Lab Manual

CL2001 – Data Structures

Fall-2023

National University of Computer and Emerging Sciences-FAST
Karachi Campus

Data Structures Lab#5

Course: Data Structures (CL2001) **Semester:** Fall 2023

Instructor: Shafique Rehman **T.A:** N/A

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session**.**

Contents:

- Recursion in Detail
- Backtracking

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

```
public class Main {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

Example Explained

When the sum() function is called, it adds parameter k to the sum of all numbers smaller than k and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:

10 + sum(9)

10 + (9 + sum(8))

10 + (9 + (8 + sum(7)))

...

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
Since the function does not call itself when k is 0, the program stops there and returns the result.

Halting Condition

Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter k becomes 0.

It is helpful to see a variety of different examples to better understand the concept.

Java Recursion Example 2: Factorial Number

```
public class RecursionExample3 {
   static int factorial(int n){
        if (n == 1)
          return 1;
        else
          return(n * factorial(n-1));
   }

public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5));
}
}
```

Output:

```
Factorial of 5 is: 120
```

Working of above program:

factorial(5)

factorial(4)

factorial(3)

factorial(2)

factorial(1)

return 1

return 2*1 = 2

return 3*2 = 6

return 4*6 = 24

return 5*24 = 120

In this example, the function is calculating the factorial of five. The halting condition for this recursive function is when n is equal 1:

Example 3: Factorial of 4

```
class Factorial {

    static int factorial( int n ) {
        if (n != 0)  // termination condition
            return n * factorial(n-1); // recursive call
        else
            return 1;
    }

    public static void main(String[] args) {
        int number = 4, result;
        result = factorial(number);
        System.out.println(number + " factorial = " + result);
    }
}
```

