# ECE 220:Computer Systems and Programming
# Lecture 12: Recursion, recursive sorting and Recursion with Backtracking

Instructor: Bruce X.B. Yu

Time: Monday, 9:00-10:20 AM (EE), 10:30-11:50 AM (CompE)
Friday, 13:00-14:20 PM (EE), 14:30-15:50 PM (CompE)

Location: LT Building North A 418/420

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics.
- Like iteration -- can be used interchangeably;
  sometimes recursion results in a simpler solution.
- Must have at least 1 **base case** (terminal case) that ends the recursive process.

Example: n!

# Factorial:

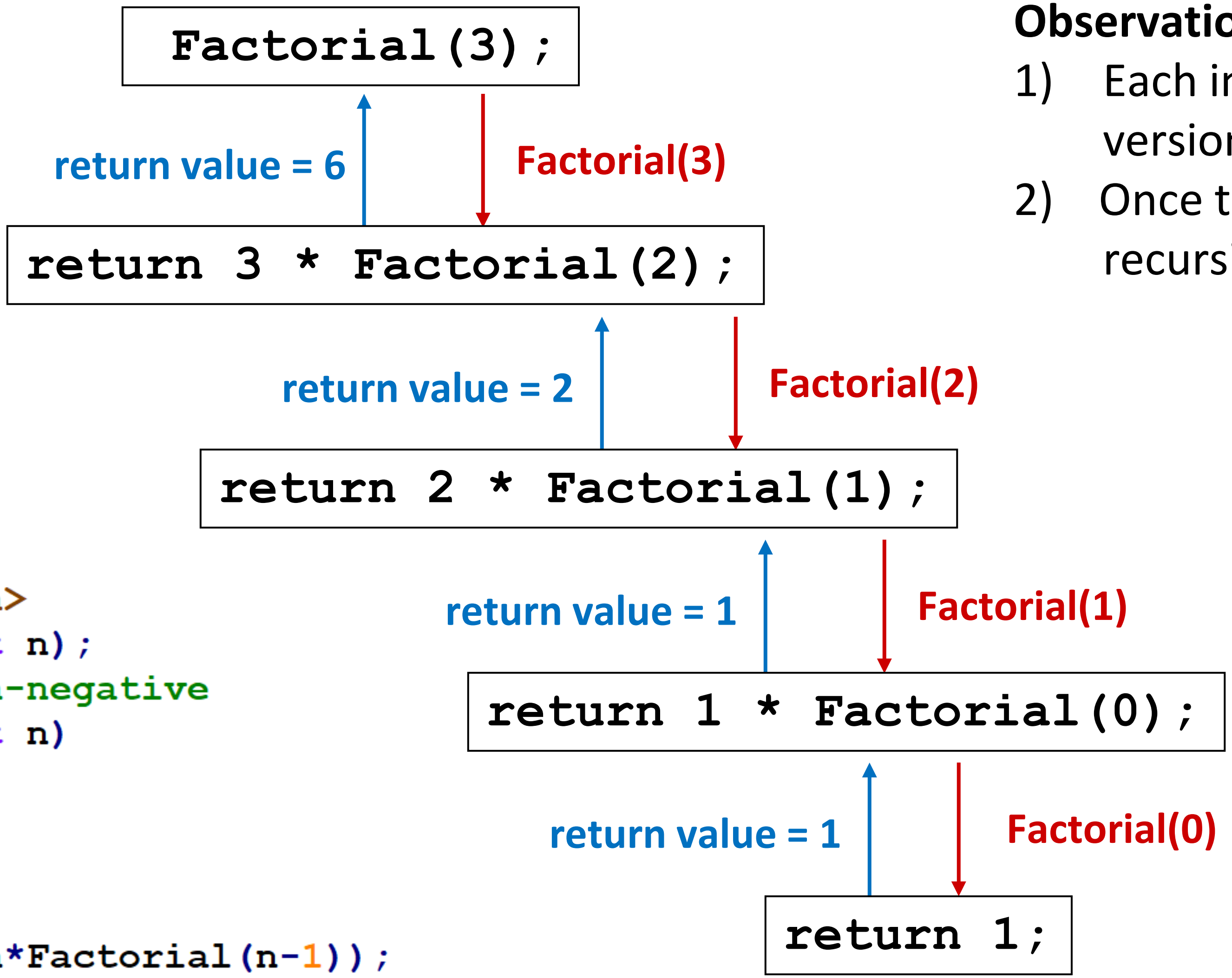
$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$

$$n! = \begin{cases} n \cdot (n-1)! & , n > 0 \\ 1 & , n = 0 \end{cases}$$

```
int Factorial(int n)
{

      if
       Return  ….


 else

 return

}
```

```c
#include <stdio.h>
int Factorial(int n);
//assume n is non-negative
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return (n*Factorial(n-1));
}

int main()
{
    int n=3;
    int result = Factorial(n);
    printf("Factorial(%d)=%d \n",n,result);

    return 0;
}
```

# Executing Factorial




**Observation:**

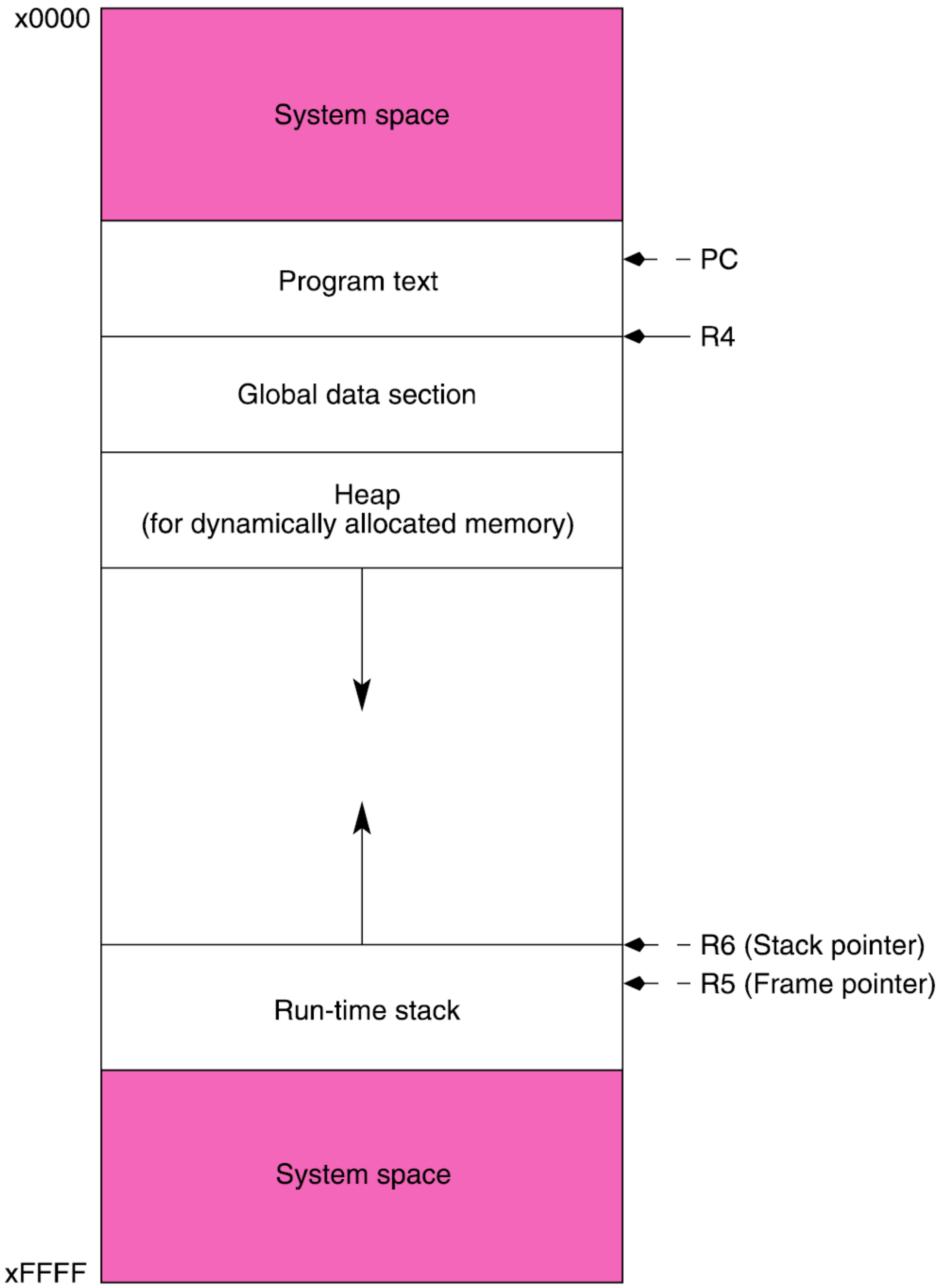
1) Each invocation solves a smaller version of the problem;
2) Once the base case is reached, recursive process stops.

```c
#include <stdio.h>
int Factorial(int n);
//assume n is non-negative
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return (n*Factorial(n-1));
}
```

