:

Cost: 10x30x40 =12000

Total cost: 6000+12000=18000

Calculate Cost of A(BC)

First calculate: BC

Cost: 20x30x40 =24000

Result: 20x40

Now multiply with A: A(BC)

Cost: 10x20x40=8000

Total: 24000+8000 =32000

Matrix multiplication is associative, but the computational cost depends on parenthesization.

What is the Matrix Chain Multiplication Problem?

Suppose we have: A1, A2, A3,….,A_n

We need to find the parenthesization that minimizes the total number of scalar multiplications.

We cannot change: The order of matrices.

We can change: The placement of parentheses.

Example

Given: A × B × C × D

Possible parenthesizations include:

((AB)C)D

(A(BC))D

(AB)(CD)

A((BC)D)

A(B(CD))

For larger numbers of matrices, the number of possible parenthesizations increases rapidly.

Therefore, checking every possibility becomes inefficient.

This is where Dynamic Programming is useful.

Define the DP State

Let M[i][j] represent the minimum number of scalar multiplications needed to multiply matrices A_i through A_j.

Base Case

If: i=j there is only one matrix.

No multiplication is necessary.

Therefore M[i][i]=0

Example: A1

Requires 0 multiplications

Breaking the Chain

Suppose we have $A_i\ A_{i+1}\ldots\ldots A_j$

We can split the chain at any position k: $i \leq k < j$

This produces $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$

Cost of a Split

Suppose: $A_i \ldots A_k$

has already been calculated.

$A_{k+1} \ldots A_j$ has also been calculated.

Their multiplication requires an additional cost.

If the dimensions are represented by array: p0,p1,…,pn

then the multiplication cost is: $p_{i-1} \times p_k \times p_j$

Recurrence Relation

$$M[i][j] = \min_{i \le k < j} [M[i][k] + M[k+1][j] + p_{i-1} p_k p_j]$$

This is the **key formula** for Matrix Chain Multiplication.

the Formula in Simple Words

remember:

Cost = Left Part + Right Part + Cost of multiplying the two results

Take the minimum over all possible split positions.

Example

Suppose we have four matrices:

A1=10×20

A2=20×30

A3=30×40

A4=40×30

Dimension Array P=[10,20,30,40,30]

Therefore

A1 = P0 × P1 = 10 × 20

A2 = P1 × P2 = 20 × 30

A3 = P2 × P3 = 30 × 40

A4 = P3 × P4 = 40 × 30

DP Table

M[i][j]

Diagonal elements are zero.

	A1	A2	A3	A4
A1	0			
A2		0		
A3			0	
A4				0

Chain Length = 2 First calculate adjacent matrices.

M[1][2] =10×20×30=6000

M[2][3]=20x30x40=24000

M[3][4]=30x40x30=36000

Table:

	A1	A2	A3	A4
A1	0	6000		
A2		0	24000	
A3			0	36000
A4				0

Chain Length = 3

M[1][3] There are two possible splits.

Split 1 A1|(A2A3)

Cost: M[1][1]+M[2][3]+10×20×40

=0+24000+8000=32000

Split 2 (A1A2)|A3

Cost: M[1][2]+M[3][3]+10×30×40

=6000+0+12000= 18000

M[1][3]=min(32000,18000) =18000

Calculate M[2][4]

split 1: A2|(A3A4)

Cost: M[2][2]+M[3][4]+20×30×30

=0+36000+18000=54000

Split 2: (A2A3)|A4

Cost: M[2][3]+M[4][4]+20×40×30

=24000+0+24000=48000

Therefore M[2][4]=min(54000,48000)=48000

Final Calculation: M[1][4]

consider the complete chain: A1A2A3A4

Split 1 A1|(A2A3A4)

Cost: M[1][1]+M[2][4]+10×20×30

=0+48000+6000=54000

Split 2 (A1A2)|(A3A4)

Cost: M[1][2]+M[3][4]+10×30×30

=6000+36000+9000=51000

Split 3 (A1A2A3)|A4

Cost: M[1][3]+M[4][4]+10×40×30

=18000+0+12000=30000

M[1][4]= min(54000,51000,30000)=30000

Final DP Table

	A1	A2	A3	A4
A1	0	6000	18000	**30000**
A2	—	0	24000	48000
A3	—	—	0	36000
A4	—	—	—	0

Therefore, the minimum number of scalar multiplications is: 30000

Finding the Optimal Parenthesization

M[1][4]=30000 which is the minimum cost came from k=3 i.e. (A1A2A3)A4

M[1][3]=18000 which is the minimum cost came from k=2 i.e. (A1A2)A3

So the final optimal parenthesization is: ((A1A2)A3)A4

Matrix Chain Multiplication Algorithm

```
MatrixChainOrder(P, n)
FOR i = 1 TO n
    M[i][i] = 0
FOR length = 2 TO n
    FOR i = 1 TO n-length+1
        j = i + length - 1
        M[i][j] = ∞
        FOR k = i TO j-1
            cost = M[i][k]
                 + M[k+1][j]
                 + P[i-1] * P[k] * P[j]
            IF cost < M[i][j]
                M[i][j] = cost
RETURN M[1][n]
```

Why Do We Start With Small Chains?

- Because larger problems depend on smaller problems.

For example: M[1][4] depends on

M[1][1],M[2][4] or M[1][2],M[3][4] or M[1][3],M[4][4]

Therefore, smaller values must be calculated first.

This is the **bottom-up Dynamic Programming approach**.

Complexity Analysis

- For every pair (i,j), we try different split positions k.
- The standard Matrix Chain Multiplication DP has:
- **Time Complexity** O(n3)
- Space Complexity O(n2) because an n×n DP table is maintained.

Why $O(n^3)$?

There are approximately three nested loops:

FOR chain length

    FOR starting position

        FOR split position

Therefore: $O(n) \times O(n) \times O(n)$

Question 1

Given A1=5×10, A2=10×20, A3=20×5

Find the minimum number of scalar multiplications.

Question 2

Given: A1=10×30, A2=30×5, A3=5×60

Find the optimal parenthesization.

Question 3

Given the following matrices: A1=10×20, A2=20×30, A3=30×40, A4=40×30

Using Dynamic Programming:

- Construct the dimension array.
- Define the DP table.
- Write the recurrence relation.
- Calculate the DP table.
- Find the minimum number of scalar multiplications.
- Find the optimal parenthesization.
- State the time and space complexity.

Longest Common Subsequence (LCS) Using Dynamic Programming

Definition

The Longest Common Subsequence of two sequences is the longest sequence of elements that appears in both sequences in the same relative order, but not necessarily consecutively.

Important: A subsequence is obtained by deleting zero or more elements without changing the order of the remaining elements.

Example

Consider:

String X = ABCDE

String Y = ACE

The LCS is: ACE

Length of LCS = 3

The characters A, C, and E appear in both strings in the same order, although they are not consecutive in ABCDE.

