Data structures and Algorithms

Adipta Basu

Introduction

What is a Data Structure?

Data Structures are different ways of organising data on your computer, so that they can be used efficiently.

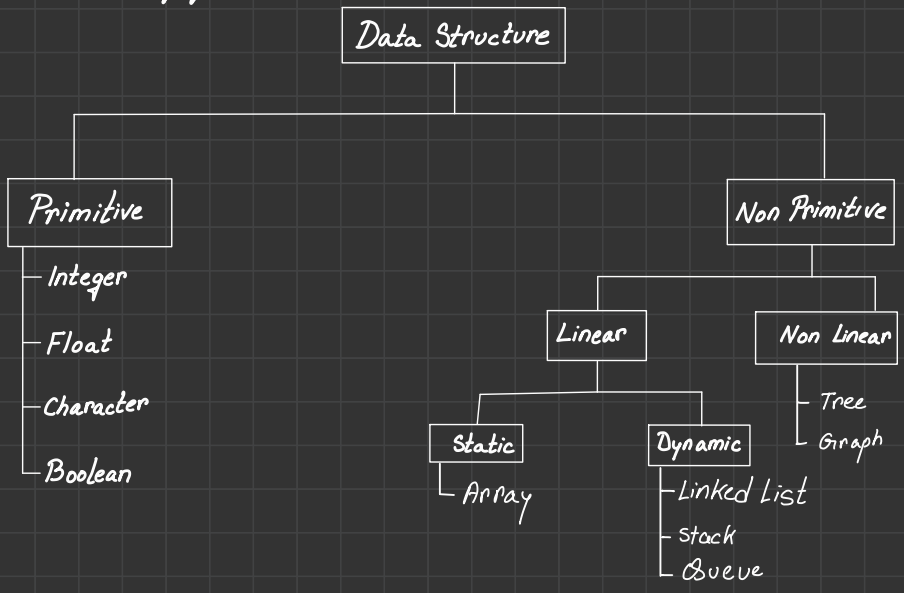
What is an Algorithm?

A set of instructions to perform a task.

What makes a good algorithm?

- Correctness
- Efficiency.

Types of Data Structures



Types of Algorithms:

1. Simple Recursive Algorithms → Such algorithms works in a way like iterative algorithms.

2. Divide and Conquer Algorithms
   - Divides the problem into smaller subproblems of the same types, and solve these problems recursively.
   - Combine the solutions into a solution to the original problem.
   Examples: Quick Sort and Merge Sort.

3. Dynamic Programming Algorithms
   - They work based on memoization.
   - To find the best solution.

4. Greedy Algorithms
   - We take the best we can without thinking about future consequences.
   - We hope that by choosing a local optimal solution at each step, we will end up with a global optimum solution.

5. Brute Force Algorithms
   - It simply tries all possibilities until a satisfactory solution is found.

6. Randomized Algorithms
   - Use a random number atleast once during the computation to make a decision.

Recursion

What is Recursion?

Recursion is a way of solving a problem by having a function call itself.

- Perform the same operation multiple times with different inputs.
- In every step we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion, otherwise infinite loop will occur.

Why we need Recursion?

1. Recursive thinking is really important in programming and it helps you break-down big problems into smaller ones and easier to use.
   - When to use recursion?
     - If you can divide the problem into similar sub problems
2. The prominent usage of recursion in data structures like trees and graphs.

The Logic Behind Recursion

1. A method calls itself.
2. Exit from infinite loop

Recursive vs Iterative Solutions

Points	Recursion	Iteration	
Space Efficient	No	Yes	No stack memory exep needed.
Time Efficient	No	Yes	For recursion, system needs more time to push or pop to stack.
Easy to code	Yes	No	We use recursion, where we know, it has sub-problems.

When to Use/Avoid Recursion?

When to use it?

- When we can easily breakdown a problem into similar subproblem.
- When we are fine with extra overhead (both time and space) that it comes with it.
- When we need a quick working solution, instead of an efficient one.
- When traversing a tree.
- When we use memoization in recursion.

When to avoid it?

- If space and time complexity matters to us.
- Recursion is more memory. If we use embedded memory. For example, if an application takes more memory on the phone, it is not efficient.
- Recursion can be slow.

Writing Recursion in 3 Steps

Step 1: Figure out the main recursive flow.

Step 2: Define the base condition, which will be the stopping criteria.

Step 3: Define checks for corner cases.

Examples:

1. Write the Fibonacci Recursion in 3 steps

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Step 1. Developing the recursive flow.

$5 = 3 + 2 \Rightarrow f(n) = f(n-1) + f(n-2)$

Step 2. Establishing the base case for the stopping criteria.

For $n=0$ or $n=1$, it will return the number itself.

Step 3. Corner case, or constraint handling

So, if $n < 0$, return (-1).

