chapter 5- Recursion

College of computing Science,
Department of Software Engineering, DBU
Instructor: Bekele M.(MSc in Software Eng.)
Address: Email: bekele18@gmail.com
December 2024 E.C

Outlines

- Recursive definitions
- Function calls and recursive implementation
- Tail recursion
- Nondaily recursion
- indirect recursion
- Nested recursion
- Excessive recursion
- Backtracking.

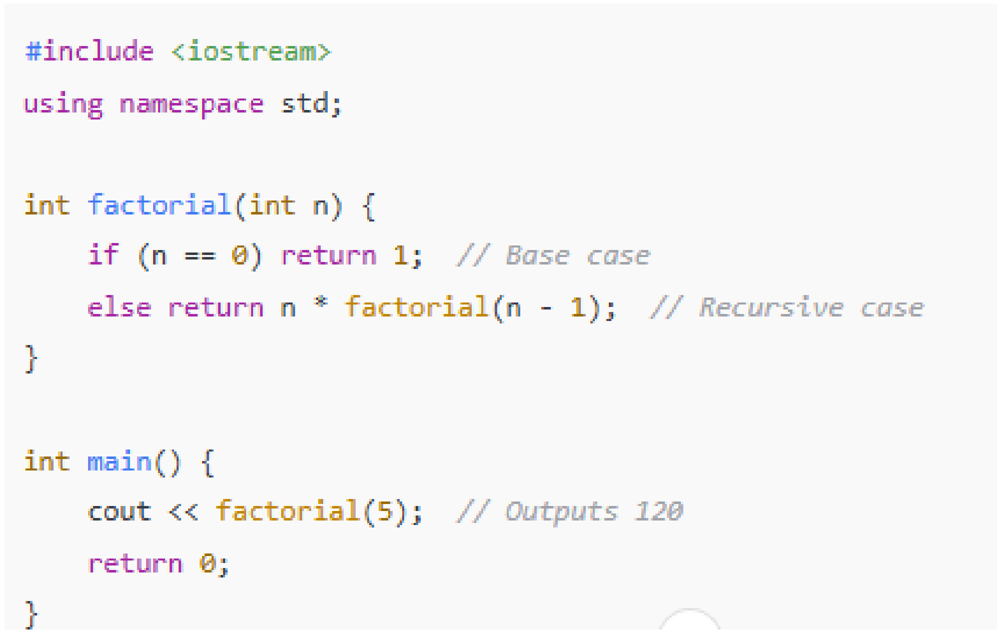
Recursive

- Recursive is a programming technique where a function calls itself to solve a problem. It often divides the problem into smaller subproblems.
- There are base and recursive cases.
  - **Anchor or Base Case:** The simplest form of the problem, which can be solved directly.
  - **Recursive Case:** The larger problem is reduced into smaller subproblems, often with the function calling itself.

Example: Factorial function:

•Factorial is defined recursively:

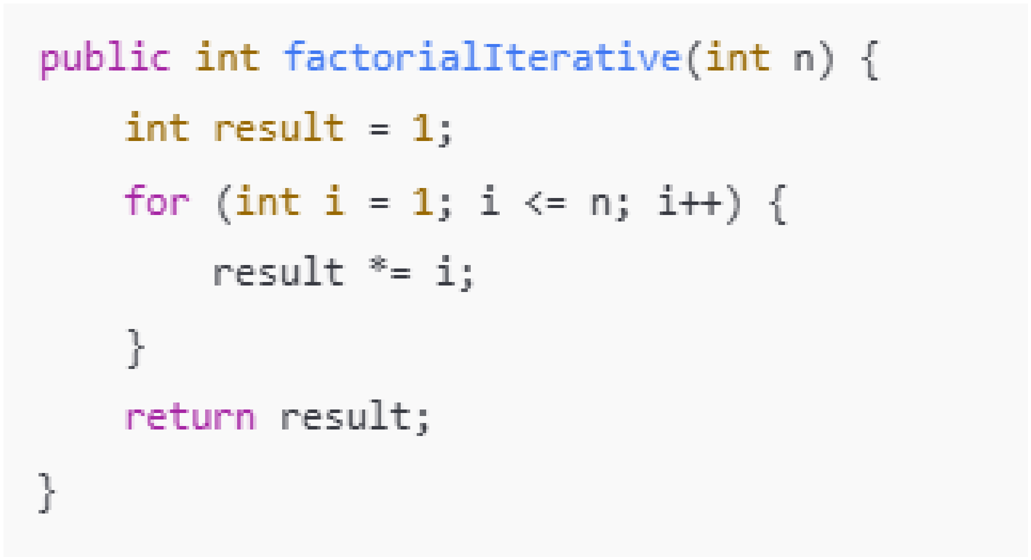
- •n! = 1 if n = 0
- •n! = n * (n - 1)! if n > 0



```
#include <iostream>
using namespace std;

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

int main() {
    cout << factorial(5);  // Outputs 120
    return 0;
}
```