Why Use Dynamic Programming for LCS?

- A simple approach is to generate all subsequences of one string and check which ones occur in the other string.
- However, a string of length mm can have 2m2^m subsequences. Checking all of them becomes computationally expensive for large inputs.
- Dynamic Programming (DP) reduces repeated computation by solving smaller subproblems and storing their results.

Two Important Properties

1. Optimal Substructure

- The optimal LCS solution can be constructed using solutions to smaller LCS subproblems.

2. Overlapping Subproblems

- The same smaller subproblems may be encountered repeatedly when solving the problem recursively.
- Therefore, Dynamic Programming is suitable for solving LCS efficiently.

Formulation of the LCS Problem

Let the two input strings be: X=x1,x2,…,xm and Y=y1,y2,…,yn

Define: DP[i][j]=length of the LCS of X[1...i] and Y[1...j]

In simple terms, DP[i][j] stores the LCS length considering the first i characters of X and the first j characters of Y.

Base Case

If either string is empty, the LCS length is zero.

DP[i][0]=0, DP[0][j]=0

This is because an empty string has no common characters with another string.

Recurrence Relation

- There are two cases:

Case 1: Characters Match

- If the current characters are equal: X[i]=Y[j]

Then, the matching character is included in the LCS.

DP[i][j]=DP[i−1][j−1]+1

We add 1 to the LCS length of the two preceding prefixes.

Case 2: Characters Do Not Match

If the current characters are different: X[i]!=Y[j]

We consider two possibilities:

- Exclude the current character of X.
- Exclude the current character of Y.

We take the maximum of these two possibilities.

DP[i][j]=max(DP[i−1][j],DP[i][j−1])

Complete Recurrence Relation

$$DP[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ DP[i-1][j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(DP[i-1][j], DP[i][j-1]) & \text{otherwise} \end{cases}$$

Important Point: When characters match, move diagonally and add 1. When they do not match, take the maximum of the top and left cells.

Example: Constructing the LCS DP Table

Find the LCS of the following strings:

X = ABCD

Y = ACBD

Step 1: Identify the Input Lengths

Length of X = 4

Length of Y = 4

The DP table will have: $(m+1)\times(n+1)=5\times5$

The extra row and column represent empty prefixes.

Step 2: Initialize the DP Table

Set the first row and first column to zero.

X \ Y	Ø	A	C	B	D
Ø	0	0	0	0	0
A	0				
B	0				
C	0				
S	0				

Step 3: Fill the Table

Compare characters one by one.

Example 1: X[1] = A, Y[1] = A Characters match.

DP[1][1]=DP[0][0]+1=0+1=1

Example 2: X[1] = A, Y[2] = C Characters do not match.

DP[1][2]=max(DP[0][2],DP[1][1]) =max(0,1)=1

Step 4: Completed DP Table

X \ Y	Ø	A	C	B	D
Ø	0	0	0	0	0
A	0	1	1	1	1
B	0	1	1	2	2
C	0	1	2	2	2
D	0	1	2	2	3

Find the LCS Length

The LCS length is stored in the bottom-right cell:

DP[4][4]=3

Therefore, the length of the LCS is 3.

Step 6: Recover the Actual LCS

To find the actual subsequence, trace backward from the bottom-right cell:

At D, the characters match. Include D and move diagonally.

At C and B, the characters do not match. Move toward a cell with the same maximum value.

At C and C, the characters match. Include C and move diagonally.

At A and A, the characters match. Include A.

The characters are collected in reverse order: D, C, A.

Reverse them: LCS=ACD

Algorithm for LCS Using Dynamic Programming

```
LCS(X, Y, m, n)
1. Create a DP table of size (m+1) × (n+1)
2. FOR i = 0 TO m
      DP[i][0] = 0
3. FOR j = 0 TO n
      DP[0][j] = 0
4. FOR i = 1 TO m
      FOR j = 1 TO n
         IF X[i] == Y[j]
            DP[i][j] = DP[i-1][j-1] + 1
         ELSE
            DP[i][j] = MAX(
               DP[i-1][j],
               DP[i][j-1]
            )
5. RETURN DP[m][n]
```

Complexity Analysis

Let:

m= length of the first string

n= length of the second string

Time Complexity

The DP table contains (m+1)(n+1) cells.

Each cell requires constant-time work. O(mn)

Space Complexity

The algorithm stores a DP table of size (m+1)(n+1). O(mn)

Applications of LCS

Application	Use of LCS
File comparison	Identifying common lines or sequences between file versions
Version control systems	Supporting change detection and diff operations
DNA sequence analysis	Finding common patterns in biological sequences
Text comparison	Measuring similarities between two text sequences
Plagiarism detection	Identifying shared sequences of text as one component of similarity analysis

Applications of LCS

Application	Use of LCS
File comparison	Identifying common lines or sequences between file versions
Version control systems	Supporting change detection and diff operations
DNA sequence analysis	Finding common patterns in biological sequences
Text comparison	Measuring similarities between two text sequences
Plagiarism detection	Identifying shared sequences of text as one component of similarity analysis

Q1 .Consider the following two strings:

X = ABCBDAB

Y = BDCABA

Using the Dynamic Programming approach:

- Construct the complete LCS table.
- Find the length of the LCS.
- Trace back the table to obtain one LCS.

Q2. Find the Longest Common Subsequence of:

X = AGGTAB

Y = GXTXAYB

Using Dynamic Programming:

- Construct the DP table.
- Show how each cell is calculated.
- Determine the length of the LCS.
- Trace the table to obtain the actual LCS.
- State the time and space complexity.

Q3.For the following strings:

X = ABCDGH

Y = AEDFHR

Using Dynamic Programming:

- Find the LCS length.
- Construct the DP table.
- Recover the actual LCS from the table.
- Explain the traceback procedure.

Q4. Consider:

X = ABCBDAB

Y = BDCABA

- Construct the LCS DP table.
- Find the length of the LCS.
- Identify two different LCSs, if possible.
- Explain why more than one LCS can exist.

Backtracking

Learning Objectives

By the end of this lecture, students will be able to:

- Define the Backtracking technique.
- Explain the concept of state-space tree.
- Understand candidate generation, constraint checking, and backtracking.
- Differentiate Backtracking from Brute Force and Dynamic Programming.
- Solve the N-Queens problem using Backtracking.
- Understand the Subset Sum problem using Backtracking.
- Write general Backtracking pseudocode.
- Analyze the time and space complexity of Backtracking algorithms.

Introduction to Backtracking

Suppose you are solving a maze. You start from the entrance and choose a path. If you reach a dead end, what will you do?

You will:

Go back to the previous position. Try another path. Continue until you find the destination.

This basic idea is the foundation of Backtracking.