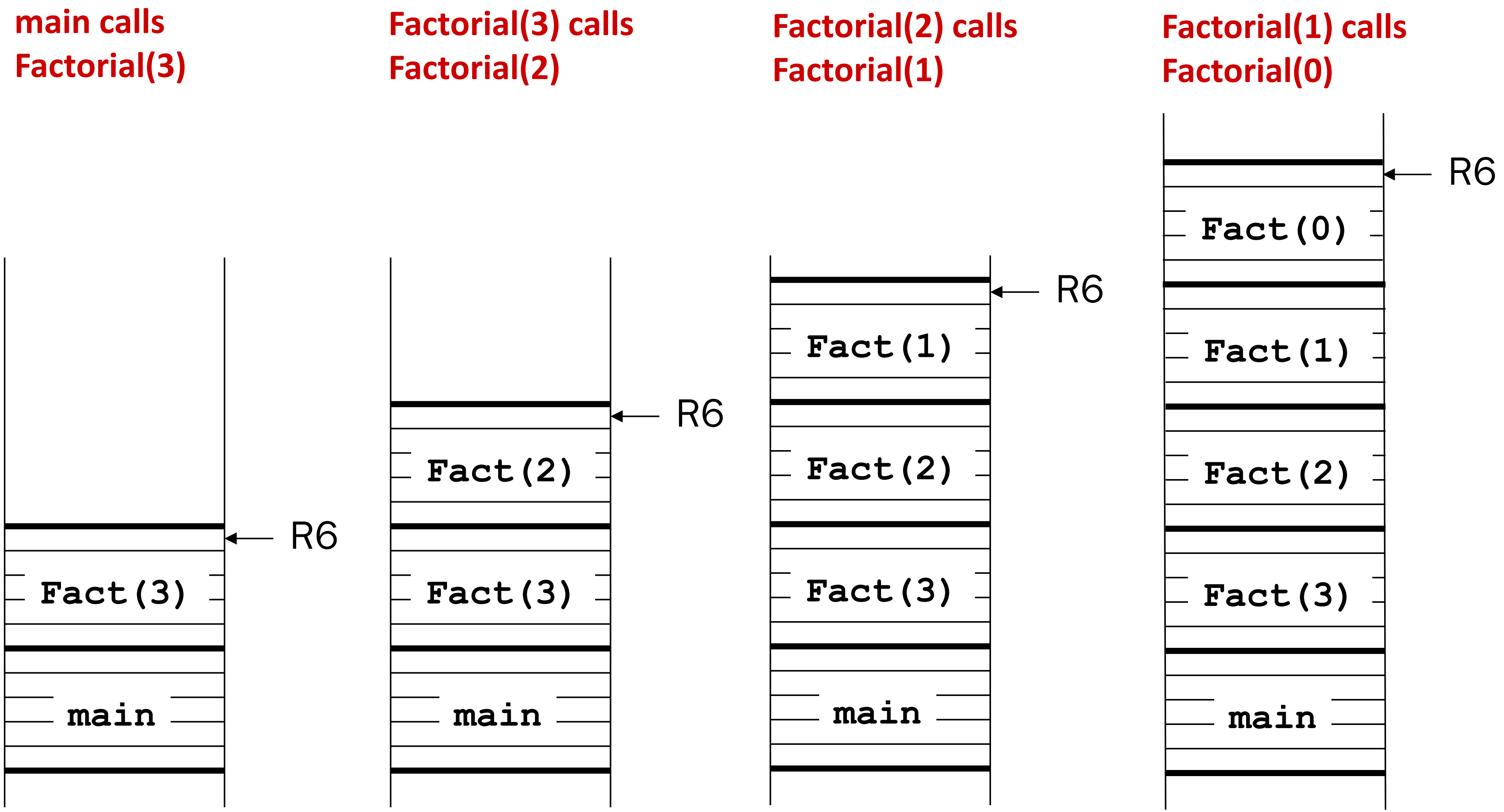
# Space for Variables

1. Global data section
   (global variables)
2. Run-time stack
   (local variables)

- R4 (global pointer) points the first global variable
- R5 (frame pointer) points the first local variable
- R6 (stack pointer) points the top of run-time stack



# Run-Time Stack During Execution of Factorial

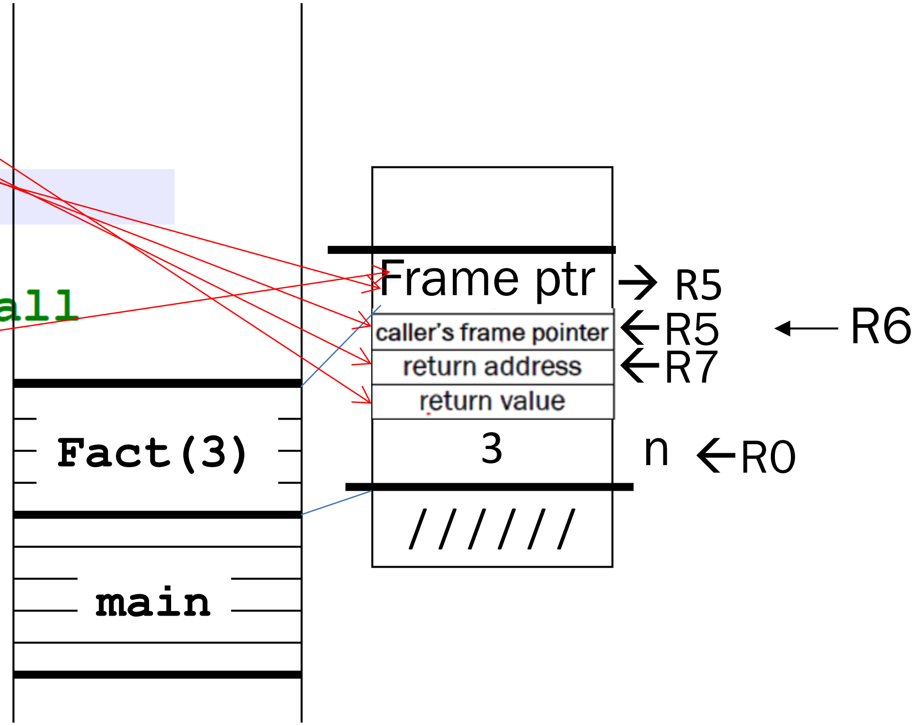


# C to LC3 implementation of n! (test case n=3)

```
.ORIG x3000
; push argument
    LD R6, STACK_TOP
    AND R0,R0,#0
    ADD R0,R0,#3
    ADD R6,R6,#-1  ;R6 <- R6-1;
    STR R0,R6,#0   ;push argument n
; call subroutine
    JSR FACTORIAL
; pop return value from run-time stack (to R0)
    LDR R0,R6,#0
    ADD R6, R6, #2
;Store the result
    STR R0,R6,#0   ;dump the result at STACK_TOP
    HALT
```

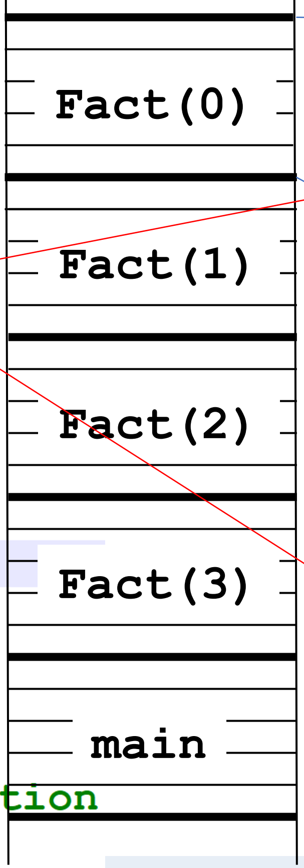
```
FACTORIAL:
; push callee's bookkeeping info onto the run-time stack
; allocate space in the run-time stack for return value
    ADD R6, R6, #-1
; store caller's return address and frame pointer
    ADD R6, R6, #-1
    STR R7, R6, #0
    ADD R6, R6, #-1
    STR R5, R6, #0
; Update frame pointer for the callee
    ADD R5, R6, #-1

; if (n>0)
    LDR R1, R5, #4
    ADD R2, R1, #-1     R2 ← 2n-1
    BRn ELSE
; compute fn = n * factorial(n-1)
; caller-built stack for factorial(n-1) function call
; push n-1 onto run-time stack
    ADD R6, R6, #-1
    STR R2, R6, #0
; call factorial subroutine
    JSR FACTORIAL
; pop return value from run-time stack (to R0)
    LDR R0, R6, #0
    ADD R6, R6, #1
```

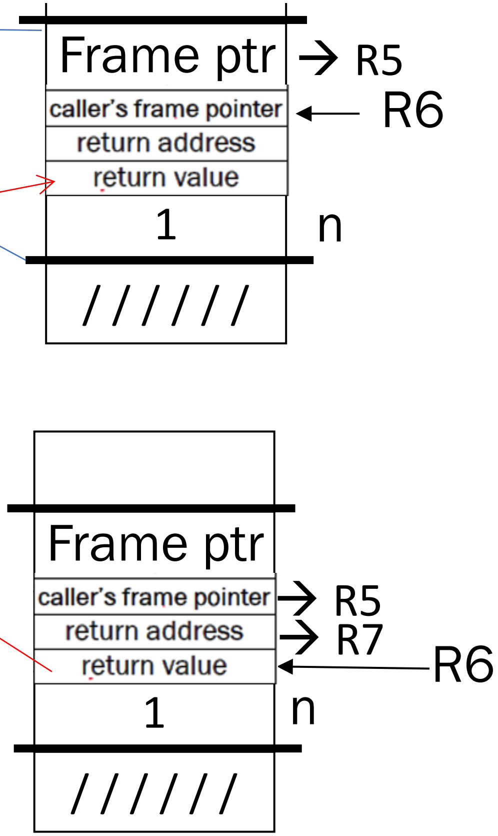
```c
#include <stdio.h>
int Factorial(int n);
//assume n is non-negative
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return (n*Factorial(n-1));
}

int main()
{
    int n=3;
    int result = Factorial(n);
    printf("Factorial(%d)=%d \n",n,result);

    return 0;
}
```



```
; call factorial subroutine
   JSR FACTORIAL
; pop return value from run-time stack (to R0)
   LDR R0, R6, #0
   ADD R6, R6, #1
; pop function argument from the run-time stack
   ADD R6, R6, #1
; multiply n by the return value (already in R0)
   LDR R1, R5, #4
   ;MUL R2, R0, R1 ; R2 <- n * factorial(n-1)
   ST R7, SAVE_R7
   JSR MULT
   LD R7, SAVE_R7
   ADD R0, R2, #0
   BRnzp RETURN
ELSE:
; store value of 1 in to the memory of return value
   AND R0, R0, #0
   ADD R0, R0, #1
; tear down the run-time stack and return
RETURN:
; write return value to the return entry
   STR R0, R5, #3
; pop local variable(s) from the run-time stack
   ;no local variable for this implementation
; restore caller's frame pointer and return address
   LDR R5, R6, #0
   ADD R6, R6, #1
   LDR R7, R6, #0
   ADD R6, R6, #1 ;stack pointer is at the return value location
; return control to the caller function
   RET
```

```c
#include <stdio.h>
int Factorial(int n);
//assume n is non-negative
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return (n*Factorial(n-1));
}

int main()
{
    int n=3;
    int result = Factorial(n);
    printf("Factorial(%d)=%d \n",n,result);

    return 0;
}
```



```
; multiply subroutine
; input should be in R0 and R1
; output should be in R2
MULT
    ; save R3
    ST R3, SAVE_R3
    ; reset R2 and initialize R3
    AND R2, R2, #0
    ADD R3, R0, #0
    ; perform multiplication
    MULT_LOOP
    ADD R3, R3, #-1
    BRn MULT_DONE
    ADD R2, R2, R1
    BRnzp MULT_LOOP
    MULT_DONE
    ; restore R0
    LD R3, SAVE_R3
    RET

SAVE_R3         .BLKW #1
SAVE_R7         .BLKW #1
STACK_TOP       .FILL x4000
.END
```

