# Computational Thinking Using Python

CSE1500

[L-T-P-C: 2-0-2-3]

## MODULE – 2

Algorithm Design & Problem-Solving Strategies

PRESIDENCY UNIVERSITY

Private University Estd. in Karnataka State by Act No. 41 of 2013

# Algorithm Design Strategies

Different problems require different approaches.

Some common algorithm design strategies are:

- Brute Force
- Divide and Conquer
- Greedy Method
- Dynamic Programming
- Backtracking
- Branch and Bound

In these we will discuss about Brute fore, Divide and Conquer approaches only.

**Note:** Other approaches will be learned in higher semesters (Design and analysis of algorithms DAA)

## Brute Force

Brute force is the simplest problem-solving technique where all possible solutions are tried until the correct one is found. It does not use optimization or shortcuts.

**Key Features:**

- Straightforward and easy to implement.
- Guarantees a correct solution.
- Often inefficient for large inputs (time-consuming).
- Works well for small problem sizes.

# Applications of Brute Force

**Steps Involved**

1.Generate all possible candidate solutions.

2.Test each candidate against the problem requirements.

3.Choose the candidate(s) that satisfy the conditions.

**Examples:**

**1. Searching**

- **Linear Search**: Checking each element one by one in a list/array until the target is found.
- Used when the dataset is small or unsorted.

**2. String Matching**

- Comparing a pattern with every possible substring in a text.

**3. Traveling Salesman Problem (TSP)**:

- Trying every possible route and picking the shortest one.

# Applications of Brute Force

**Advantages**

- Simple to design and implement.
- Can be applied to a wide range of problems.

**Disadvantages**

- Very slow for large input sizes.
- Not efficient in terms of time and memory.

# Divide and Conquer

**Divide and Conquer:**

Divide and Conquer is a problem-solving technique that breaks a large problem into smaller sub-problems, solves them recursively, and then combines their results to get the final solution.

**Key Features**

- Recursive approach.
- Efficient for large data sets.
- Reduces problem complexity compared to brute force.

# Divide and Conquer

**Steps Involved**

1. **Divide**: Break the problem into smaller sub-problems of the same type.
2. **Conquer**: Solve the sub-problems recursively (if sub-problems are small, solve them directly).
3. **Combine**: Merge the solutions of the sub-problems into a final solution.

**Examples**

- **Binary Search**: Divide the array into halves and search only one half each time.
- **Merge Sort**: Divide the array into halves, sort each half, then merge.
- **Quick Sort**: Partition the array around a pivot and recursively sort sub-arrays.
- **Matrix Multiplication (Strassen's Algorithm)**: Breaking down matrix multiplication into smaller multiplications.

# Binary Search

Suppose we want to search for 25 in the sorted array:
[10, 15, 20, 25, 30, 35, 40]

- **Divide**: Split the array in half. Middle = 25.
- **Conquer**: Compare 25 with middle element.
  - If equal → found.
  - If smaller → search left half.
  - If larger → search right half.
- **Combine**: Here, we found it directly at the middle.

# Merge Sort

- **Divide and Conquer Steps in Merge Sort**:
- **Divide**: Split the array into two halves until each subarray has only one element.
- **Conquer**: Sort each half recursively.
- **Combine**: Merge the sorted halves to get a fully sorted array.

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

**Divide:**

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

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

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

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

## Conquer (Sort each subarray)

- [38] [27] → [27, 38]
- [43] [3] → [3, 43]
- [27, 38] [3, 43] → [3, 27, 38, 43]

- [9] [82] → [9, 82]
- [9, 82] [10] → [9, 10, 82]

# Combine

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

# Divide and Conquer

**Advantages**

•Faster and more efficient than brute force for many problems.
•Can be applied to complex problems like sorting, searching, and optimization.
•Provides a clear recursive structure.

**Disadvantages**

•Recursive calls may use more memory (stack overhead).
•Not always the most efficient if the problem is small.
•Requires careful design to avoid recomputation (may need dynamic programming in some cases).

# Iteration: Loops

Python supports 2 types of iterative statements.
1) while loop 2) for loop

## While loop:

It executes a sequence of statements repeatedly as long as a condition remains true. The syntax of the while loop is given as follows:

```
while test-condition:
    #Loop Body
    statement(s)
```

Here while is a key word and test-condition is a relational or logical expression which results True or False values. Condition must be followed by colon.