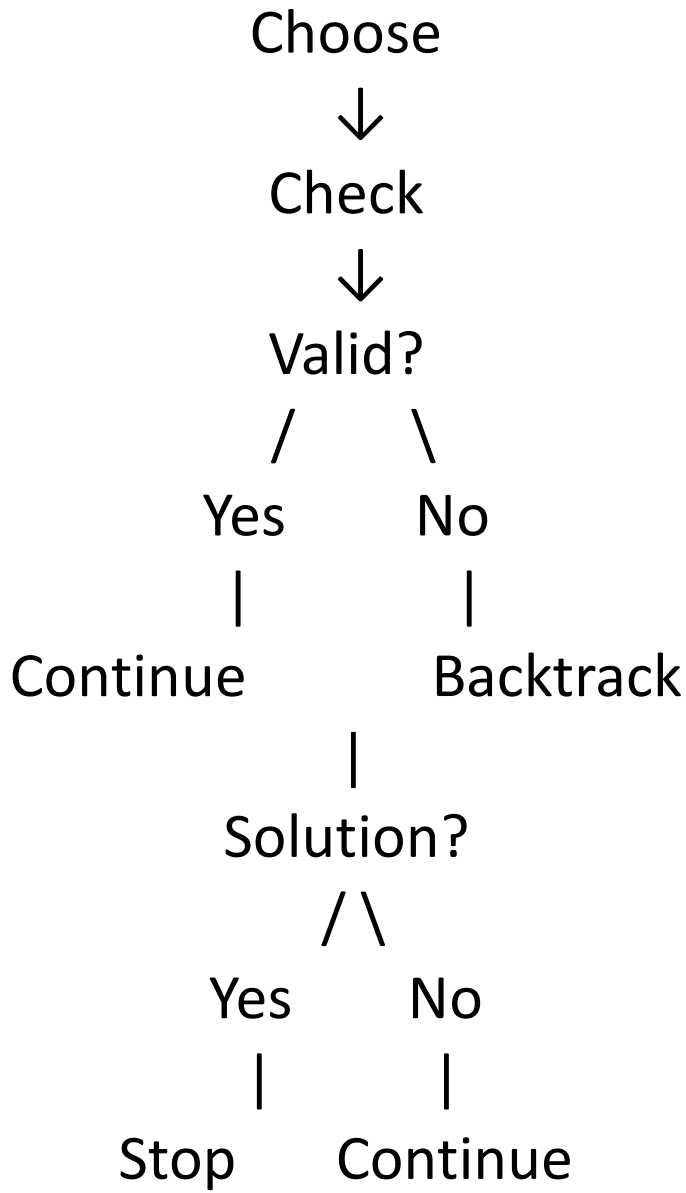
Definition

Backtracking is an algorithmic technique that constructs a solution step by step and abandons a partial solution as soon as it determines that the solution cannot lead to a valid complete solution.

In simple words:

Try → Check → Continue if valid → Go back if invalid.

Basic Strategy





Why Do We Need Backtracking?

Many problems require finding a solution from a large number of possible combinations.

For example:

N-Queens

Sudoku

Maze solving

Graph coloring

Hamiltonian Cycle

Subset Sum

Permutation generation

Crossword puzzles

Trying every possibility blindly can be expensive.

Backtracking improves the search by eliminating invalid partial solutions early.

Backtracking vs Brute Force

Brute Force: Brute force generally tries all possible solutions.

For example, suppose there are:

A

B

C

Possible arrangements include:

ABC

ACB

BAC

BCA

CAB

CBA

A brute-force method may generate all possibilities and then check them.

Backtracking

Backtracking checks constraints while constructing the solution.

If a partial solution is already invalid, it does not generate unnecessary extensions.

Key Difference

Brute Force: Generate first, check later.

Backtracking: Check during generation and abandon invalid paths early.

State-Space Tree

A **state space tree** is a tree-like hierarchical structure where each node represents a specific problem state and each edge represents a valid transition or action taken to move from one state to another

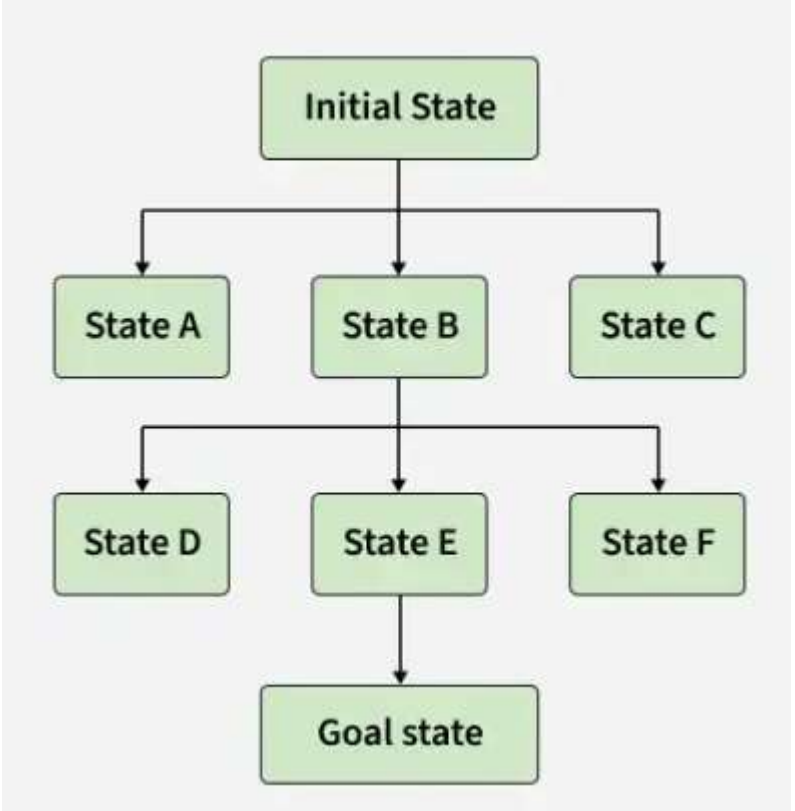
Example

Suppose we want to select elements from:

{A, B, C}

At each step, we can either:

- Include an element
- Exclude an element



Important Terminology

Students should remember these terms:

1. State Current configuration of the problem.

2. State-Space Tree Tree containing possible states.

3. Solution A complete valid configuration.

4. Partial Solution A solution that is not yet complete.

5. Constraint A condition that must be satisfied.

6. Promising Node A node that may still lead to a valid solution.

7. Non-Promising Node A node that cannot lead to a valid solution.

Such a node is pruned.

8. Backtrack Return to the previous decision point and try another choice.

General Structure of Backtracking

The general approach is:

1. Start with an empty solution.
2. Select a candidate.
3. Check whether the candidate is valid.
4. If valid:
    Add it to the solution.
5. Recursively continue.
6. If the solution becomes invalid:
    Remove the candidate.
    Try another candidate.

The important operation is:

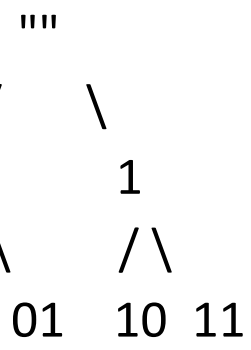
Choose → Explore → Undo

The "undo" operation is what makes Backtracking different from ordinary recursion.