# Recursive Binary Search

```c
// C program to implement binary search using recursion
#include <stdio.h>

// A recursive binary search function. It returns location
// of x in given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    // checking if there are elements in the subarray
    if (r >= l) {

        // calculating mid point
        int mid = (l + r) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it can only
        // be present in left subarray
        if (arr[mid] > x) {
            return binarySearch(arr, l, mid - 1, x);
        }

        // Else the element can only be present in right
        // subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}
```

Ref: https://www.geeksforgeeks.org/binary-search/#

# Fibonacci Series

$$f(n) = f(n-1) + f(n-2)$$

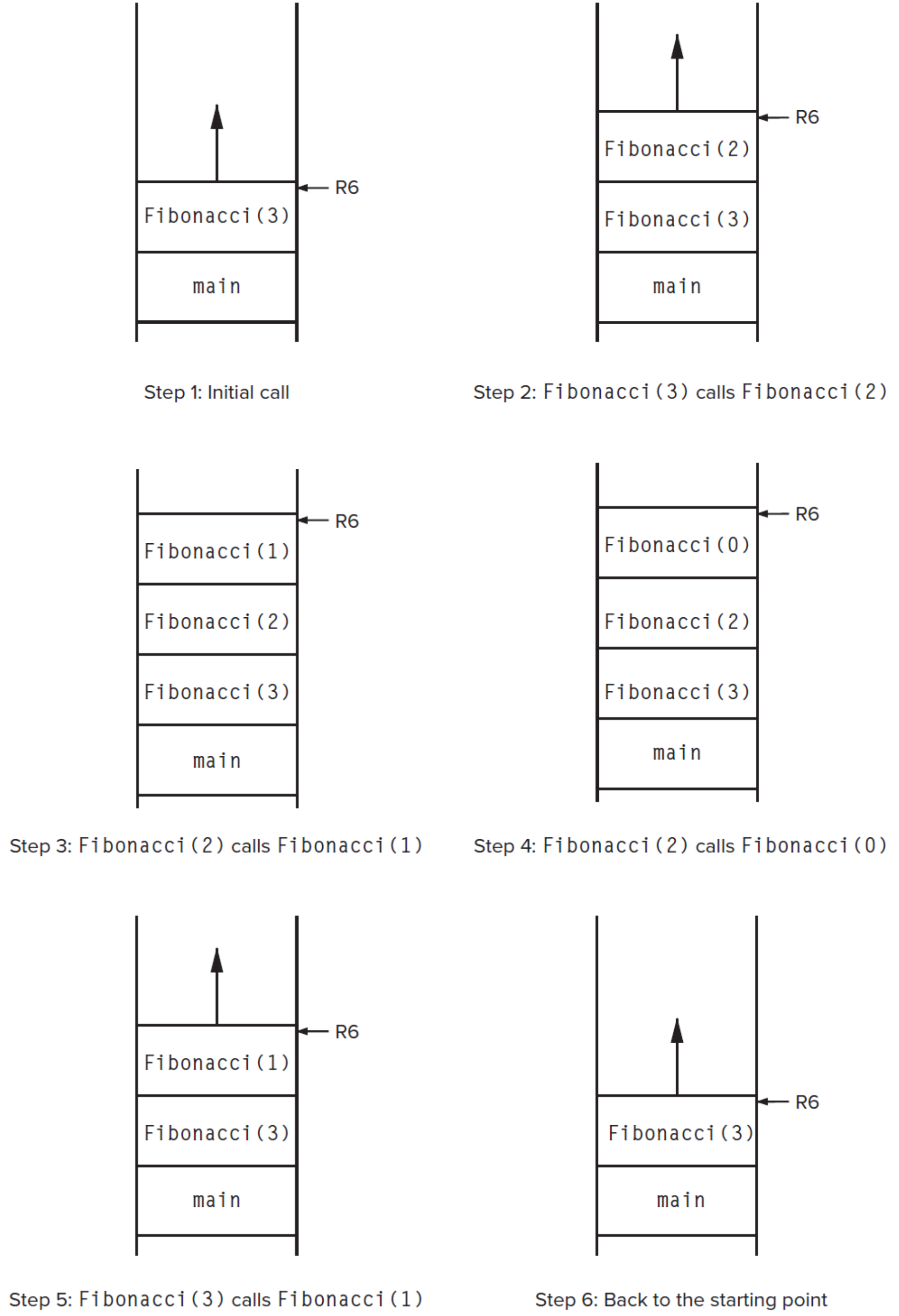
$$f(1) = 1$$

$$f(0) = 1$$

```c
#include <stdio.h>

int Fibonacci(int n);

int main(void)
{
    int in;
    int number;

    printf("Which Fibonacci number? ");
    scanf("%d", &in);

    number = Fibonacci(in);
    printf("That Fibonacci number is %d\n", number);
}

int Fibonacci(int n)
{
    int sum;

    if (n == 0 || n == 1)
        return 1;
    else {
        sum = (Fibonacci(n-1) + Fibonacci(n-2));
        return sum;
}
```

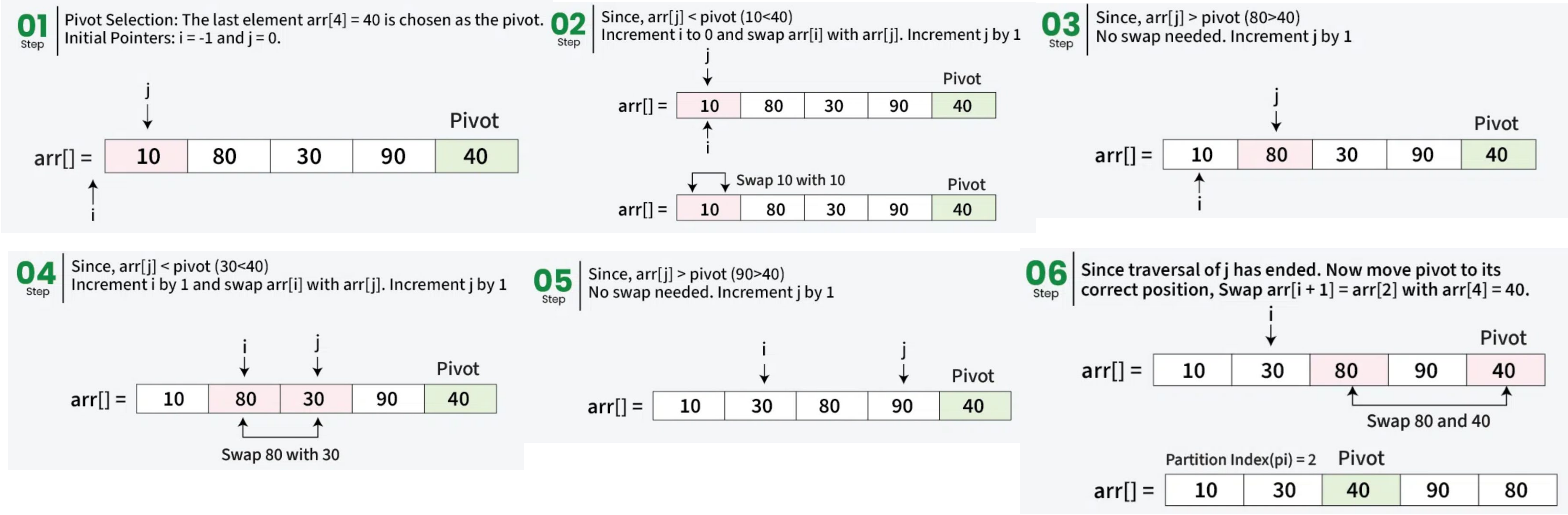
```c
#include <stdio.h>

int Fibonacci(int n);

int main(void)
{
    int in;
    int number;

    printf("Which Fibonacci number? ");
    scanf("%d", &in);

    number = Fibonacci(in);
    printf("That Fibonacci number is %d\n", number);
}

int Fibonacci(int n)
{
    int sum;

    if (n == 0 || n == 1)
        return 1;
    else {
        sum = (Fibonacci(n-1) + Fibonacci(n-2));
        return sum;
}
```

Consider, n=3



# int partition(int array[], int l, int h)

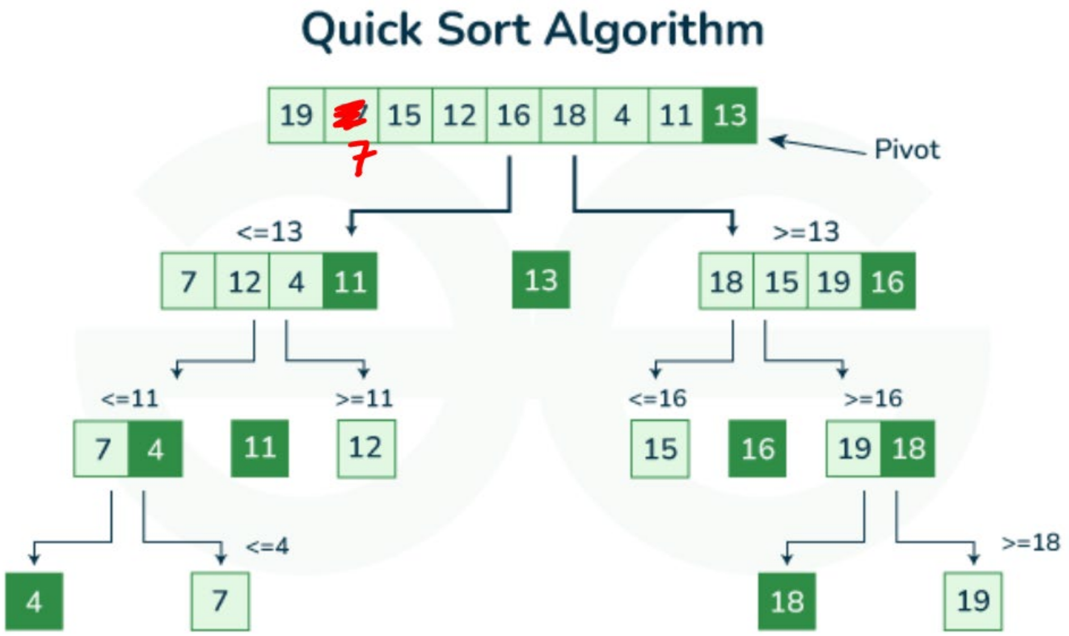
ZJUI




*The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as **i** . While traversing, if we find a smaller element, we swap the current element with **arr[i]**. Otherwise, we ignore the current element.*