# while loop

As long as the test condition is true, the statements (body of the loop) are executed. Once the condition becomes false, the iteration terminates and control continues with the first statement after the while loop.

**Note** that if the condition false very first time then body of the loop would never be executed.

**Example)**

```
# to display numbers from 1 to 5
n=1
while n<=5:
    print(n)
    n=n+1
print(“end of the program")
```

Note : we use indentation to define body of the loop.

# Sum of numbers from 1 to 10

```
i = 1
sum = 0
while i <= 10:
        sum += i
        i += 1
print("Sum =", sum)
```

# Printing even numbers less than 10

```
i = 2
while i < 10:
        print(i)
         i += 2
```

# Multiplication Table:

```
num = int(input("Enter a number: "))

print(f"\nMultiplication Table of {num}")

i = 1
while i <= 10:     # loop from 1 to 10
        print(f"{num} x {i} = {num * i}")
        i += 1
```

# Factorial of a number using while loop

```
num = 5
factorial = 1
while num > 0:
        factorial *= num
        num -= 1
print("Factorial =", factorial)
```

# for loop

The for loops in Python are slightly different from the for loops in other programming languages (such as c and java).
The Python for loop iterates through sequences such as list, tuple, dictionary, set, or a string.
It iterates through each value in a sequence.

The syntax of for loop is given as follows:

```
for variable in sequence:
    statement(s)   # block of code
```

variable -> takes the value of each item in the sequence one by one.
sequence -> a collection (like list, tuple, string, range, dictionary, etc.).
The block of code executes once for each value in the sequence.

# for loop

Example 1) iterating a string using for loop:

```
skill = 'Python’
# iterate over each character in skill
for x in skill:
        print(x)
```

Example 2)  Using range()

```
for i in range(5):
      print(i,end=“:")
```

Output:  0:1:2:3:4

**Note:** By default, range(n) generates numbers from 0 to n-1.

# for loop

Example 3) Custom Range with Step

```
for i in range(10, 20, 2):
    print(i)
```

Output :
10
12
14
16
18

Note: optional keyword else can be used  with while and for loop.

# EVEN AND ODD

```
for i in range(2, 11, 2):
        print(i)

for i in range(1, 10, 2):
    print(i)
```

# Sum of first 10 numbers

```
sum= 0
for i in range(1, 11):
        sum += i
print("Sum =", total)
```

# Factorial of a number using for loop

```
num = 5
factorial = 1
for i in range(1, num + 1):
        factorial *= i
print("Factorial =", factorial)
```

# break, continue, pass statements

**break:** Python break statement is used to terminate the current loop and resumes execution at the next statement. The break statement can be used in both Python while and for loops. Its syntax is

```
break
```

**continue :** The `continue` keyword is used to end the current iteration in a `for` loop (or a `while` loop), and continues to the next iteration. Its syntax is

```
continue
```

**pass :**

- The pass statement in Python is a **null operation**.
- When Python executes it, **nothing happens**.
- It is mainly used as a **placeholder** in code, where you haven't written logic yet, but the syntax requires a statement.

Its syntax is

```
pass
```

# while else

This else statement is executed only when the while loops are executed completely and the condition becomes false. If we break the while loop using the "break" statement then the else statement will not be executed.

```
While (condition):
    # Code to execute while the condition is true
else:
   # Code to execute after the loop ends naturally
```

```
n = 1
sum = 0
# While loop to calculate the sum
while n <= 5:
    sum += n
    n += 1
else:
   print("Loop completed. Sum =", sum)
print("end of the program")
```

O/p:

Loop completed. Sum = 15

end of the program

# for else

## For else :

**How for...else works**

- Loop completes normally: The else block executes after the for loop has gone through all the items in the iterable.
- Loop terminated by break: The else block is completely skipped if a break statement is executed inside the loop.

**Student Activity:** use break statement in while, for and analyse the output.

```
for i in range(5):
    print(i)
else:
    print("Finally finished!")
```

O/p:

```
0
1
2
3
4
Finally finished!
```