Example of Backtracking

Suppose we want to create a binary string of length 2. At each position, we can choose 0 or 1

State-space tree:



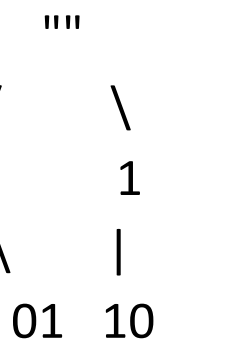
Solutions: 00 01 10 11

There is no pruning here because every generated string is valid.

Now suppose we introduce a constraint:

The binary string must not contain two consecutive 1s.

Then:



The branch 11 is rejected. This is **pruning.**

The N-Queens Problem

The N-Queens problem is one of the most important examples of Backtracking.

Problem

Place N queens on an N × N chessboard such that no two queens attack each other.

A queen can attack another queen if they are in the:

Same row

Same column

Same diagonal

Therefore:

No two queens should share

- the same row,
- the same column,
- or the same diagonal.

Four-Queens Problem

Let's solve:

Place 4 queens on a 4 × 4 chessboard.

Represent a solution as: [2, 4, 1, 3]

Meaning:

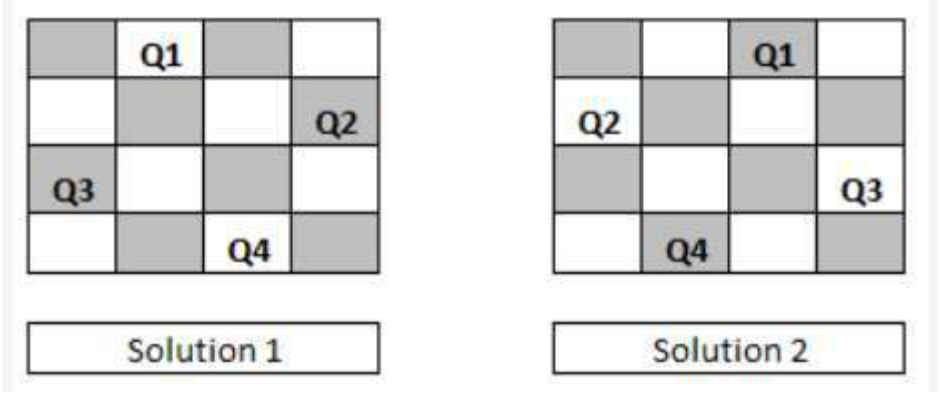
Row 1 → Queen in column 2

Row 2 → Queen in column 4

Row 3 → Queen in column 1

Row 4 → Queen in column 3

Board:



How Backtracking Solves N-Queens

We place queens one row at a time.

For each row:

1. Try column 1.
2. Check whether the position is safe.
3. If safe, place the queen.
4. Move to the next row.
5. If no safe position exists, go back to the previous row.
6. Move the previous queen to another column.
7. Continue.

Key Idea

We do not place all queens randomly and then check.

Instead:

We reject an unsafe position immediately.

N-Queens Safety Conditions

Suppose a queen is being placed at: (row, col)

We must check:

Condition 1: Same Column

No previously placed queen should have the same column. col_i!=col

Condition 2: Same Main Diagonal

Two cells are on the same diagonal if row - col is equal.

Condition 3: Same Secondary Diagonal

Two cells are on the same diagonal if row + col is equal.

N-Queens Backtracking Algorithm

```
NQueens(row)
    IF row > N
        print solution
        return TRUE
    FOR col = 1 TO N
        IF isSafe(row, col)
            place queen at (row, col)
            IF NQueens(row + 1) == TRUE
                return TRUE
            remove queen from (row, col)
    return FALSE
```

Safe function

```
isSafe(row, col)
    FOR each previously placed queen
        IF same column
            return FALSE
        IF same diagonal
            return FALSE
    return TRUE
```

Subset Sum Problem

Another important Backtracking application is Subset Sum.

Problem

Given a set of numbers and a target sum, determine whether there exists a subset whose sum equals the target.

Example:

Set = {3, 4, 5, 2} Target = 9

One possible solution: 4 + 5 = 9

Therefore, a subset exists.

Subset Sum Pseudocode

```
SubsetSum(index, currentSum)
    IF currentSum == target
        return TRUE
    IF index == n
        return FALSE
    IF currentSum > target
        return FALSE
    // Include current element
    IF SubsetSum(index + 1, currentSum + A[index])
        return TRUE
    // Exclude current element
    IF SubsetSum(index + 1, currentSum)
        return TRUE
    return FALSE
```

Example – Subset Sum

Consider: A = {2, 3, 7, 8} Target = 10

Try: 2

Include 2.

Then: 2 + 3 = 5

Then: 5 + 7 = 12

Since: 12 > 10 we backtrack.

Try another branch:

2 + 8 = 10

Therefore:{2,8} is a solution.

Complexity of Backtracking

Backtracking does not have one fixed complexity.

For many combinatorial problems, the worst-case time complexity is exponential.

For example, if each of n elements has two choices:1. Include2. Exclude

then the state-space tree may contain: 2^n possibilities.

For subset type problem $O(2^n)$.

For permutation type problem like N-Queen problem O(n!).

Space Complexity

Backtracking usually uses recursion.

If the maximum recursion depth is \(n\):

Advantages of Backtracking

1. Avoids exploring some unnecessary possibilities.
2. Naturally handles constraint-satisfaction problems.
3. Often easier to implement recursively.
4. Can find one or all valid solutions.
5. Pruning can significantly reduce practical search.

Limitations of Backtracking

1. Worst-case complexity can be exponential.
2. Performance can become poor for large input sizes.
3. Efficiency depends heavily on effective pruning.
4. Designing a good promising function can be difficult.
5. Some problems require exploring a very large state-space tree.

Sorting

Sorting is the process of arranging a collection of elements in a specified order, usually ascending or descending.

Ascending Order (Smallest → Largest)

10, 20, 30, 40, 50

Descending Order(Largest → Smallest)

50, 40, 30, 20, 10

The Sorting Problem

Given: A = [a1, a2, a3, ..., an]

The sorting problem is to rearrange the elements such that:

Ascending: a1<a2<a3…….<an

Descending: a1>a2>a3>…….>an

The elements themselves remain the same; only their positions are changed.

Example

Input: 64 25 12 22 11

Output in ascending order:

11 12 22 25 64

Why Do We Need Sorting?

Sorting is important because it can make other operations easier and faster.

Applications

- Searching
- Database management
- Ranking students
- Employee salary processing
- Finding minimum/maximum values
- Preparing reports
- Data analysis
- Duplicate detection
- Scheduling
- File organization

Example

Searching for 85 in:

10 20 30 40 50 60 70 80 85 90

is easier using Binary Search because the data is sorted.