# Recursive Quick Sort

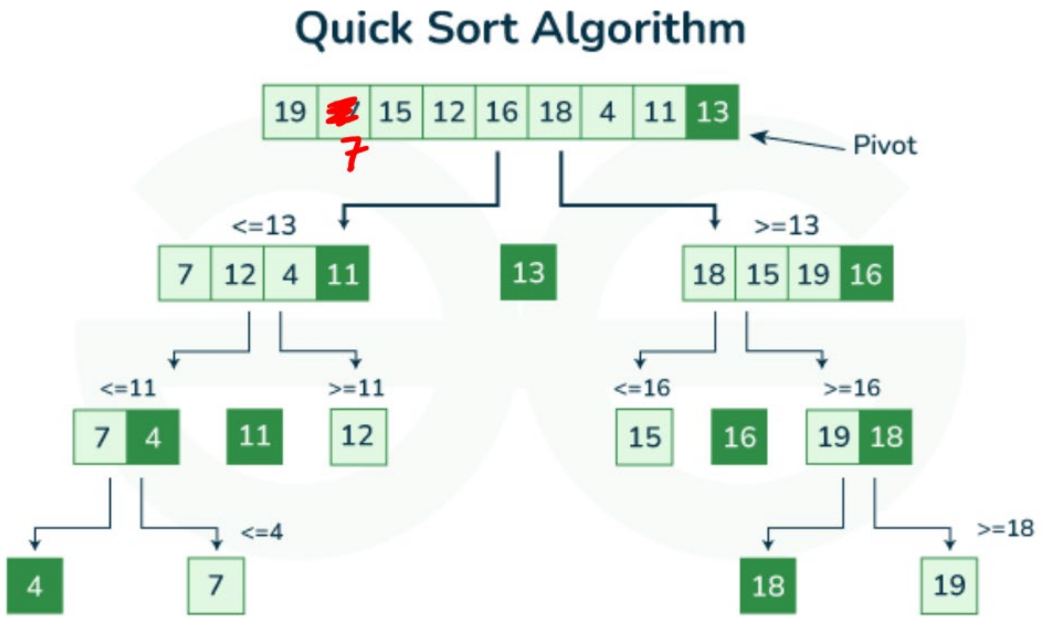
```
void swap(int *a, int *b)
{
    int tmp;
    tmp=*a;
    *a=*b;
    *b=tmp;
}

void print(int array[], int size)
{
    int i;
    for(i=0;i<size;i++)
    printf("%d ", array[i]);
        printf("\n");
}

int main()
{
    int size=8;
    int l=0;
    int h=size-1;
    int array[8]={10, 20, 80, 30,100, 90, 15, 40};
    quickSort(array,l,h);
    print(array, size);
    return 0;
```




https://www.geeksforgeeks.org/quick-sort-in-c/

# Recursive Quick Sort (cont.)

```
void quickSort(int array[], int l, int h)
{
    if (l<h){
    // Call partion() to get the partition index
    int p=partition(array, l, h);
    // Call partion() to partition the left side
    quickSort(array, l, p-1);
    // Call partion() to partition the right side
    quickSort(array, p+1, h);
    }

}
```




```
int partition(int array[], int l, int h)
{
int i=l-1;
int x=array[h];
int j;
for (j=l;j<=h-1;j++){
    if (array[j]<=x)
    {
        i++;
        swap(&array[i], &array[j]);

    }

}
swap(&array[i+1], &array[h]);
return (i+1);
}
```

```c
void bubble_recursion(int *array, int n)
{
if(n==0)
return;
int i, temp, swap = 0;

    //sort number in ascending order

        swap = 0;
        for(i=0;i<n;i++)
        {
            //swap the two numbers if order is incorrect
            if(array[i]>array[i+1])
            {
                temp = array[i];
                array[i] = array[i+1];
                array[i+1] = temp;
                //set the swap flag
                swap = 1;
            }
        }
n--;
if (swap==0)
return;
bubble_recursion(array, n);
}
```

```c
#include <stdio.h>
#define SIZE 8
void bubble_recursion(int *array, int n);

int main()
{
    int n = SIZE-1;
    int array[] = {6,5,3,1,8,7,2,4};
int i;
bubble_recursion(array, n);

    printf("sorted array: \n");
    for(i=0;i<SIZE;i++){
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

# Recursive Merge-Insertion Sort