Recursion vs. Iteration

- Recursion
  - It involves the function calling itself.
  - Each recursive call creates a new instance of the function, storing information in the call stack.
  - After reaching the base case, the function returns the result back through the call stack.
- Iteration
  - involves loops and can be more efficient.
- While both can solve the same problem, recursion is often more intuitive for problems that have an inherent recursive structure.
- Example: Factorial using iteration



```java
public int factorialIterative(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

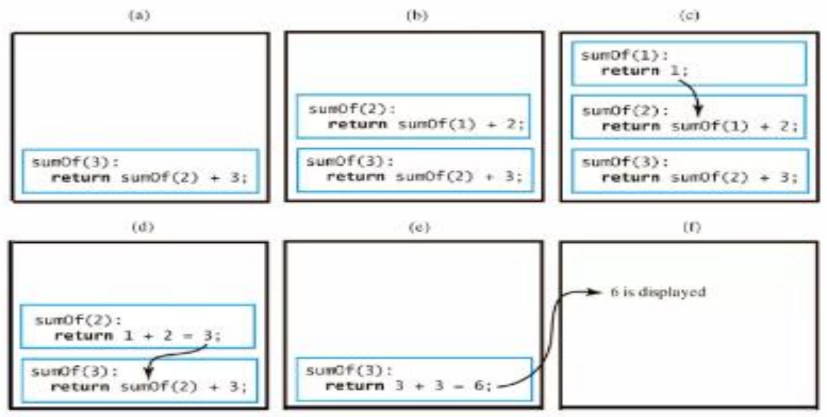
Function Calls and Recursive Implementation

- When a function is called, the runtime stack plays a crucial role in keeping track of the execution context. The stack contains an activation record or stack frame for each function call, which includes:
    - Function parameters: Values passed to the function.
    - Local variables: Variables declared within the function.
    - Return address: Where the program should continue after the function finishes.
    - Dynamic link: A reference to the calling function’s activation record.
    - Return value (if applicable).
- In the case of recursive functions, every recursive call pushes a new activation record onto the stack.

Function Calls and Recursive Implementation

- Examples: compute the sum 1+2+3+...+n for an integer n>o

```cpp
#include <iostream>
using namespace std;
int sumof(int n) {
     if (n == 1) return 1;
     return n + sumof(n - 1);
}
int main() {
     int n;
     cin >> n;
     cout << sumof(n) << endl;
     return 0;
}
```



The stack of activation records
during the execution of a call to **sumOf(3)**

- Initial Call: sumof(3) → Activation record for sumof(3) is pushed.

```
Stack: sumof(3)
```

- Recursive Call: sumof(3) calls sumof(2) → Activation record for sumof(2) is pushed.

```
Stack: sumof(3), sumof(2)
```

- Recursive Call: sumof(2) calls sumof(1) → Activation record for sumof(1) is pushed.

```
Stack: sumof(3), sumof(2), sumof(1)
```

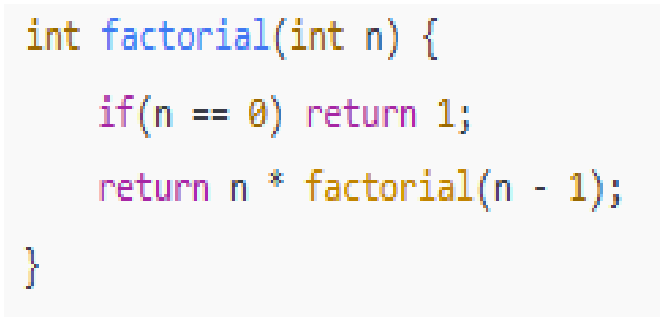
- Base Case: sumof(1) returns 1 → Activation record for sumof(1) is popped.