Stable Sorting

A sorting algorithm is called stable if equal elements maintain their original relative order after sorting.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set.

Example:

Student A: 80

Student B: 70

Student C: 80

If sorted by marks:

Student B: 70

Student A: 80

Student C: 80

Student A still appears before Student C.

In-Place Sorting

- An **in-place sorting algorithm** requires only a small amount of additional memory apart from the input array.

Bubble Sort → Stable , in-place

Insertion Sort → Stable, in-place

Selection Sort → not stable, in-place

Bubble Sort

Bubble Sort repeatedly compares adjacent elements.

If they are in the wrong order, they are swapped.

Example: 5 3

Since 5 > 3 swap them.

3 5

Repeated comparisons gradually move the largest element toward the end.

This is why it is called Bubble Sort.

Bubble Sort Example

Consider: A = [5, 1, 4, 2, 8]

Pass 1 Compare 5 and 1: 5 1 → 1 5

Array: 1 5 4 2 8

Compare 5 and 4: 5 4 → 4 5

1 4 5 2 8

Compare 5 and 2: 5 2 → 2 5

1 4 2 5 8

Compare 5 and 8: No swap.

1 4 2 5 8

At the end of Pass 1: **1 4 2 5 8**

The largest element 8 has reached its final position.

Pass 2 Start: 1 4 2 5 8

Compare 1 and 4: No swap.

1 4 2 5 8

Compare 4 and 2: 4 2 → 2 4

1 2 4 5 8

Compare 4 and 5: No swap.

Final: 1 2 4 5 8

Now 5 is also in its final position. We can see the array is already in sorted order. If we will go for pass 3 then there will be no swap.

Bubble Sort Pseudocode

```
BubbleSort(A, n)
FOR i = 0 TO n-2
    swapped = false
    FOR j = 0 TO n-i-2
        IF A[j] > A[j+1]
            swap(A[j], A[j+1])
            swapped = true
    IF swapped == false
        BREAK
```

Bubble Sort Complexity

Worst Case Consider: 5 4 3 2 1 $O(n^2)$

Average Case $O(n^2)$

Best Case If the array is already sorted 1 2 3 4 5

No swaps occur. O(n)

Space Complexity O(1)

Properties

Property	**Bubble Sort**
Stable	Yes
In-place	Yes
Best	O(n)
Average	$O(n^2)$
Worst	$O(n^2)$
Space	O(1)

Selection Sort

Selection Sort divides the array into two parts

1. Sorted portion 2. Unsorted portion

At each pass:

1. Find the minimum element in the unsorted portion.
2. Swap it with the first element of the unsorted portion.
3. Expand the sorted portion.

Example

Consider A = [64, 25, 12, 22, 11]

Pass 1 Find the minimum element:

64 25 12 22 11

↑

min

Minimum = 11.

Swap 64 and 11: now array becomes 11 25 12 22 64

Sorted portion: 11

Pass 2 Unsorted portion: 25 12 22 64

Minimum = 12.

Swap: 12 and 25

11 12 25 22 64

11 12 | 25 22 64

Pass 3 Unsorted: 25 22 64

Minimum = 22.

Swap:25 and 22

11 12 22 |25 64

Pass 4 Unsorted: 25 64

Minimum = 25.

No effective change:

11 12 22 25 64

Final sorted array: 11 12 22 25 64

Algorithm

```
SelectionSort(A, n)
FOR i = 0 TO n-2
    minIndex = i
    FOR j = i+1 TO n-1
        IF A[j] < A[minIndex]
            minIndex = j
    IF minIndex != i
        swap(A[i], A[minIndex])
```

Property	**Selection Sort**
Stable	Generally No
In-place	Yes
Best	$O(n^2)$
Average	$O(n^2)$
Worst	$O(n^2)$
Space	O(1)

Insertion Sort

Insertion Sort works similarly to the way we arrange playing cards in our hand.

Suppose you receive cards one by one. You keep the cards already received sorted and insert each new card into its correct position.

Example

Cards: 5

Then receive: 3

Insert 3 before 5:

3 5

Then receive: 4

Insert 4 between 3 and 5:

3 4 5

This is the basic idea of Insertion Sort.

Example

Consider A = [5, 3, 4, 1, 2]

Initially: 5 | 3 4 1 2 The first element is considered sorted.

Pass 1 Key = 3

Compare with 5: 5 > 3

Shift 5 right: _ 5 4 1 2

Insert 3: 3 5 4 1 2

Sorted portion: 3 5 | 4 1 2

Pass 2 Key = 4

Compare 4 with 5: 5 > 4

Shift 5: 3 _ 5 1 2

Insert 4: 3 4 5 1 2

Pass 3 Key = 1

Shift larger elements: 3 4 5 1 2

becomes:_ 3 4 5 2

then: 1 3 4 5 2

Pass 4 Key = 2

Shift 5, 4 and 3: 1 _ 3 4 5

Insert 2: 1 2 3 4 5

Final sorted array: 1 2 3 4 5

Algorithm

```
InsertionSort(A, n)
FOR i = 1 TO n-1
    key = A[i]
    j = i - 1
    WHILE j >= 0 AND A[j] > key
        A[j+1] = A[j]
        j = j - 1
    A[j+1] = key
```

Property	Insertion Sort
Stable	Yes
In-place	Yes
Best	O(n)
Average	$O(n^2)$
Worst	$O(n^2)$
Space	O(1)

Merge Sort

Merge Sort is a classic Divide and Conquer sorting algorithm.

Basic Idea

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves.

The important operation is the **merge**.

Example

Consider: A = [38, 27, 43, 3, 9, 82, 10]

Step 1 – Divide [38, 27, 43, 3, 9, 82, 10]

Divide into: [38, 27, 43] [3, 9, 82, 10]

Further divide:

[38] [27, 43] [3, 9] [82, 10]

Continue:

[38] [27] [43] [3] [9] [82] [10]

Now each subarray contains one element.

A single element is already sorted.

Merge Process

Merge: [27] + [43]

Result: [27, 43]

Merge: [38] + [27, 43]

Compare:

38 and 27 → 27

38 and 43 → 38

remaining → 43

Result: [27, 38, 43]

Similarly:

[3] + [9] → [3, 9]

[82] + [10] → [10, 82]

Then:

[3, 9] + [10, 82]

→ [3, 9, 10, 82]

Finally: [27, 38, 43] + [3, 9, 10, 82]

Result: [3, 9, 10, 27, 38, 43, 82]

Properties

- Stable: **Yes**
- In-place: **Generally No**
- Paradigm: **Divide and Conquer**

Case	**Complexity**
Best	O(n log n)
Average	O(n log n)
Worst	O(n log n)
Auxiliary Space	O(n)

Quick Sort

- Quick Sort is another **Divide and Conquer** sorting algorithm.
- Its central concept is the **pivot**.