```c
#include <stdio.h>
#define SIZE 8
void merge_recursion(int *array, int i);
int main()
{
    int i=1;
    int array[] = {6,5,3,1,8,7,2,4};
merge_recursion(array, i);
    printf("sorted array: \n");
    for(i=0;i<SIZE;i++){
        printf("%d ", array[i]);
    }
    printf("\n");
    return 0;
}
```

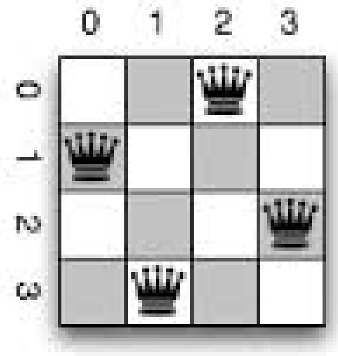
```c
void merge_recursion(int *array, int i)
{
if (i==SIZE)
return;
int j, temp;
        temp = array[i];
        for(j=i-1;(j>=0 && (temp < array[j]));j--)
        {
            //shift element to the right
            array[j+1] = array[j];
        }
        //insert at the proper location
        array[j+1] = temp;
i++;
merge_recursion(array, i);
}
```

## Recursive Backtracking:

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem statement.

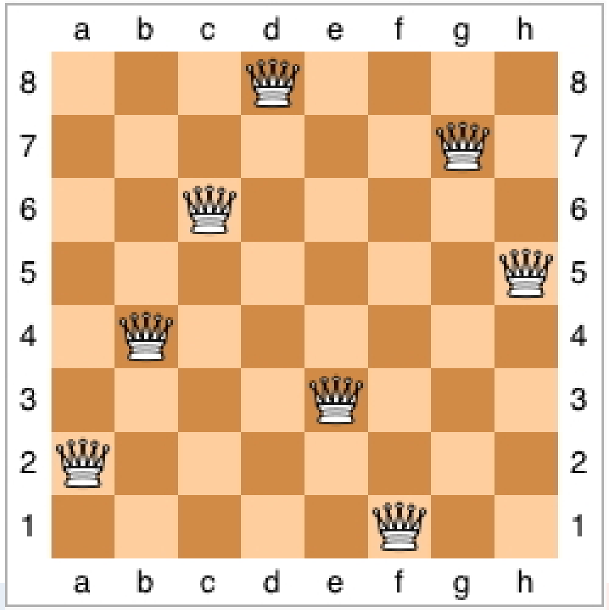
# N queens problem using recursive Backtracking

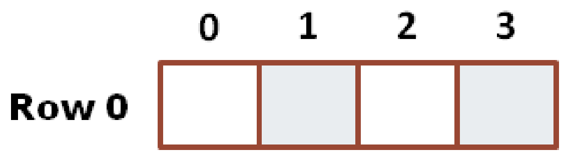
- Place N queens on an NxN chessboard so that none of the queens are under attack;

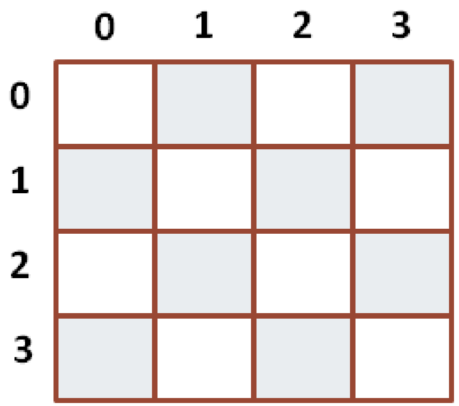



| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |

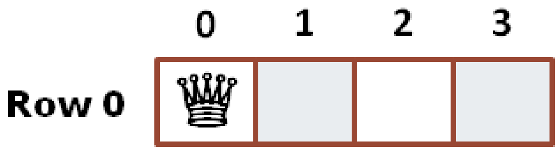
- Brute force: total number of possible placements:
- ~$N^2$ Choose N ~ 4.4 B (N=8)

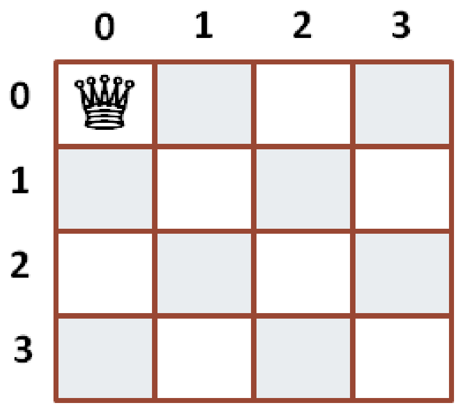




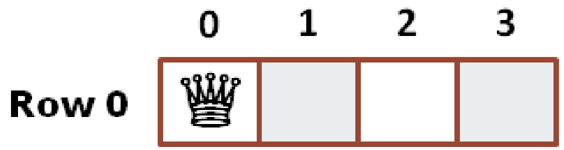

QueenPosition[ ]

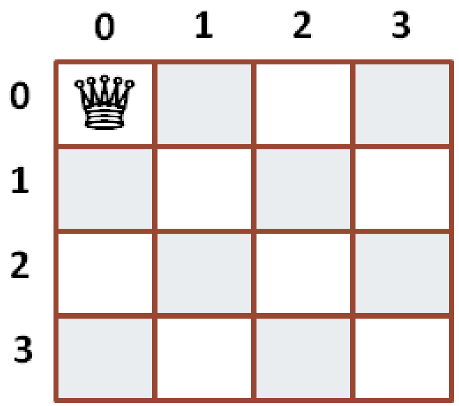



QueenPosition[ ]

Place **0**th Queen on the **0**th Column of **0**th Row

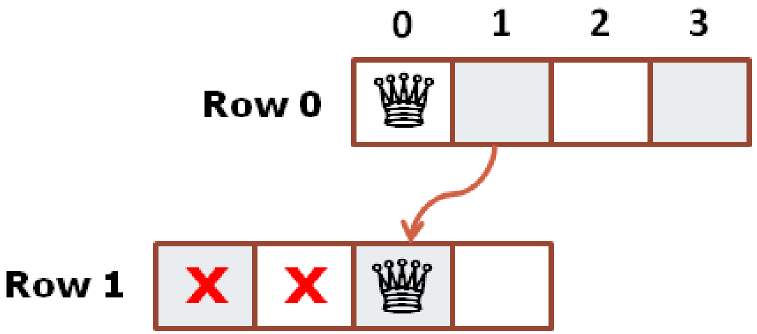


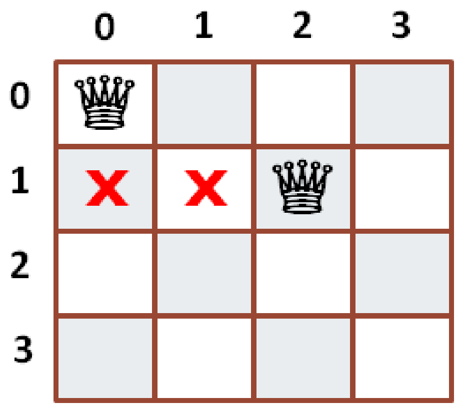




QueenPosition[ ]

**Row 0** (0,0)

Add **0**$^{th}$ Queen’s position to position array



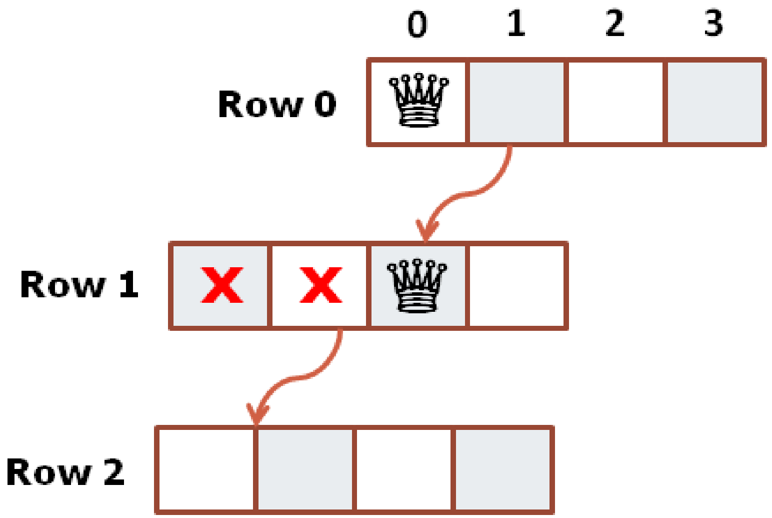



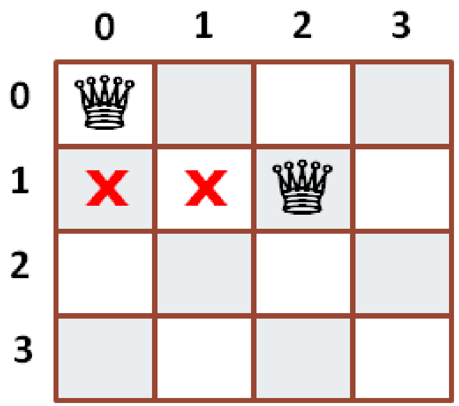

QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,0) |
| **Row 1** | (1,2) |

Go to the next level of recursion.

Place the **1**$^{st}$ queen on the **1**$^{st}$ row such that she does not attack the **0**$^{th}$ queen and add that to Positions.

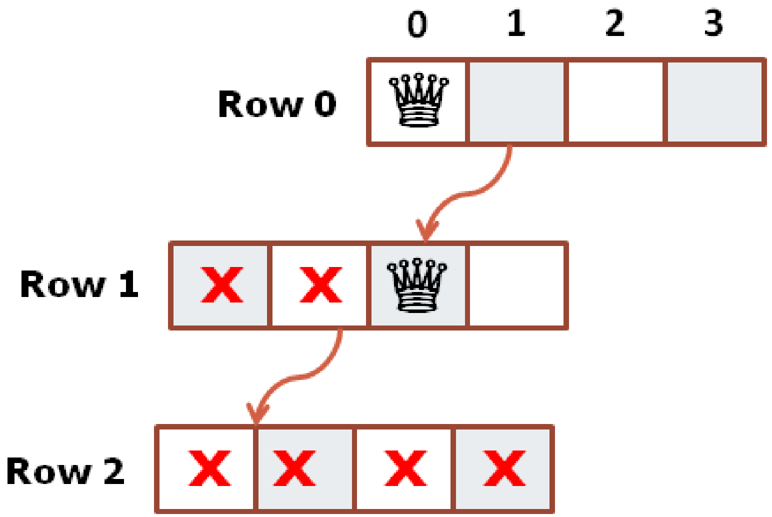


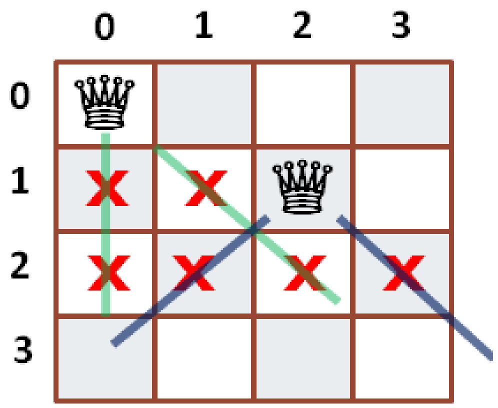




QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,0) |
| **Row 1** | (1,2) |

In the next level of recursion, find the cell on **2**$^{nd}$ row
such that it is not under attack from any of the available queens.