Java Implementation

```
pubic clan SimpleRecursion
{
   public static void main (String[] args)
   {
      System.out.println (getFiboNum (15));
   }
   public static long getFiboNum (int num)
   {
      long output = 0;
      if (num < 0)
         output = -1;
      else if (num == 0 || num == 1)
         output = num;
      else
         output = getFiboNum (n-1) + getFiboNum (n-2);
   }
}
```

Javascript Implementation

```
function getFiboNum(num)
{
   let output=0;
   if(num<0)
      output=-1;
   else if(num==0 || num==1)
      output=num;
   else
      output=getFiboNum(num-1)+getFiboNum(n-2);
   return output;
}
function Main(num)
{
   console.log(getFiboNum(num));
}
Main(4);
```

2. WAP to find the sum of all digits in a number.

Java

```
public clan SumOfDigits
{
   public static void main(String[] args)
   {
      System.out.println(sumOfDigits(1234);
   }
   public static int sumOfDigits(int n)
   {
      int sum=0;
      if(n!=0)
         sum=(n%10)+sumOfDigits(n/10);
      return sum,
   }
}
```

JS

```
function Main(num)
{
    console.log(sumOfDigits(num));
}
function sumOfDigits(num)
{
    let sum=0;
    if(num<=0)
        sum=0;
    else
        sum=(num%10)+sumOfDigits(Math.floor(num/10));
}
Main(1234);
```

3. WAP to find the GCD of two numbers.

Euclidean GCD Algorithm

gcd(48,16)

Step 1. 48/18 = 2, Remainder 12

Step 2. 18/12 = 1, Remainder 6

Step 3. 12/6 = 2, Remainder 0

∴ gcd(0,b) = b

& gcd(a,b) = gcd(b%a, a)

a	b	b%a
18	48	12
12	18	6
6	12	0

Java

```
public class EucledianGCD
{
   public static void main(String[] args)
   {
      System.out.println(getGCD(18,48));
   }
   public static int getGCD(int a, int b)
   {
      if(a==0)
        return b;
      else
        return getGCD(b%a,a);
   }
}
```

JS

```
function getGCD(a,b)
{
   if(a==0)
     return b;
   else
     return getGCD(b%a,a);
}
function Main()
{
   console.log(getGCD(18,48));
}
Main();
```

4. WAP to convert a number to binary from decimal.

Logic: 13 to binary

Division by 2	Quotient	Remainder
13/2	6	1
6/2	3	0
3/2	1	1
1/2	0	1

$13_{10} = 1101_2$

Java

```
public class NumToBinary
{
   public static void main(String[] args)
   {
      System.out.println(getBin(13));
   }
   public static long getBin(int n)
   {
      if(n>0)
         return (getBin(n/10)*10)+(n%2);
      else
         return 0;
   }
}
```

Complexity

What is Big O?

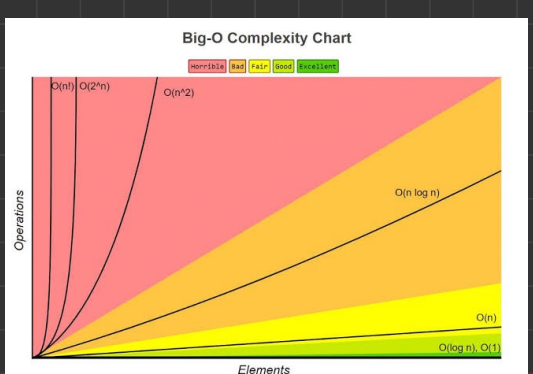
Big O is the language and metric we use to describe the efficiency of algorithms.

Big O Notations

- **Big O:** It is the complexity that is going to be less or equal to the worst case.
- **Big Ω (Omega):** It is a complexity that is going to be atleast more than the best case.
- **Big Θ (Theta):** It is a complexity that is within bounds of the worst and best case.

Runtime Complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Looping through array elements
$O(\log N)$	Logarithmic	Binary search in sorted array
$O(N^2)$	Quadratic	Looking at every index of the array twice. Ex→ Bubble Sort
$O(2^N)$	Exponential	Double recursion in Fibonacci



Dropping Constants and Non-Dominant Terms

Drop Constants
$O(2N) \rightarrow O(N)$

Drop Non-Dominant Terms
$O(N^2 + N) \rightarrow O(N^2)$

How to measure the code using Big O?

Rule No.	Description	Complexity
Rule 1	Any assignment statements that are executed once, regardless of the size of the problem.	$O(1)$
Rule 2	A simple for loop from 0 to n (with no internal loops)	$O(n)$
Rule 3	A nested loop with the same type takes quadratic time complexity.	$O(n^2)$
Rule 4	A loop, in which the controlling parameter is divided by 2 at each step.	$O(\log N)$
Rule 5	When dealing with multiple statements, just add them.	

Sample Evaluation

```
public static void findBiggestNumber ( [] sampleArray)
{
  let biggestNumber = sampleArray[0];                          O(1)
  for(let index =1; sampleArray.length ; index ++)             O(n)
  {                                                                  } O(n)
    if( sampleArray[index] > biggestNumber )                   O(1)  }
    {                                                                } O(1)
      biggestNumber = sampleArray[index];                      O(1)  }
    }
  }
  System.out.println( biggestNumber );                         O(1)
}
```

Time complexity $\Rightarrow O(1) + O(n) + O(1) = O(n)$