DATA STRUCTURES AND ALGORITHMS

- LECTURE 2-

(2ND YEAR OF STUDY)

Skopje, 2023/24

Contents

2. Algorithms

2.1. Introduction

- The word **Algorithm** means:
  - *A set of finite rules or instructions to be followed in calculations or other problem-solving operations; or*
  - *A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations.*
- Therefore, Algorithm refers to a *sequence of finite steps* to solve a particular problem for a finite-sized input.

2.2. Classification

- The algorithms can be *classified* in various ways.
- They are:

1. Implementation Method
2. Design Method
3. Design Approaches
4. Other Classifications

- The classification of algorithms is important for several reasons: Organization, Problem solving, Performance comparison, Reusability, Research.

2.2. Classification

1. **Classification by Implementation Method:** There are primarily three main categories:

- **Recursion or Iteration:** A ***recursive algorithm*** is an algorithm which calls itself again and again until a base condition is achieved, whereas ***iterative algorithms*** use ***loops*** and/or ***data structures*** like stacks, queues to solve any problem. Every recursive solution can be implemented as an iterative solution and vice versa.

2.2. Classification

- **Exact or Approximate:** Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem.

2.2. Classification

- **Serial or Parallel or Distributed Algorithms:** In *serial* algorithms, one instruction is executed at a time, while *parallel* algorithms are those in which we divide the problem into sub-problems and execute them on different processors. If parallel algorithms are distributed on different machines, then they are known as distributed algorithms.

2.2. Classification

2. **Classification by Design Method:** There are primarily seven main categories:

- **Greedy Method:** In the ***greedy method***, at each step, a decision is made to choose the *local optimum*, without thinking about the future consequences.
- **Divide and Conquer:** The ***Divide and Conquer*** strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.

2.2. Classification

- **Dynamic Programming:** The approach of ***Dynamic programming*** is similar to ***divide and conquer***. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. "Dynamic" means we dynamically decide, whether to call a function or retrieve values from the table.

2.2. Classification

- **Linear Programming:** In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs.
- **Reduction (Transform and Conquer):** In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.

2.2. Classification

- **Backtracking:** This technique is very useful in solving combinatorial problems that have *a single unique solution*. Where we have to find the correct combination of steps that lead to fulfillment of the task. Such problems have multiple stages and there are multiple options at each stage. This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution.

2.2. Classification

- **Branch and Bound:** This technique is very useful in solving combinatorial optimization problem that have *multiple solutions* and we are interested in finding the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree. As the program progresses each state combination is explored, and the previous solution is replaced by new one if it is not the optimal than the current solution.

2.2. Classification

3. **Classification by Design Approaches:** There are two approaches:

- **Top-Down approach:** Breaking down a complex problem into smaller, more manageable sub-problems and solving each sub-problem individually. Designing a system starting from the highest level of abstraction and moving towards the lower levels.
- **Bottom-up approach:** Building a system by starting with the individual components and gradually integrating them to form a larger system. Solving sub-problems first and then using the solutions to build up to a larger problem solution.

2.2. Classification

4. **Other Classifications:** Apart from classifying the algorithms into the above broad categories, the algorithms can be classified into other broad categories like:

- **Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as ***randomized algorithms***. **Example:** Randomized Quicksort Algorithm.

2.2. Classification

- **Classification by complexity:** Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as ***time complexity analysis***. **Example:** Some algorithms take linear time, while some take exponential time.

2.2. Classification

- **Classification by Research Area:** In Computer Science each field has its own problems and needs efficient algorithms. **Example:** Sorting Algorithms, Searching Algorithms, Machine Learning Algorithms, etc.
- **Branch and Bound Enumeration and Backtracking:** These are mostly used in Artificial Intelligence.

2.3. Types of algorithms

- An algorithm is a step-by-step procedure to solve a problem.
- A good algorithm should be optimized in terms of **time** and **space**. Different types of problems require different types of algorithmic techniques to be solved in the most optimized manner.
- There are many types of algorithms, but the most important and fundamental algorithms are here discussed.

2.3. Types of algorithms

1. **Brute Force Algorithm:** This is the most basic and simplest type of algorithm. It is a straightforward approach to a problem, i.e., the first approach that comes to our mind on seeing the problem. More technically it is just like iterating every possibility available to solve that problem.

- **Example:** If there is a lock of 4-digit PIN. The digits to be chosen from 0-9, then the brute force will be trying all possible combinations one by one like: 0001, 0002, 0003, 0004, and so on until we get the right PIN. In the worst case, it will take 10,000 tries to find the right combination.

2.3. Types of algorithms

2. **Divide and Conquer Algorithm:** As the name suggests, it breaks the problem into parts, then solves each part and after that again merges the solved subtasks to get the actual problem solved.

- A typical *Divide and Conquer algorithm* solves a problem using following three steps:
  - **Divide:** Break the given problem into subproblems of same type.
  - **Conquer:** Recursively solve these subproblems.
  - **Combine:** Appropriately combine the answers.
- This is the primary technique mentioned in the two sorting algorithms: ***Merge Sort*** and ***Quick Sort***.

2.3. Types of algorithms

3. **Greedy Algorithm:** It works step-by-step, and always choose the steps which provide immediate profit/benefit. The choice made by the greedy approach does not consider future data and choices. In some cases making a decision that looks right at that moment gives the best solution (Greedy), but in other cases, it doesn't. The Greedy technique is used for *optimization* problems (where we have to find the *maximum* or *minimum* of something).