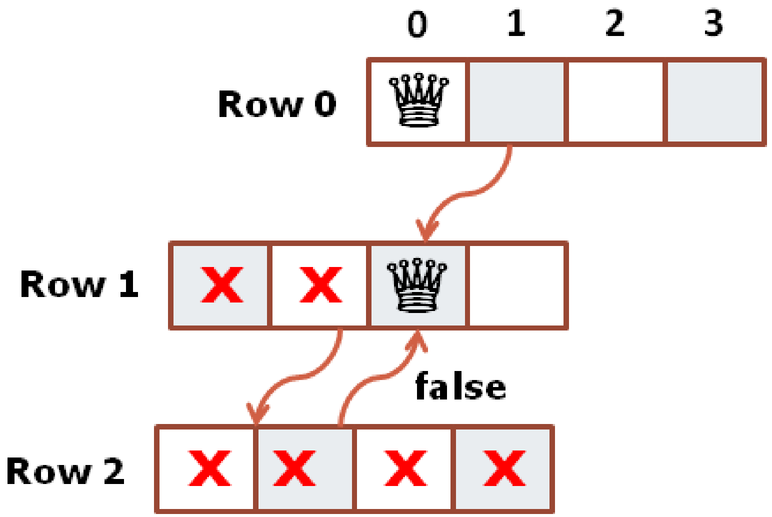


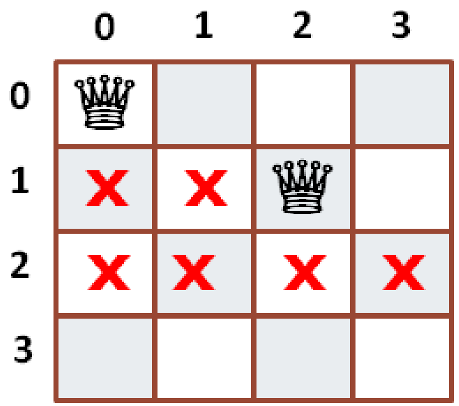




QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,0) |
| **Row 1** | (1,2) |

But cell **(2,0)** and **(2,2)** are under attack from **0**th queen and cell **(2,1)** and **(2,3)** are under attack from **1**st queen.

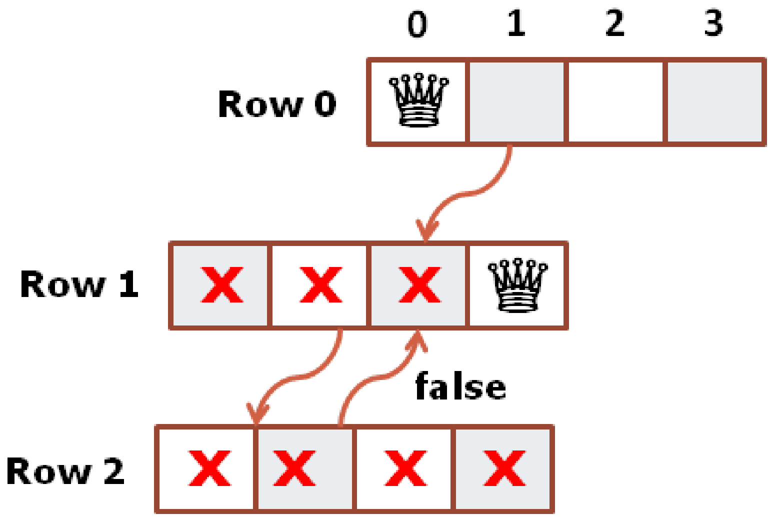


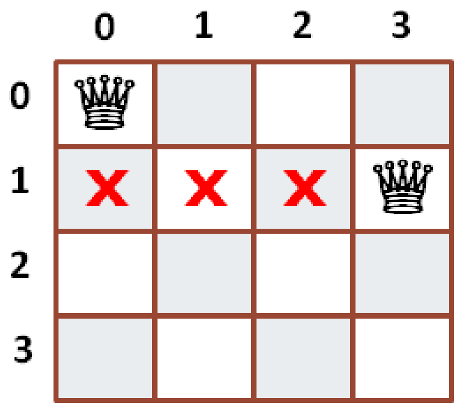




QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,0) |
| **Row 1** | (1,2) |

So function will return false to the calling function.



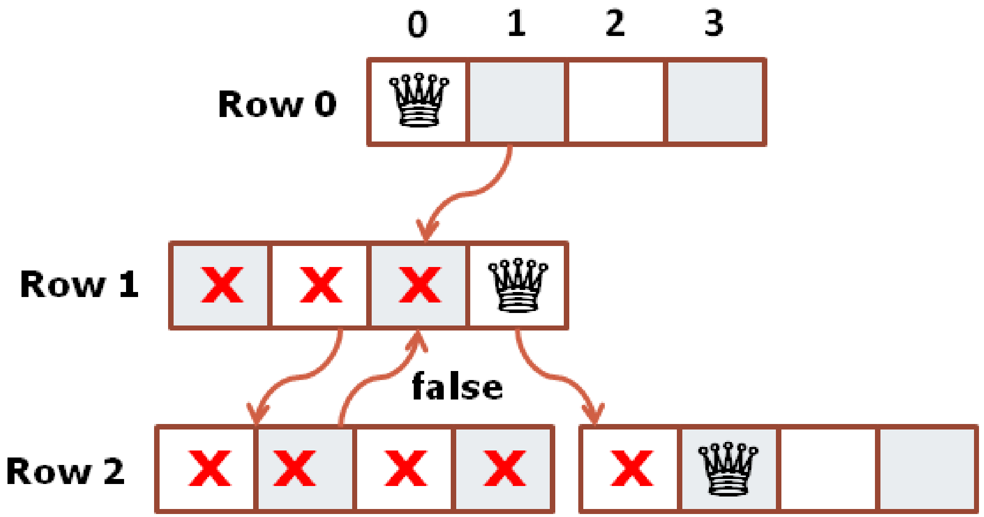

QueenPosition[ ]

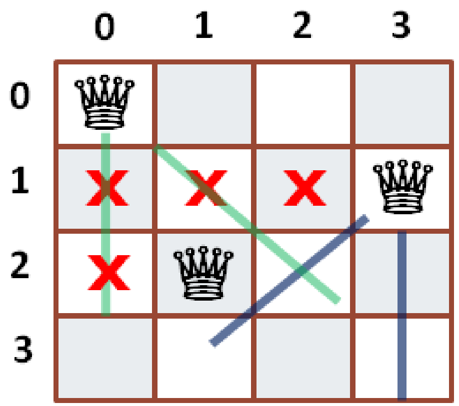
**Row 0** (0,0)

~~**Row 1** (1,2)~~

**Row 1** (1,3)

Calling function will try to find next possible place for the
**1**st queen on **1**st row and update the queen position in position array.



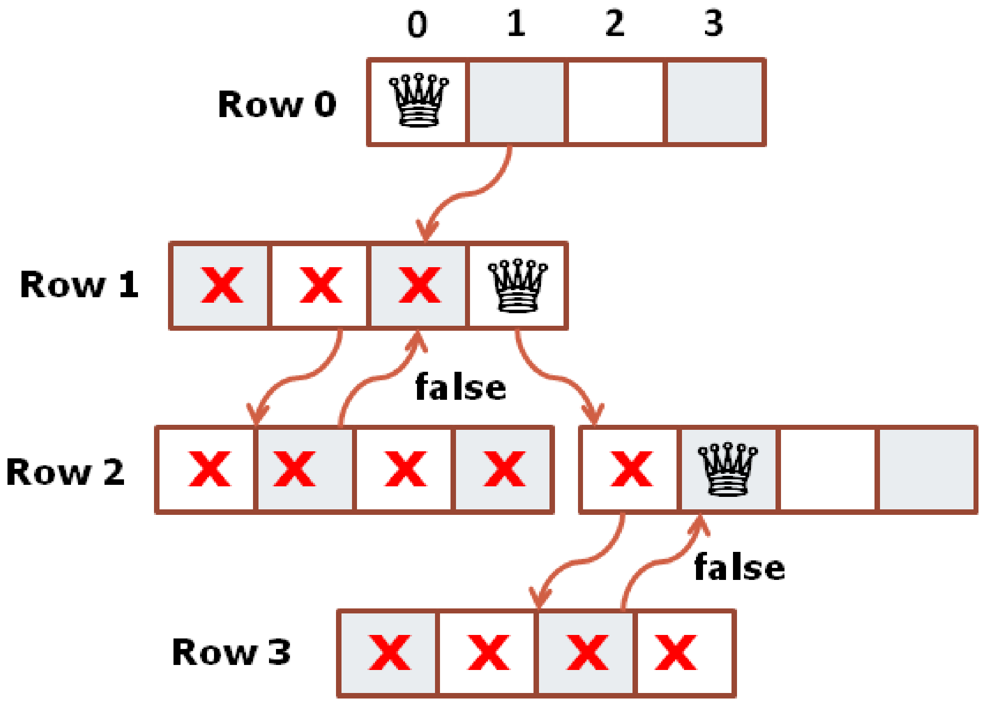



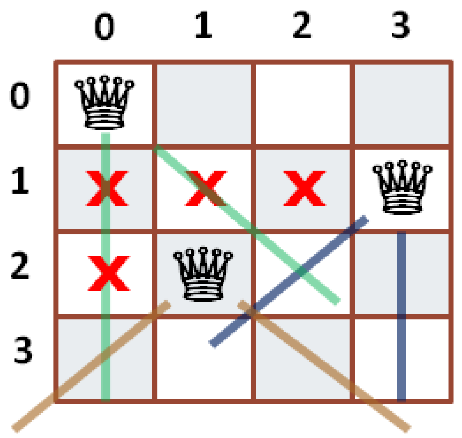

QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,0) |
| **Row 1** | (1,3) |
| **Row 1** | (2,1) |

Again find the cell on **2nd** row such that
it is not under attack from any of the available queens.

Placing the queen in cell **(2,1)** as it is not
under attack from any of the queen.

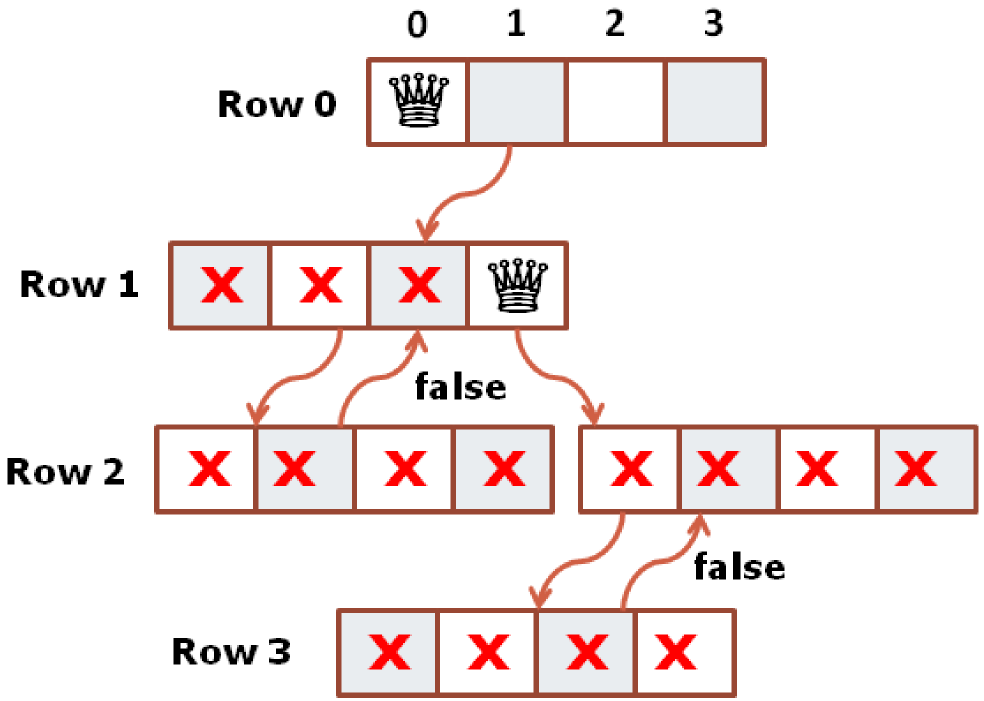


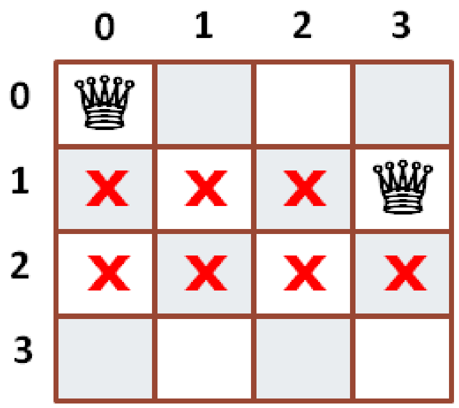




QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,0) |
| **Row 1** | (1,3) |
| **Row 1** | (2,1) |

For **3**rd queen, no safe cell is available on **3**rd row.
So function will return false to calling function.



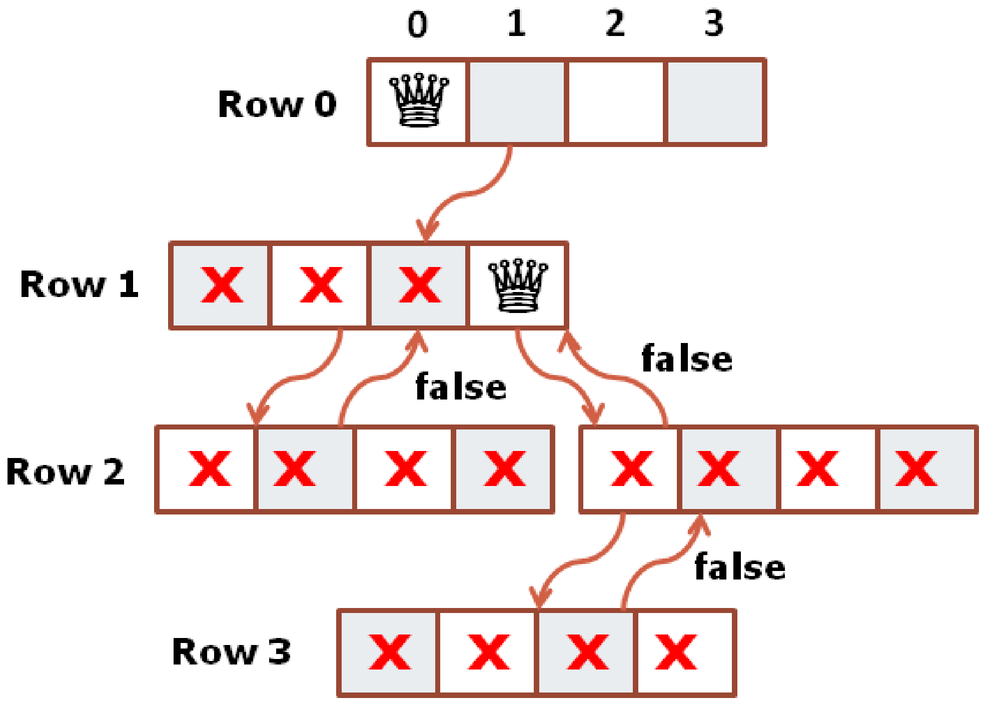

QueenPosition[ ]

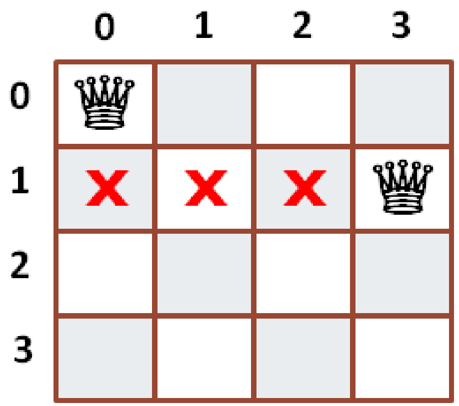
**Row 0** (0,0)

**Row 1** (1,3)

~~**Row 1** (2,1)~~

Queen at the **2**nd row tries to find next safe cell.

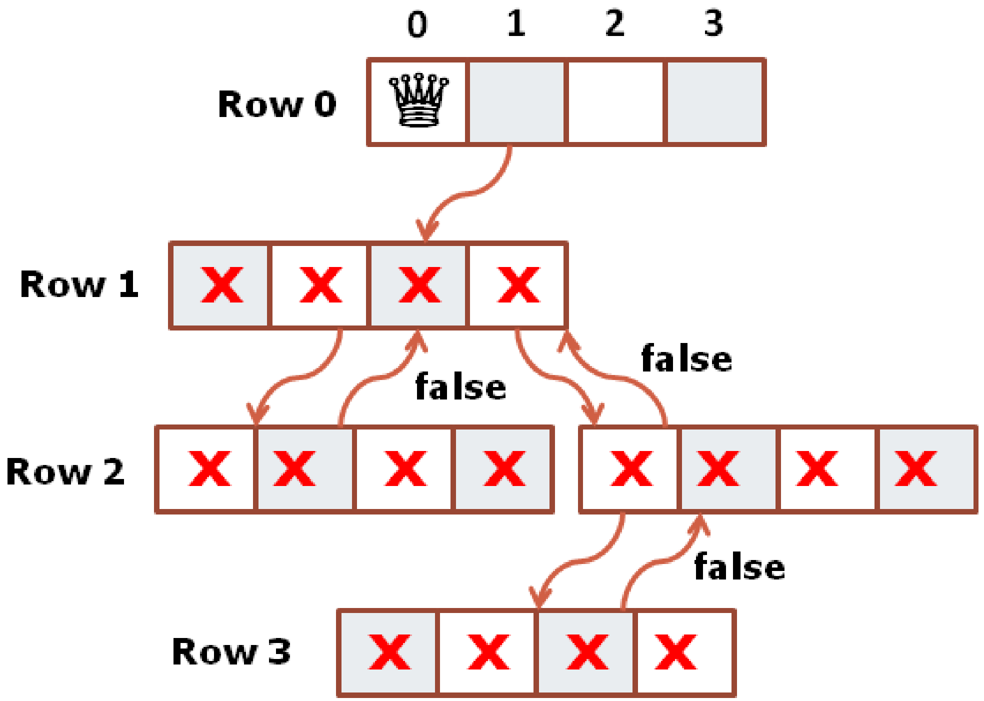


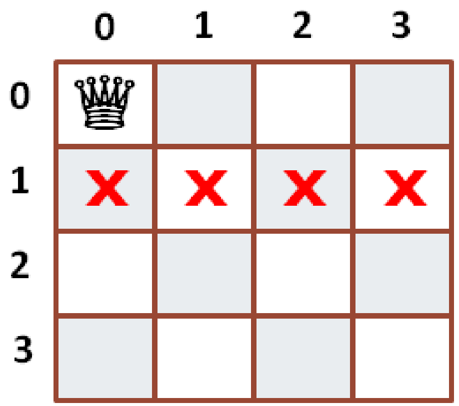




QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,0) |
| **Row 1** | (1,3) |

But as both remaining cells are under attack from other queens, this function also returns false to its calling function.



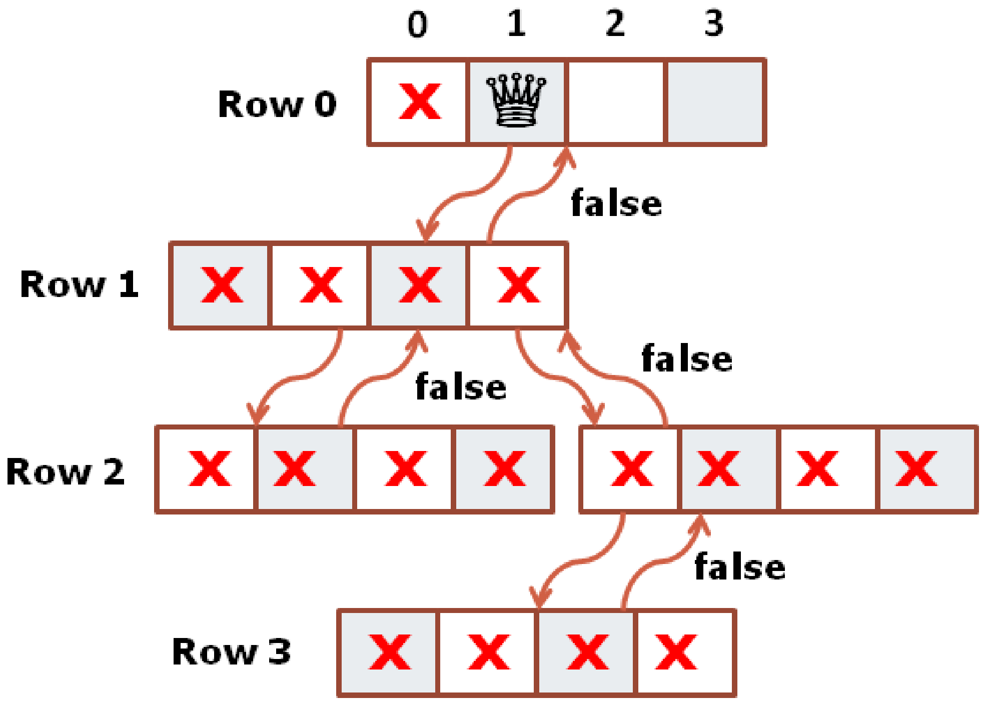



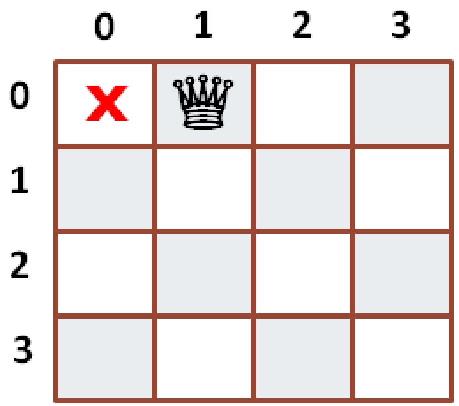

QueenPosition[ ]

**Row 0** (0,0)

~~**Row 1** (1,3)~~

Queen at the **1**$^{st}$ row tries to find next safe cell.