Basic Steps

1. Select a pivot.
2. Partition the array around the pivot.
3. Elements smaller than pivot go to the left.
4. Elements greater than pivot go to the right.
5. Recursively sort both sides.

Example

Consider:

A = [10, 7, 8, 9, 1, 5]

Choose: Pivot = 5

After partitioning: [1] 5 [10, 7, 8, 9]

Now 5 is in its correct position.

Recursively sort the right part: [10, 7, 8, 9]

Choose an appropriate pivot and continue partitioning.

Eventually: [1, 5, 7, 8, 9, 10]

Partition Concept

Suppose: A = [8, 3, 7, 6, 2, 5]

Take: Pivot = 5

After partitioning, conceptually:

Elements < 5 | 5 | Elements > 5

For example: [3, 2] | 5 | [8, 7, 6]

The exact arrangement depends on the partition scheme.

After partitioning, the pivot reaches its final position.

Then we recursively sort the two subarrays.

Algorithm

```
QUICKSORT(A, low, high)
    IF low < high
        p = PARTITION(A, low, high)
        QUICKSORT(A, low, p-1)
        QUICKSORT(A, p+1, high)


PARTITION(A, low, high)
    pivot = A[high]
    i = low - 1
    FOR j = low TO high - 1
        IF A[j] <= pivot
            i = i + 1
            swap(A[i], A[j])
    swap(A[i+1], A[high])
    return i + 1
```

Complexity

Best, average= O(nlogn)

Worst=$O(n^2)$

Space=O(n)

Properties

Stable: Generally No

In-place: Yes for common partition implementations

Paradigm: Divide and Conquer

Heap Sort

Heap Sort is based on a data structure called a **Heap**.

A heap is a complete binary tree satisfying the heap property.

There are two main types:

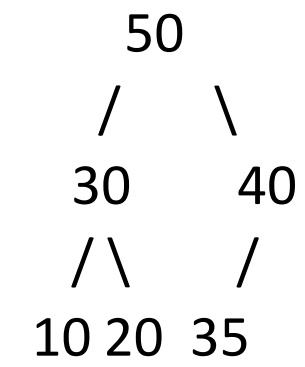
Max Heap Parent is greater than or equal to its children. Parent ≥ Children

Min Heap Parent is less than or equal to its children. Parent ≤ Children

For ascending Heap Sort, we normally use a Max Heap.

Complete Binary Tree Every level is completely filled except possibly the last, which is filled from left to right.

Example:



This is a complete binary tree.

Heap Sort Basic Idea

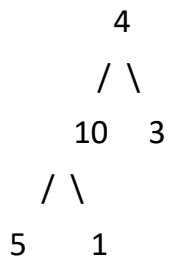
For ascending order:

1. Build a Max Heap. The largest element is at the root.
2. Swap root with the last element.
3. Reduce heap size.
4. Restore the Max Heap property.
5. Repeat.

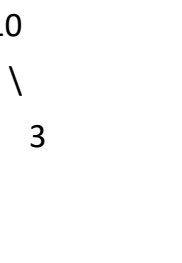
Example

A = [4, 10, 3, 5, 1]

Build complete binary tree



Build a Max Heap:



[10,5,3,4,1]

Largest element = 10.

Swap 10 with the last element i.e 1 . After swapping [1, 5, 3, 4, 10]

Step-by-Step Example

Sort the array [4, 10, 3, 5, 1]:

Build Max-Heap:

Rearrange array to satisfy heap property: [10, 5, 3, 4, 1].

Root node is 10 (the largest element).

Iteration 1: Swap root 10 with last element 1. [1, 5, 3, 4, 10].

Reduce heap size to 4.

Heapify root . Heap becomes [5, 4, 3, 1, | 10].

Iteration 2: Swap root 5 with last unsorted element 1 . [1, 4, 3, 5, 10].

Reduce heap size to 3.

Heapify root. Heap becomes [4, 1, 3, | 5, 10].

Iteration 3 & 4: Continue swapping and heapifying until unsorted size is 1. Final Sorted Array: [1, 3, 4, 5, 10].

HEAPSORT(A, n)

```
BUILD_MAX_HEAP(A, n)
FOR i = n-1 DOWNTO 1
    swap(A[0], A[i])
    HEAPIFY(A, 0, i)
```

Build Max Heap

BUILD_MAX_HEAP(A, n)

```
FOR i = floor(n/2)-1 DOWNTO 0
    HEAPIFY(A, i, n)
```

HEAPIFY(A, i, heapSize)

```
largest = i
left = 2*i + 1
right = 2*i + 2
IF left < heapSize AND A[left] > A[largest]
    largest = left
IF right < heapSize AND A[right] > A[largest]
    largest = right
IF largest != i
    swap(A[i], A[largest])
    HEAPIFY(A, largest, heapSize)
```

Case	**Complexity**
Best	O(n log n)
Average	O(n log n)
Worst	O(n log n)
Space	O(1)

Properties

Stable: No

In-place: Yes

Paradigm: Heap-based sorting

Guaranteed worst-case O(nlogn)

Radix Sort

- Radix Sort is different from the previous comparison-based sorting algorithms.
- Bubble, Selection, Insertion, Merge and Quick Sort primarily determine order through **comparisons**.
- Radix Sort processes numbers **digit by digit**.

Basic Idea of Radix Sort

For decimal numbers, process:

1. Units digit
2. Tens digit
3. Hundreds digit

...

Usually, LSD Radix Sort starts from the least significant digit.

Example

Consider: A = [170, 45, 75, 90, 802, 24, 2, 66]

We process: Units → Tens → Hundreds

Pass 1 – Write the units digit

Number	**Units**
170	0
45	5
75	5
90	0
802	2
24	4
2	2
66	6

Stable sorting by units digit gives: 170, 90, 802, 2, 24, 45, 75, 66

Pass 2 – Tens Digit

Now sort according to the tens digit, while maintaining stability.

Result: 802, 2, 24, 45, 66, 170, 75, 90

Pass 3 – Hundreds Digit

Now sort by hundreds digit.

Result: [2, 24, 45, 66, 75, 90, 170, 802]

Radix Sort Pseudocode

```
RADIX_SORT(A)
    max = largest element in A
    exp = 1
    WHILE max / exp > 0
        COUNTING_SORT_BY_DIGIT(A, exp)
        exp = exp * 10
```

Complexity

Let:

- n = number of elements
- d = number of digits
- k = range of each digit, e.g. 10 for decimal digits

If Counting Sort is used for each digit: O(d(n+k))

Properties

- Stable: **Yes**
- Comparison-based: **No**
- Suitable for: integers/fixed-format keys
- Complexity: O(d(n+k))

Comparison Table of Sorting Algorithms

Feature	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort	Radix Sort
Basic Technique	Repeatedly swaps adjacent elements	Selects minimum/maximum element	Inserts element into sorted portion	Divide and Conquer	Divide and Conquer	Heap-based	Digit-by-digit sorting
Type	Comparison	Comparison	Comparison	Comparison	Comparison	Comparison	Non-comparison
Best Case	$O(n)$[1]	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(d(n+k))$
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(d(n+k))$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(d(n+k))$
Space Complexity	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(\log n)^2$	$O(1)$	$O(n+k)$
Stable?	Yes	No	Yes	Yes	No	No	Yes
In-place?	Yes	Yes	Yes	Generally No	Yes	Yes	No
Suitable for	Small/nearly sorted data	Small data	Small/nearly sorted data	Large data, stability	General-purpose fast sorting	Guaranteed $O(n \log n)$	Integers/fixed-length keys

Searching

- **Searching** is the process of locating a particular element or record in a collection of data.
- The element being searched for is called the **search key**.