```
Stack: sumof(3), sumof(2)
```

- Returning from sumof(2): sumof(2) returns 2 + 1 = 3 → Activation record for sumof(2) is popped.
- Returning from sumof(3): sumof(3) returns 3 + 3 = 6 → Activation record for sumof(3) is popped.
- Final Result: sumof(3) returns 6.

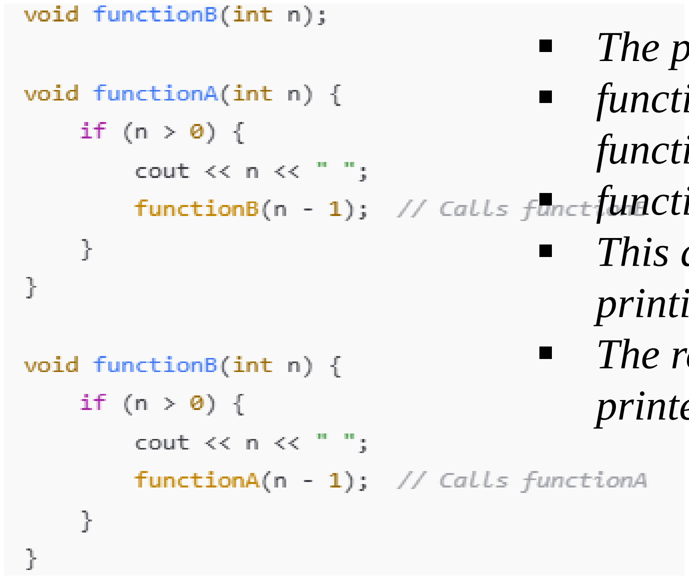
Types of Recursion

- **Direct Recursion**: Direct recursion occurs when a function calls itself within its own definition. The function repeats its execution in a recursive manner until a base condition is met, which stops the recursion. Example:



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

- **Indirect Recursion:** A function calls another function, which then calls the first function. Example:

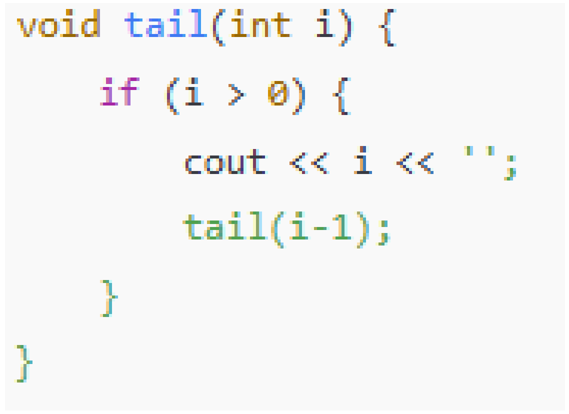


```
void functionB(int n);

void functionA(int n) {
    if (n > 0) {
        cout << n << " ";
        functionB(n - 1);   // Calls functionB
    }
}

void functionB(int n) {
    if (n > 0) {
        cout << n << " ";
        functionA(n - 1);   // Calls functionA
    }
}
```

- *The program asks the user to input a number num.*
- *functionA(num) is called, printing num and calling functionB(num - 1).*
- *functionB(num - 1) prints its value and calls functionA(num - 2).*
- *This alternation between functionA and functionB continues, printing decreasing numbers.*
- *The recursion stops when n <= 0 and no more values are printed.*

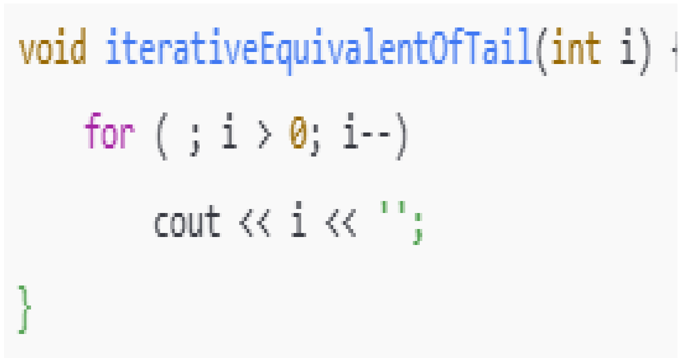
Tail Recursion and Non-Tail Recursion

- Tail recursion occurs when a recursive call is the last operation in the function. No further computation is done after the recursive call.
- Tail recursion can be optimized by compilers or converted to an iterative solution, making it more efficient.
- Example of tail recursion