```
4 factorial = 24
```

In the above example, we have a method named factorial(). The factorial() is called from the main() method. with the number variable passed as an argument.

Here, notice the statement,

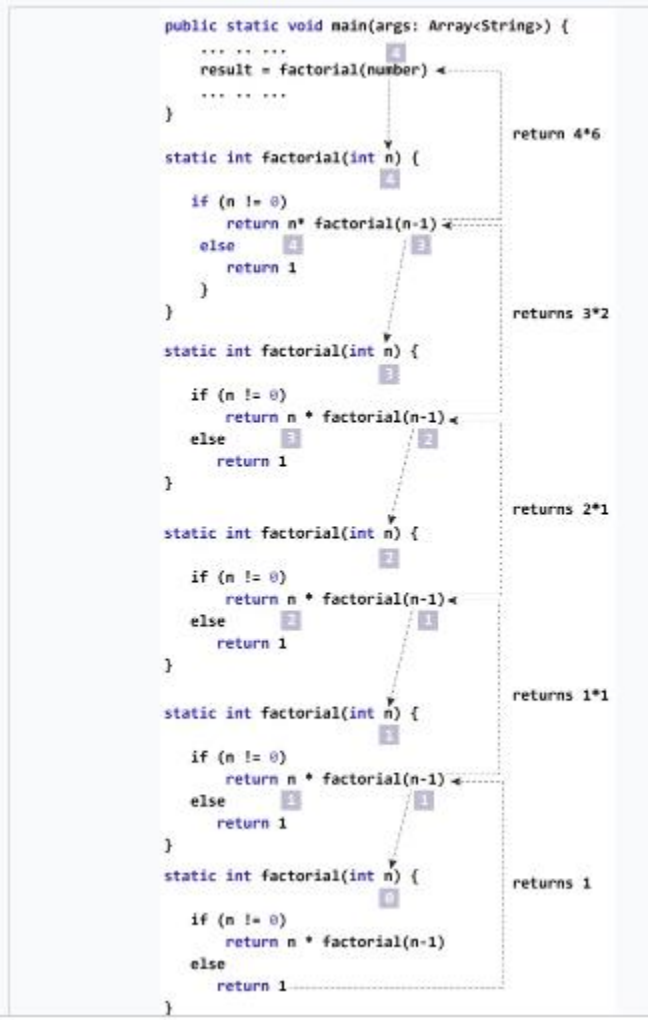
return n * factorial(n-1);

The factorial() method is calling itself. Initially, the value of n is 4 inside factorial(). During the next recursive call, 3 is passed to the factorial() method. This process continues until n is equal to 0.

When n is equal to 0, the if statement returns false hence 1 is returned. Finally, the accumulated result is passed to the main() method.

Working of Factorial Program

The image below will give you a better idea of how the factorial program is executed using recursion.



The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

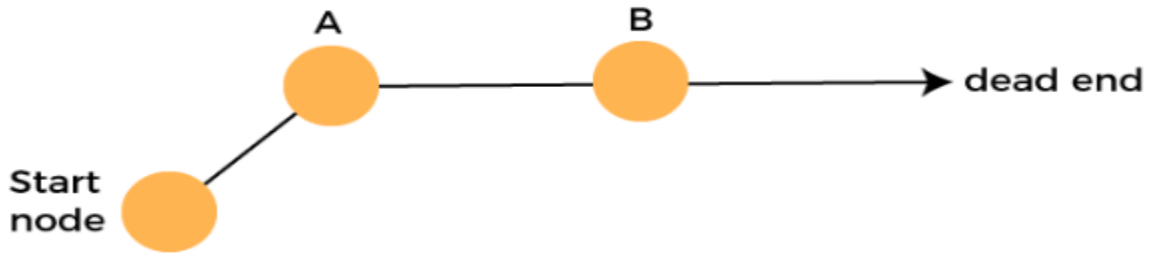
Backtracking

Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

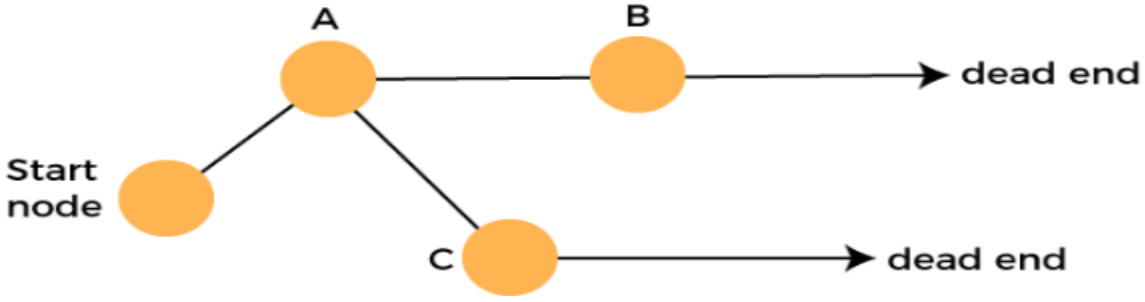
How does Backtracking work?

Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.

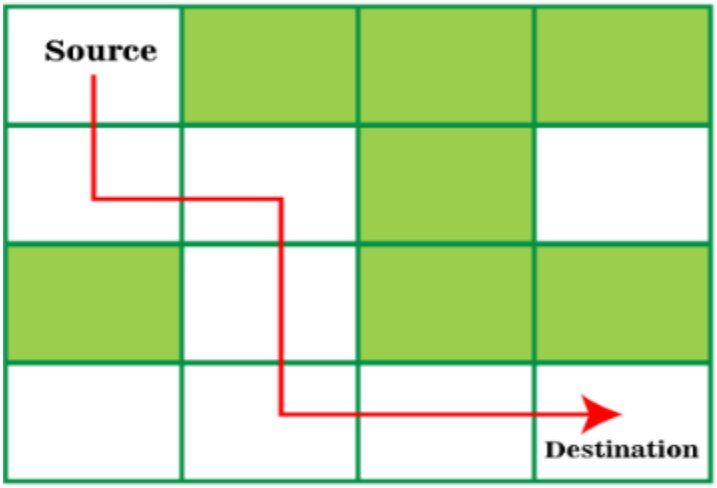


Backtracking works in this way until we find the final solution.

Example of Backtracking

Rate in Maze

A maze is provided in the form R * C matrix, where R is the total number of rows, and C is the total number of columns present in the matrix (R may or may not become equal to C). The cell m[0][0] is the source (observe the following diagram). From the source, the rat starts its journey. The rat has to reach cell m[R - 1][C - 1], the destination cell. The rat can only move in the rightward (→) or in the downward (↓) direction from the cell where it is currently present. Also, note that the rat can only move to its adjacent cell. For example from the cell m[0][0], it can move to m[0][1] or m[1][0]. From m[0][0] it cannot move to m[0][2] or m[2][0] directly as cells m[0][2] or m[2][0] is not adjacent to m[0][0].



Algorithm

On the basis of the above approach, the following algorithm is written.

Create a result matrix (result[R][C]), and fill each cell of the matrix with 1's.

Create a recursive method that takes the initial matrix, the result matrix, and the current position of the rat (which is (0, 0) initially).

If the position goes out of the matrix or the rat lands in the position is not valid, then return. It is because that path cannot be the correct path. Therefore, no need to proceed further.

Put the value of the cell result[r][c] as 0 and then check whether the current cell is the destination cell or not. If the destination cell is reached, display the result matrix and terminate.

If the destination cell is not reached, then recursively check for the position (r + 1, c) and (r, c + 1).

If the rat cannot reach the destination cell form the position (r, c), then unmark the position or cell (r, c), that is, result[r][c] = 1.