Example

Suppose the student roll numbers are: [12, 25, 31, 42, 56, 68, 75]

If we want to find roll number 42, the searching algorithm checks the collection and reports its position or confirms that the key exists.

Common searching techniques include:

- Linear Search
- Binary Search
- Binary Search Tree
- Hashing

The best technique depends on whether the data is sorted, how frequently the data changes, and how quickly searches must be performed.

Symbol tables

- A symbol table is a data structure that stores information associated with keys, allowing the information to be retrieved efficiently using those keys.
- A symbol table is commonly used in compilers, interpreters, assemblers and other software systems.

Example: Symbol table in a compiler

Consider the following declarations in a program:

int age;

float salary;

char grade;

- A compiler may maintain a symbol table containing information such as:

Identifier (Key)	Data Type	Scope	Other Information
age	int	Local	Memory location
salary	float	Local	Memory location
grade	char	Local	Memory location

When the compiler encounters salary, it looks up the symbol table to retrieve information about the variable.

Basic operations of a symbol table

Operation	Description
Search(key)	Finds information associated with a key
Insert(key, value)	Adds a new key-value pair
Delete(key)	Removes an entry
Update(key, value)	Modifies the stored information
Traverse()	Visits all entries

Applications

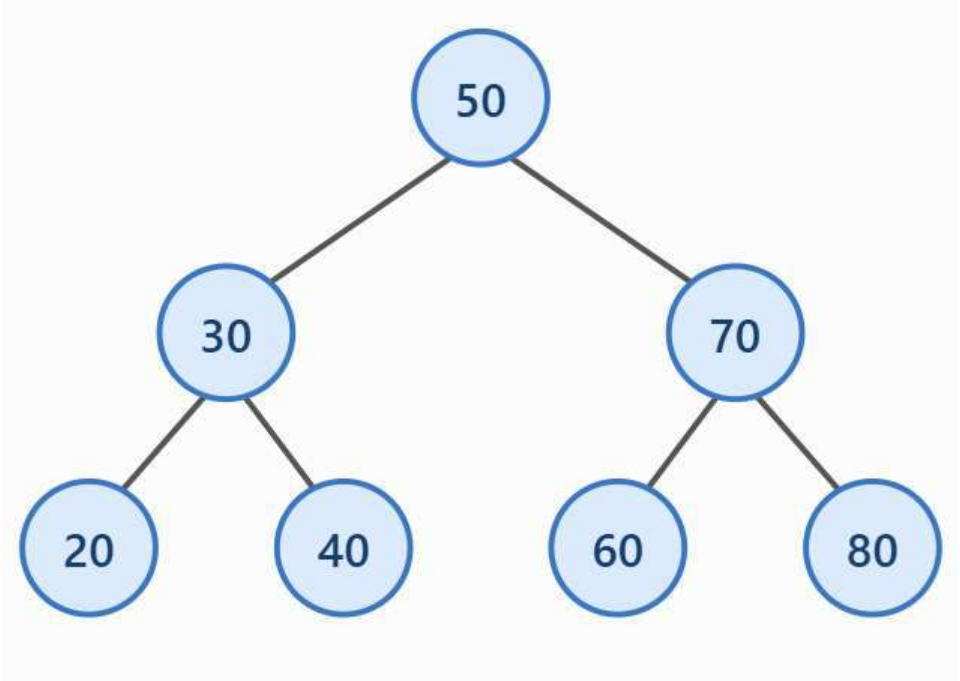
- Compiler symbol management
- Database indexing
- Dictionaries and maps
- Variable and function lookup
- Configuration management
- Name resolution in programming languages

Binary Search Tree (BST)

A Binary Search Tree is a binary tree in which, for every node:

- All keys in the left subtree are smaller than the node's key.
- All keys in the right subtree are greater than the node's key.
- Both left and right subtrees follow the same rule.

Ex: Keys 50, 30, 70, 20, 40, 60, 80



Operations on a BST

1. Searching

Algorithm: BST_Search

```
BST_Search(root, key)
    if root = NULL or root.key = key
        return root
    if key < root.key
        return BST_Search(root.left, key)
    else
        return BST_Search(root.right, key)
```

At each step, one subtree is eliminated from consideration.

- Best case: O(1), when the root contains the key.
- Average case: O(log n) for a reasonably balanced tree under typical assumptions.
- Worst case: O(n), when the tree is highly skewed.

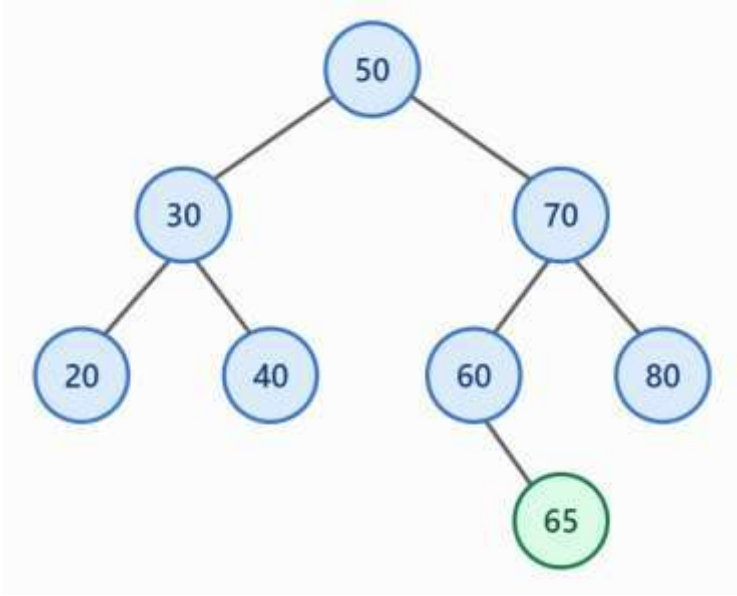
2. Insertion

To insert a key:

1. Start at the root.
2. Compare the new key with the current node.
3. Move left if the key is smaller; move right if greater.
4. Insert the key at the first empty position.

Example: Insert 65 into the previous BST.

- 65 < 50? No, move right.
- 65 < 70? Yes, move left.
- 65 > 60? Yes, move right.
- Insert 65 as the right child of 60.