But as queen is in the last cell, it will retuen false to
Its calling function.

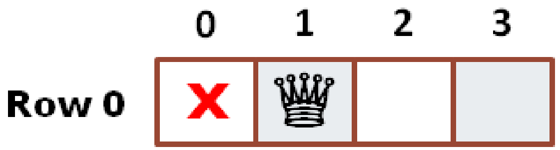


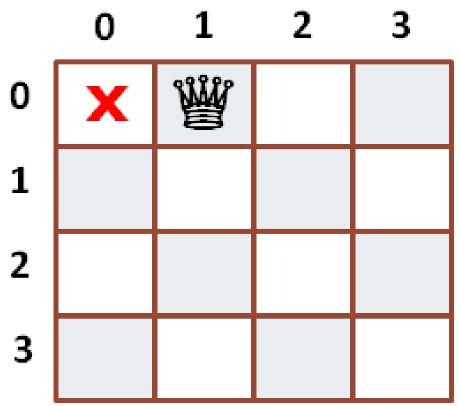




QueenPosition[ ]

~~**Row 0** (0,0)~~
**Row 0** (0,1)

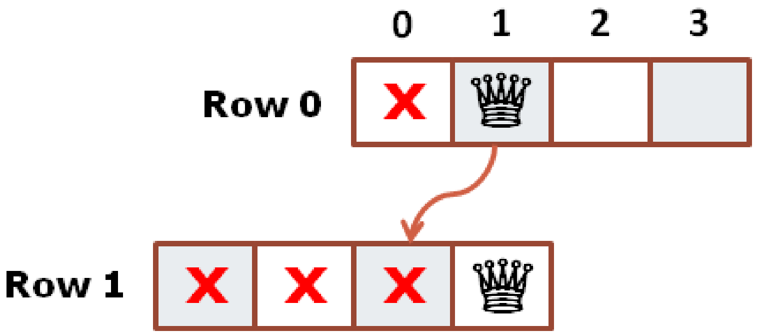
Queen at the **1**st row tries to find next safe cell.
Let us remove these failed recursion calls from the screen.

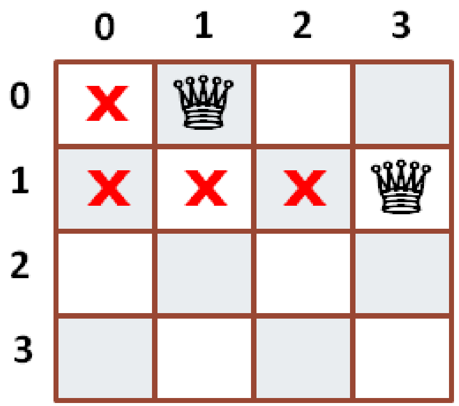



QueenPosition[ ]

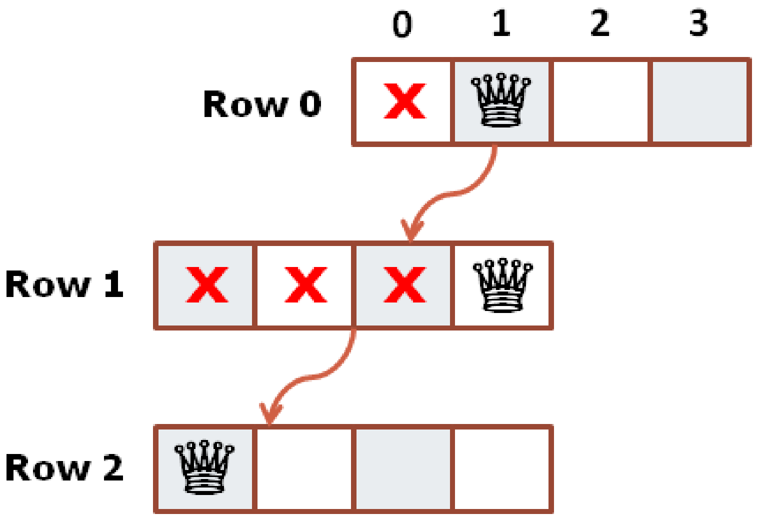
**Row 0** (0,1)

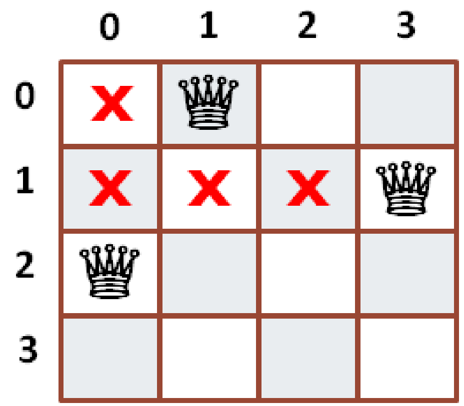



QueenPosition[ ]

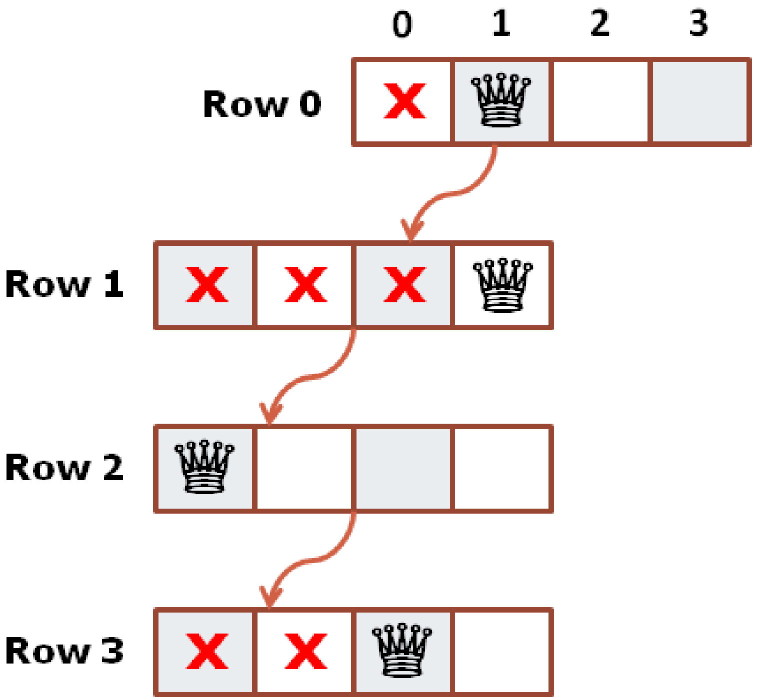
| | |
|---|---|
| **Row 0** | (0,1) |
| **Row 1** | (1,3) |

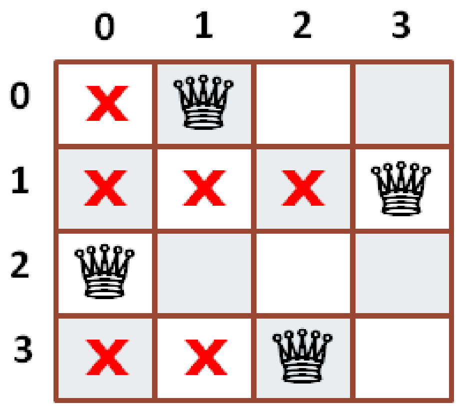



QueenPosition[ ]

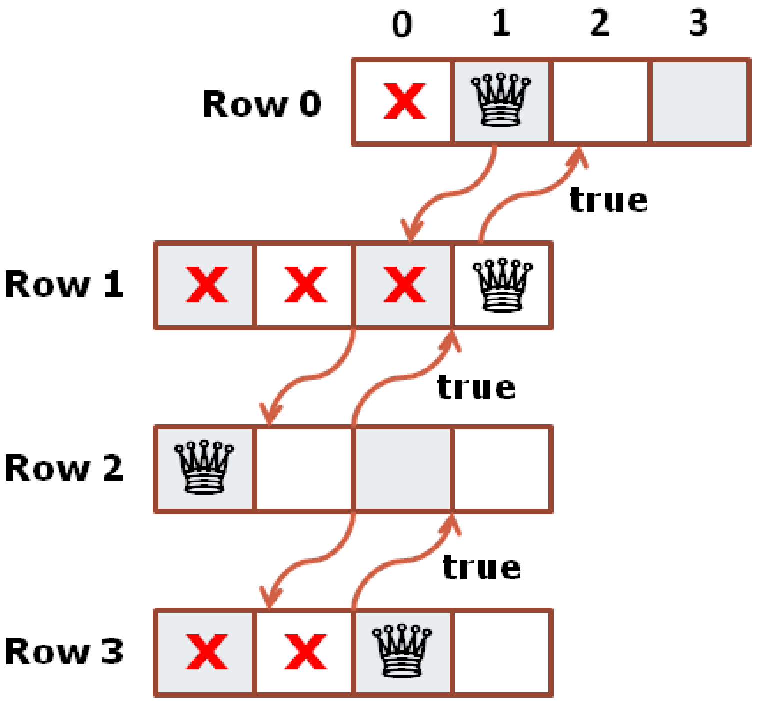
| | |
|---|---|
| **Row 0** | (0,1) |
| **Row 1** | (1,3) |
| **Row 2** | (2,0) |

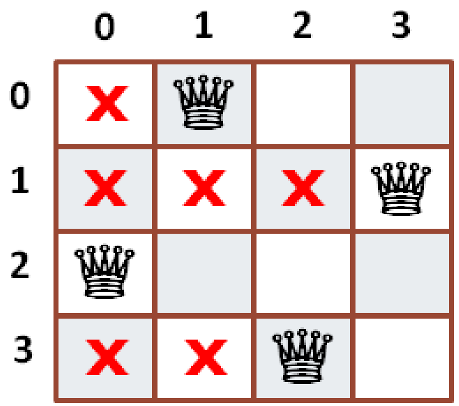



QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,1) |
| **Row 1** | (1,3) |
| **Row 2** | (2,0) |
| **Row 3** | (3,2) |




QueenPosition[ ]

| | |
|---|---|
| **Row 0** | (0,1) |
| **Row 1** | (1,3) |
| **Row 2** | (2,0) |
| **Row 3** | (3,2) |

All functions will return true to their calling function.
It means all queens are placed on the board such that they are not attacking each other.

# N Queens with backtracking



- int board[N][N] represents placement of queens
  - board[i][j] = 0: no queen at row i column j
  - board[i][j] = 1:queen at row i column j

- Initialize, for all i,j board[i][j] = 0

- Functions
  - PrintBoard(board): Prints board on the screen
  - IsSafe(borad, row, col): returns 1 if new queen can be placed at (row,col) in board
  - Solve(board, row: recursively attempts to place (N-row) queens; returns 0 if it fails

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

Initial board

Solve(board,3) returns 0

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |

# Recursive with Backtracking

- N-Queen Problem by Backtracking
  1. **Decision**
     Place a queen at a safe place.
  2. **Recursion**
     Explore the solution for the next row.
  3. **Backtrack (Undo)**
     Remove the queen if no solution for the next row.
  4. **Base case**
     Reach the goal.

N-Queen (4x4) Backtracking – CODE (Main function)