```
void tail(int i) {
    if (i > 0) {
        cout << i << ' ';
        tail(i-1);
    }
}
```

Iterative equivalent:

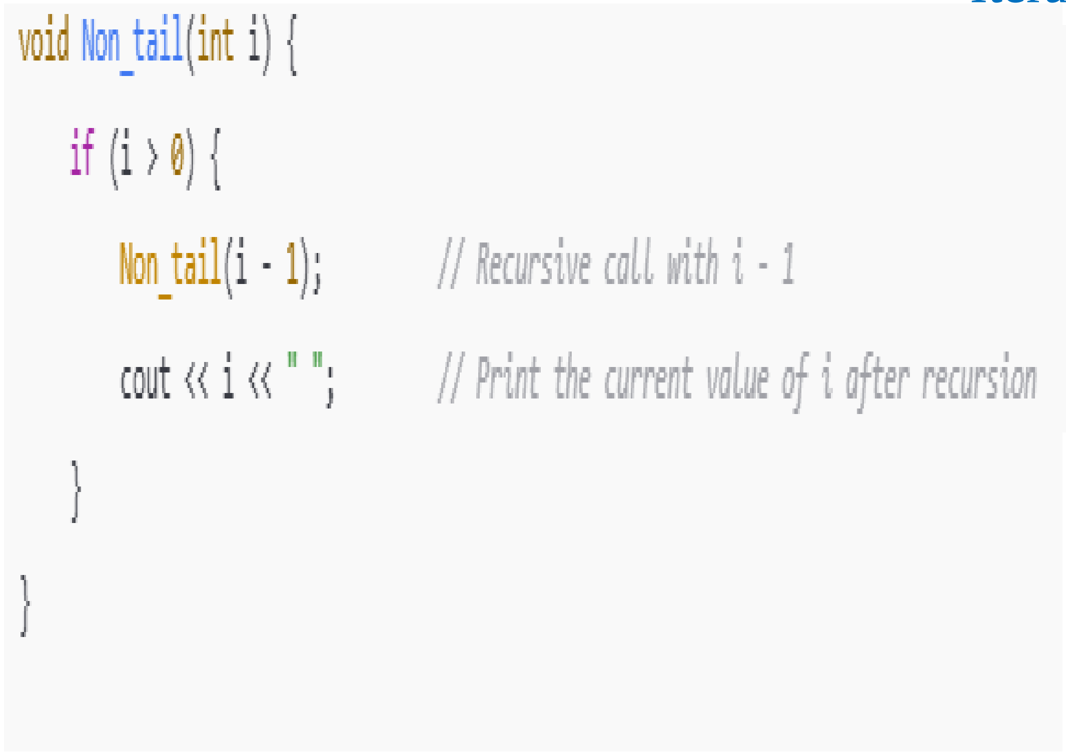


```
void iterativeEquivalentOfTail(int i) {
    for ( ; i > 0; i--)
        cout << i << ' ';
}
```

- *Initial Call (tail(3)): The function prints 3 and makes a recursive call with i = 2.*
- *Second Call (tail(2)): The function prints 2 and makes a recursive call with i = 1.*
- *Third Call (tail(1)): The function prints 1 and makes a recursive call with*

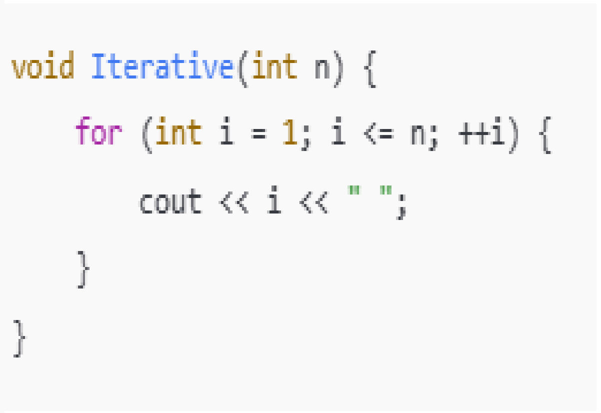
Tail Recursion and Non-Tail Recursion

- **Non-Tail Recursion** A type of recursion where the recursive call is followed by further computation or operations after the recursive call. returns.
- Characteristics:
  - The recursive call is not the last operation in the function.
  - Memory usage is higher due to stack frames not being optimized.
- Example of non-tail recursion



```
void Non_tail(int i) {
    if (i > 0) {
        Non_tail(i - 1);        // Recursive call with i - 1
        cout << i << " ";       // Print the current value of i after recursion
    }
}
```

Iterative equivalent:



```
void Iterative(int n) {
    for (int i = 1; i <= n; ++i) {
        cout << i << " ";
    }
}
```

Cont’d

- **Understanding Stack Frames & Popping in Recursion previous example**
  - **First Call: Non_tail(3)**
    - Stack frame for Non_tail(3) created.
    - Calls Non_tail(2).
  - **Second Call: Non_tail(2)**
    - Stack frame for Non_tail(2) created.
    - Calls Non_tail(1).
  - **Third Call: Non_tail(1)**
    - Stack frame for Non_tail(1) created.
    - Calls Non_tail(0).
  - **Fourth Call: Non_tail(0)**
    - Stack frame for Non_tail(0) created.
    - Base case reached (i == 0), returns.
    - Stack frame for Non_tail(0) is popped.
  - **Returning to Non_tail(1)**
    - Prints i (1).
    - Stack frame for Non_tail(1) is popped.
  - **Returning to Non_tail(2)**
    - Prints i (2).
    - Stack frame for Non_tail(2) is popped.
  - **Returning to Non_tail(3)**
    - Prints i (3).
    - Stack frame for Non_tail(3) is popped.
  - **Recap:**
    - Each recursive call adds a new stack frame.
    - When a function returns, its stack frame is popped, and control returns to the previous function.

Nested Recursion

- Nested recursion involves recursive calls within the parameters of the recursive function.
- This creates multiple layers of recursive calls.
- Example 1: Nested Recursion

The given recursive relation for $h(n)$ is defined as:

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(n)) & \text{if } n \leq 4 \end{cases}$$

- From this definition, it is given that the function has solutions for n = 0 and n > 4
- However, for n = 1, 2, 3, and 4, the function determines a value based on a recursive call that requires evaluating itself is called nested recursion

Recurrence Relation for Ackermann Function:

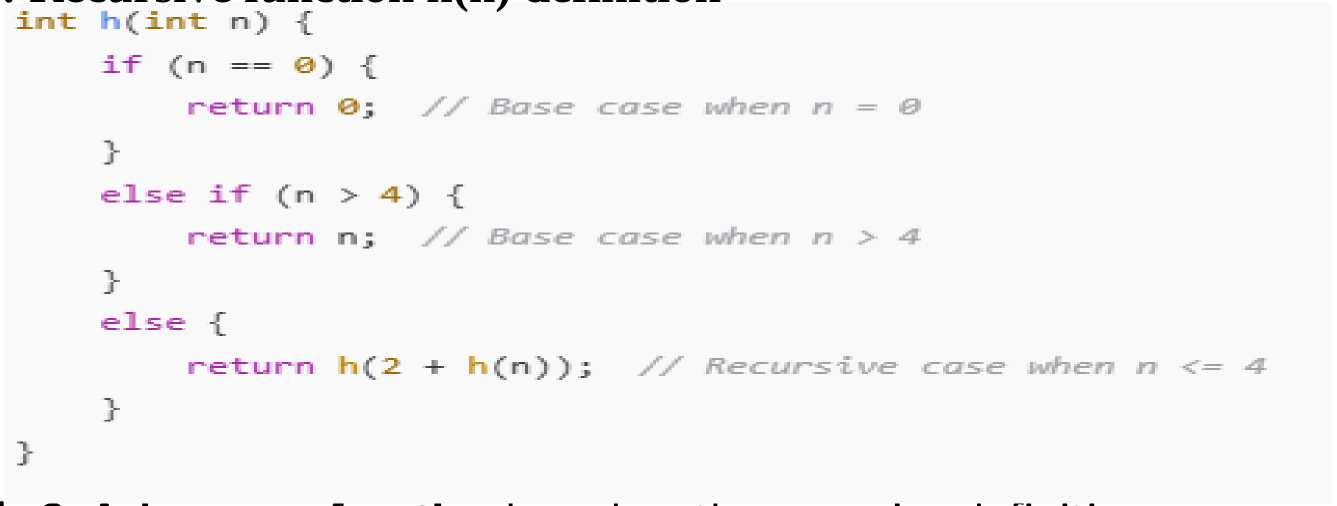
Example 2

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

The **Ackermann function** is a classic example of **nested recursion** that demonstrates extremely fast-growing values, posing challenges for both computation and iterative expression.

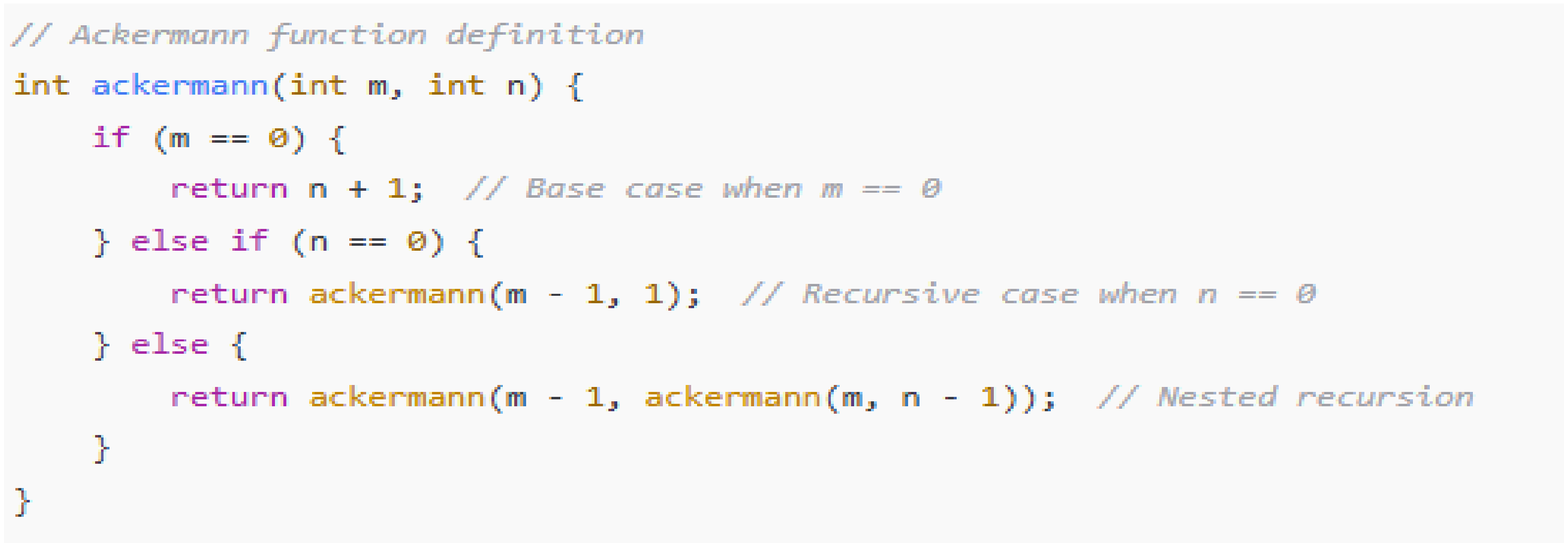
Cont’d

Example 1: Recursive function h(n) definition