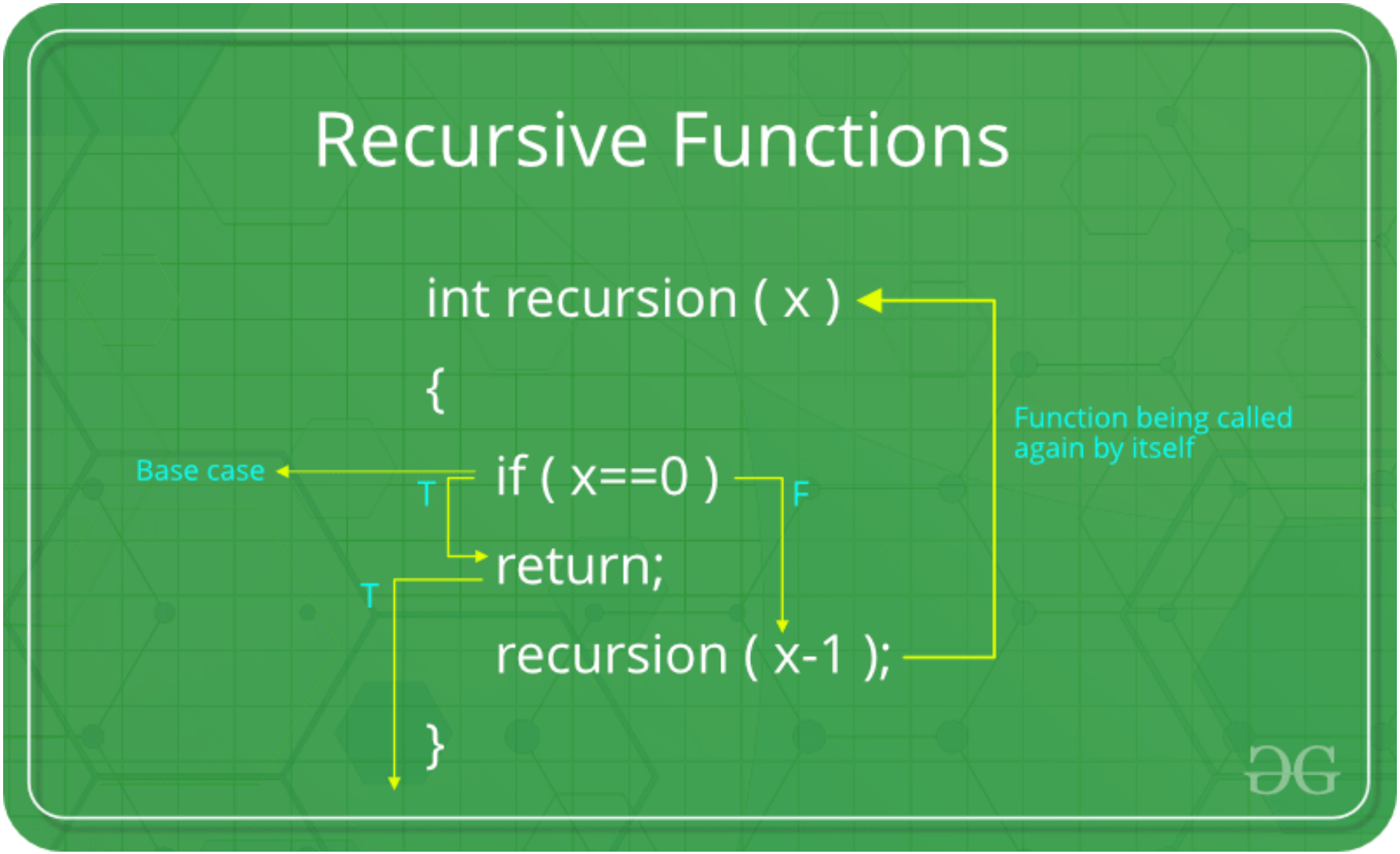
- Standard Greedy algorithms: ***Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm.***

2.3. Types of algorithms

4. **Recursion:** It is one of the most important algorithms which uses the concept of *code reusability* and *repeated usage* of the same piece of code.

- The point which makes Recursion one of the most used algorithms is that it forms the base for many other algorithms such as:
  - ***Tree traversals***
  - ***Graph traversals***

2.3. Types of algorithms



2.3. Types of algorithms

5. **Backtracking Algorithm:** This algorithm is derived from the Recursion algorithm, with the option to revert if a recursive solution fails, i.e., in case a solution fails, the program traces back to the moment where it failed and builds on another solution. So basically it tries out all the possible solutions and finds the correct one.

- Some important and most common problems of backtracking algorithms are: ***Knight's tour problem, Rat in a maze, Sudoku***.

2.3. Types of algorithms

6. **Dynamic Programming:** It is mainly an optimization over plain Recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.

- The main concept of the *Dynamic Programming algorithm* is to use the previously calculated result to avoid repeated calculations of the same subtask, which helps in *reducing* the time complexity.

2.3. Types of algorithms

- **Example:** The **Fibonacci sequence** is a sequence in which each number is the sum of the two preceding ones. Numbers that are part of the Fibonacci sequence are known as **Fibonacci numbers,** commonly denoted F_n.
- The Fibonacci numbers may be defined by the recurrence relation:

$$F_0 = 0 \text{ , } F_1 = 1 \text{ , and}$$

$$F_n = F_{n-1} + F_{n-2} \text{ , for } n > 1.$$

2.3. Types of algorithms


```
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Recursion : Exponential

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

Dynamic Programming : Linear

Exponential versus ***Linear*** Time Complexity