3. Deletion

Deleting a node from a BST has three cases.

Case 1: Node is a leaf

Delete the node directly.

Example: Delete 20. It has no children, so simply remove it.

Case 2: Node has one child

Delete the node and connect its parent directly to its child.

Case 3: Node has two children

Replace the node with either:

- Its inorder successor: the smallest key in its right subtree, or
- Its inorder predecessor: the largest key in its left subtree.

Then remove the replacement key from its original location.

Example: Delete 50 from the original tree. Its inorder successor is 60. Replace 50 with 60, then remove the original node containing 60.

4. Tree traversal

For the BST containing 50, 30, 70, 20, 40, 60, 80:

Traversal	Visiting order	Result
Inorder	Left, Root, Right	20, 30, 40, 50, 60, 70, 80
Preorder	Root, Left, Right	50, 30, 20, 40, 70, 60, 80
Postorder	Left, Right, Root	20, 40, 30, 60, 80, 70, 50

Balanced Search Tree

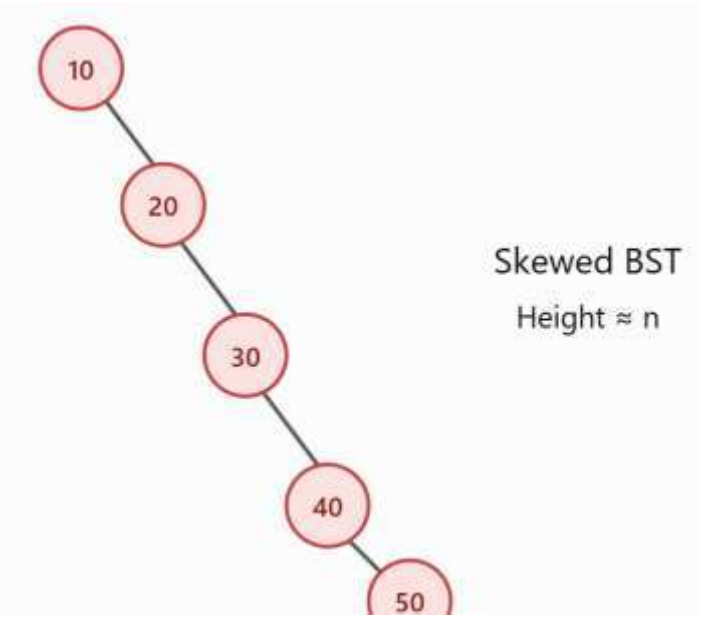
A balanced search tree is a node-based binary search tree (or multi-way tree) that automatically keeps its height small during insertions and deletions, guaranteeing **O(log n)** time complexity for search, insert, and delete operations.

A BST is efficient when its height is small. However, its performance depends on the order in which keys are inserted.

Consider inserting the sorted keys:

10, 20, 30, 40, 50

The resulting BST becomes skewed:



Searching for 50 may require examining every node, giving O(n) time.

A balanced search tree maintains its height within a controlled bound, so searching, insertion and deletion can generally be performed in O(log n) time.

AVL trees

An AVL tree is a self-balancing Binary Search Tree.

For every node, the balance factor must be −1, 0 or +1.

BF = height(left)-height(right)

If the balance factor becomes less than −1 or greater than +1 after insertion or deletion, rotations are used to restore balance.

Four AVL rotation cases

Case	Situation	Correction
LL	Insertion into left subtree of left child	Right rotation
RR	Insertion into right subtree of right child	Left rotation
LR	Insertion into right subtree of left child	Left rotation, then right rotation
RL	Insertion into left subtree of right child	Right rotation, then left rotation

Hashing and Hash Tables

Hashing is a technique that maps a key to an index in an array using a hash function.

- The objective is to support fast insertion, searching and deletion.
- For example, a student record may be stored using the roll number as its key. A hash function determines the location where the record should be placed.

Hash function

- A hash function converts a key into a valid table index.
- A simple hash function for integer keys is: $h(k)=k \bmod m$

where:

k = key

m = hash-table size

h(k) = index in the table

Example

Let the hash-table size be m=10, with keys 23, 35, 42, 54

Using h(k)=k mod 10

Key (k)	Calculation	Hash index
23	23 mod 10	3
35	35 mod 10	5
42	42 mod 10	2
54	54 mod 10	4

The keys are stored at indices 3, 5, 2 and 4, respectively.

Hash tables

A **hash table** is a data structure that stores key-value pairs in an array, using a hash function to determine their locations.

Example: With table size 10 and hash function h(k)=kmod10, inserting keys 23, 35 and 42 gives:

Index	Stored key
0	Empty
1	Empty
2	42
3	23
4	Empty
5	35
6	Empty
7	Empty
8	Empty
9	Empty

Collision in hashing

A collision occurs when two different keys map to the same table index.

Using h(k)=kmod10

h(23)=3,h(33)=3

Both keys map to index 3.

A collision-resolution method is needed to store and retrieve both correctly.

Collision-resolution techniques

1 Separate chaining Each table index stores a linked list or another collection of entries that hash to that index.

Example: Insert keys 23, 33, 43, 12 into a table of size 10 using h(k)=k mod 10.

-23 → index 3

-33 → index 3

-43 → index 3

-12 → index 2

The resulting table is:

2 12→ NULL

3 23→33→43→ NULL

Advantages: Simple collision handling; multiple keys can occupy the same bucket.

Limitation: Requires additional storage for links or bucket collections.

2. Open addressing In open addressing, all entries are stored within the hash table itself.

When a collision occurs, the algorithm probes for another available slot.

A. Linear probing

The next slot is checked sequentially.

h(k)=(h(k)+i)mod m where i=0,1,2,….

Example: Insert keys 23, 33, 43 into a table of size 10 using h(k)=k mod 10.

- 23 hashes to index 3; place it at 3.
- 33 hashes to index 3; occupied, so place it at 4.
- 43 hashes to index 3; indices 3 and 4 are occupied, so place it at 5.

Index	Value
3	23
4	33
5	43

B. Quadratic probing

- Quadratic probing uses quadratic offsets to find another slot.
- A common formula is: $h_i(k)=(h(k)+i^2) \bmod m$

C. Double hashing

- Double hashing uses a second hash function to determine the probe step.

$h_i(k)=(h_1(k)+ih_2(k)) \bmod m$

Complexity comparison

Data structure	Search: Average/Expected	Search: Worst Case	Notes
Unsorted array	$O(n)$	$O(n)$	May be simple for small datasets
Sorted array (binary search)	$O(\log n)$	$O(\log n)$	Insertion can require $O(n)$ shifting
Ordinary BST	$O(\log n)^*$	$O(n)$	Depends on tree shape
Balanced BST	$O(\log n)$	$O(\log n)$	Maintains height bounds
Hash table	$O(1)$ expected	$O(n)$	Depends on hash distribution and collision handling