```
int h(int n) {
    if (n == 0) {
        return 0;  // Base case when n = 0
    }
    else if (n > 4) {
        return n;  // Base case when n > 4
    }
    else {
        return h(2 + h(n));  // Recursive case when n <= 4
    }
}
```

Example 2: Ackermann function based on the recursive definition:



```
// Ackermann function definition
int ackermann(int m, int n) {
    if (m == 0) {
        return n + 1;  // Base case when m == 0
    } else if (n == 0) {
        return ackermann(m - 1, 1);  // Recursive case when n == 0
    } else {
        return ackermann(m - 1, ackermann(m, n - 1));  // Nested recursion
    }
}
```

Examples

For A(6, 6):

The function A(6, 6) will generate a very large number of recursive calls, as Ackermann's function grows very fast with the inputs. To illustrate, here's the breakdown:

Step-by-Step Breakdown:

1. A(6, 6) calls A(5, A(6, 5)).
2. A(6, 5) calls A(5, A(6, 4)).
3. A(6, 4) calls A(5, A(6, 3)).
4. A(6, 3) calls A(5, A(6, 2)).
5. A(6, 2) calls A(5, A(6, 1)).
6. A(6, 1) calls A(5, A(6, 0)).
7. A(6, 0) calls A(5, 1).
8. A(5, 1) calls A(4, A(5, 0)).
9. A(5, 0) calls A(4, 1).

Excessive Recursion

- Excessive recursion refers to situations where a recursive function uses a deep level of recursion that leads to problems like stack overflow, inefficient performance, or unnecessarily large computational overhead. It typically occurs when:
  - Too many recursive calls are made.
  - There’s no effective base case or termination condition for recursion.
  - However, this straightforwardness does have some drawbacks
  - Generally, as the number of function calls increases, a program suffers from some performance decrease
  - Also, the amount of stack space required increases dramatically with the amount of recursion that occurs
  - This can lead to program crashes if the stack runs out of memory
  - More frequently, though, is the increased execution time leading to poor program performance
  - Recursive calls involve overlapping subproblems that could be computed more efficiently using dynamic programming or memorization.

Excessive Recursion

- Examples