```c
#include <stdio.h>

//Solve 4x4 n Queen problem using recursion with backtracking

#define N 4
#define true 1
#define false 0

void printSolution(int board[N][N]);
int Solve(int board[N][N], int col);
int isSafe(int board[N][N], int row, int col);

int main()
{
    int board[N][N] = {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};

    //game started at row 0
    if(Solve(board,0) == false)
    {
        printf("Solution does not exist.\n");
        return 1;
    }

    printf("Solution: \n");
    printSolution(board);
    return 0;
}
```

## N-Queen (4x4) Backtracking – CODE (Solve function)

```c
int Solve(int board[N][N], int row)
{
    //base case
    if(row>=N)
        return true;

        //find a safe column(j) to place queen
    int j;
    for(j=0;j<N;j++)
    {
        //column j is safe, place queen here
        if(isSafe(board, row, j) == true)
        {
            board[row][j]=1;
            printf("Current Play: \n");
                printSolution(board);

            //increment row to place the next queen
            if(Solve(board, row+1) == true)
                return true;
            //attempt to place queen at row+1 failed,-
            //backtrack to row and remove queen
            board[row][j]=0;
            printf("Backtrack: \n");
            printSolution(board);
        }
    }
    return false;
}
```

# N-Queen (4x4) Backtracking – CODE (isSafe & PrintSolution functions)

```
int isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for(i=0;i<row;i++)
    {
        for(j=0;j<N;j++)
        {
            //check whether there's a queen at the same column or the 2 diagonals
            if(((j==col) || (i-j == row-col) || (i+j == row + col)) && (board[i][j]==1))
                return false;
        }
    }
    return true;
}


void printSolution(int board[N][N])
{
    int i,j;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}
```

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |

# Q&A



Thank You!