```
unsigned long Fib(unsigned long n) {
                if (n < 2)
                return n;
            else
           return Fib(n-2) + Fib(n-1);
            }
```

- If we use this to compute Fib(6)(which is 8), the algorithm starts by calculating Fib(4) + Fib(5)
- The algorithm then needs to calculate Fib(4) = Fib(2) + Fib(3), and finally the first term of that is Fib(2) = Fib(0) + Fib(1)= 0 + 1 = 1
- Counting the branches, it takes 25 calls to Fib() to calculate Fib(6).
- Notice that it takes almost 3 million calls to determine the 31st Fibonacci number
- This exponential growth makes the algorithm unsuitable for anything but small values of n
- Fortunately there are acceptable iterative algorithms that can be used far more effectively

Backtracking

- Backtracking is nothing but the modified process of the Brute force approach. where the technique systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v1, ..., in) of values and by traversing through the domains of the vectors until the solutions is found.

- Backtracking systematically try and search possibilities to find the solution. Also it is an important process for solving constraint satisfaction problem like crossword, Sudoku and many other puzzles. It can be more continent technique for parsing other combinatorial optimization problem.

- Basically the process is used when the problem has a number of option and just one solution have to be selected. After having a new option set means recursion, the procedure is repeated over and over until final stage.
- Brute Force approach : for a given problem try all possible solutions and select the best one.
- This was used in dynamic programming But, dynamic programming used to solve optimization problem , but in backtracking we want all solutions.
- Example there are three students B1,B2,G1 and three chairs are available. Ho many possible arrangement is possible? Answer: 3!=6 possible arrangement
- Let's considered state space tree

- But in Backtracking problem we usually have some constraints to considers. Thus considering this constraints, generating the possible solutions is the aim.
- Example let's suppose a constraints " girl can not sit in the middle" in the previous example

N-Queens Problem

Problem Description

In a NxN square board N – number of queens need to be placed considering three Condition ---

Bounded Function:

- *No two Queens can be placed in same row.*
- *No two Queens Can be places in same Column*
- *No two queens Can be placed in same Diagonal*

Examples

4-queens

8-queens(**chess board)**

N-queens

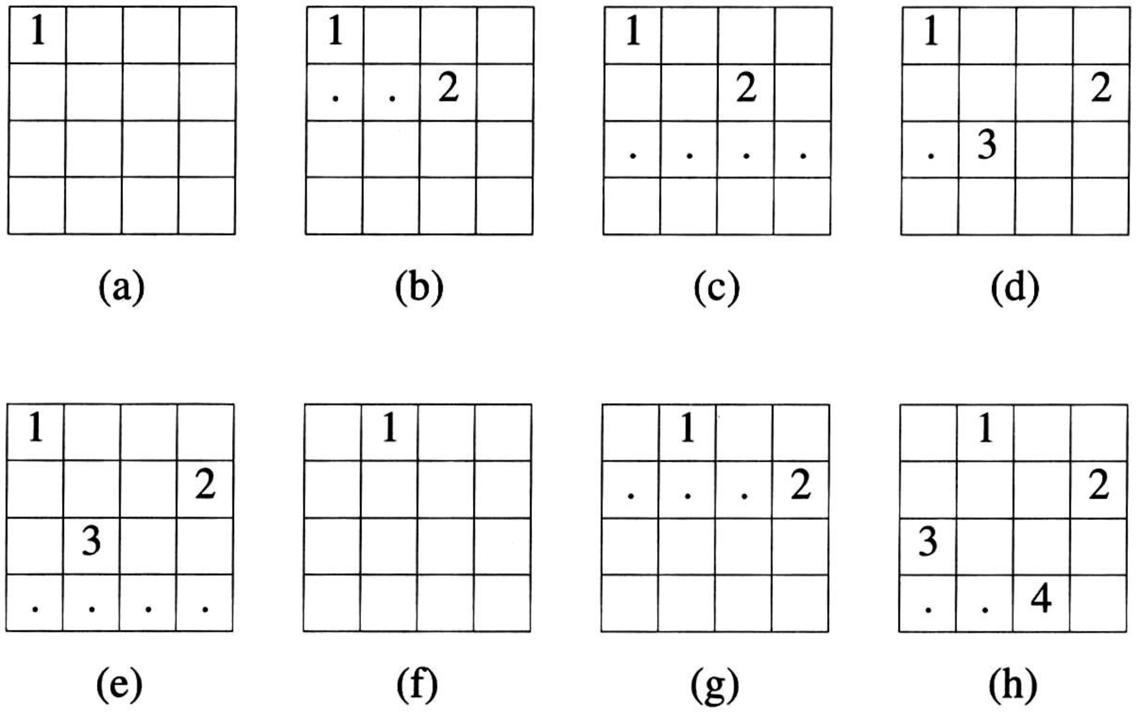
4-Queens Problem

- There are $16C_4$ ways to arrange there are 4 queens in the board. However, there is **bounding function** “no two queens are under attack”
- To minimize the problem let’s arrange Q1-first row, Q2- 2nd row, Q3- 3rd row and Q4- 4th row
- And avoiding placing two queens is the same column.
- Let’s generate state space tree

4-Queens Problem

Example (4-queens)

This example shows how the backtracking works **using Bounded Function**

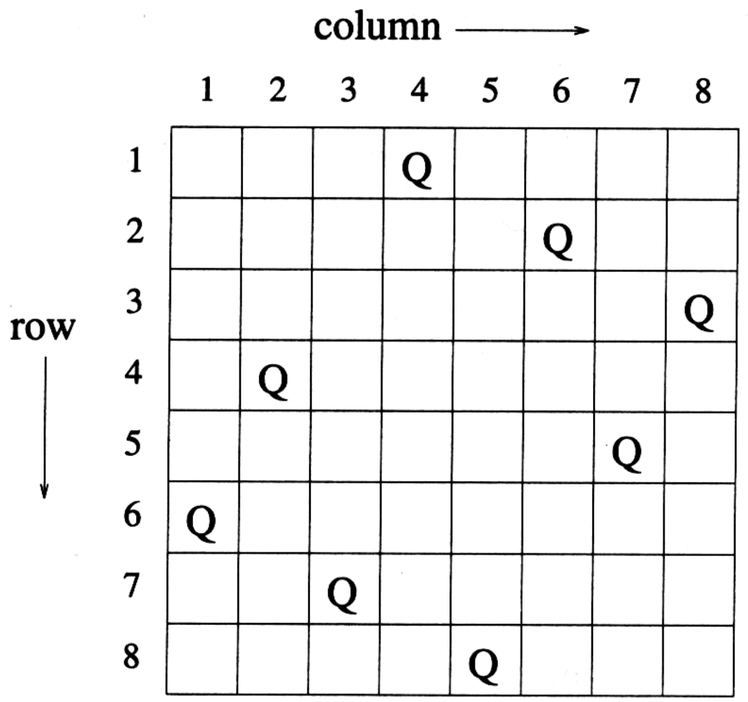


X1=Q1	X2=Q2	X3=Q3	X4=Q4

8-Queens Problems

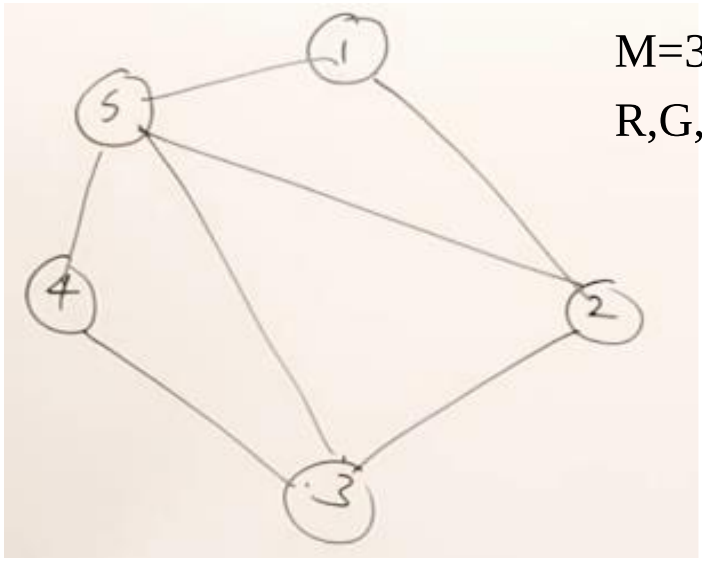
Example (Continued)

8-tuple = (4, 6, 8, 2, 7, 1, 3, 5)



Backtracking Application

- Sum of Subsets
  - **The Sum of Subset Problem is, there will be a set of distinct positive Numbers X and a given Number N. Now, we have to find all the combination of numbers from X that sum up to N. Make a Set of those Number.**
  - **If we sort the weights in nondecreasing order before doing the search, there is an obvious sign telling us that a node is nonpromising.**
  - **Example: Show the pruned state space tree when backtracking is used with**
  - **n = 4, W = 13, and w1 = 3, w2 = 4, w3 = 5, and w4 = 6.**
- Graph Colouring



M=3

R,G,B

Application Recursive Functions

- Solving Complex Tasks:
  - Breaks down complex problems into simpler, smaller instances.
  - Example: Factorial function (factorial(n) = n * factorial(n-1)).
- Divide and Conquer:
  - Solves problems by dividing them into smaller subproblems and combining the results.
  - Example: Merge Sort, Quick Sort.
- Backtracking:
  - Solves problems by exploring all possibilities and backtracking when a solution path is invalid.
  - Example: N-Queens problem.
- Dynamic Programming:
  - Uses recursion to solve problems by breaking them into overlapping subproblems and reusing previous results.
  - Example: Fibonacci sequence.
- Tree and Graph Structures:
  - Utilizes recursion for efficient traversal and manipulation of tree or graph structures.
  - Example: Binary tree traversal (In-order, Pre-order, Post-order).
- Simplification and Readability:
  - Makes code concise and easy to understand by breaking problems into simpler subproblems.
  - Example: Solving combinatorial problems like generating permutations or subsets.

End of chapter 5