

Recursion

Computing Lab

`http://www.isical.ac.in/~dfslab`

A recursive function is a function that calls itself.

- The task should be decomposable into sub-tasks that are smaller, but otherwise identical in structure to the original problem.
- The simplest sub-tasks (**called the base case**) should be (easily) solvable directly, i.e., without decomposing it into similar sub-problems.

Recursive vs. iterative implementations

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

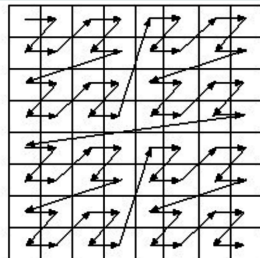
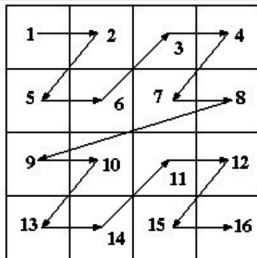
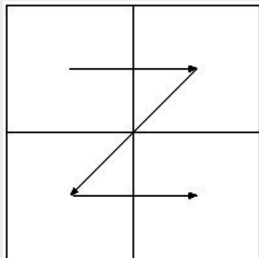
Recursive vs. iterative implementations

```
int factorial ( int n )           /* Recursive implementation */
{
    if (n < 0) return (-1);       /* Error condition */
    if (n == 0) return (1);      /* Base case */
    return(n * factorial(n-1));  /* Recursive call */
}
```

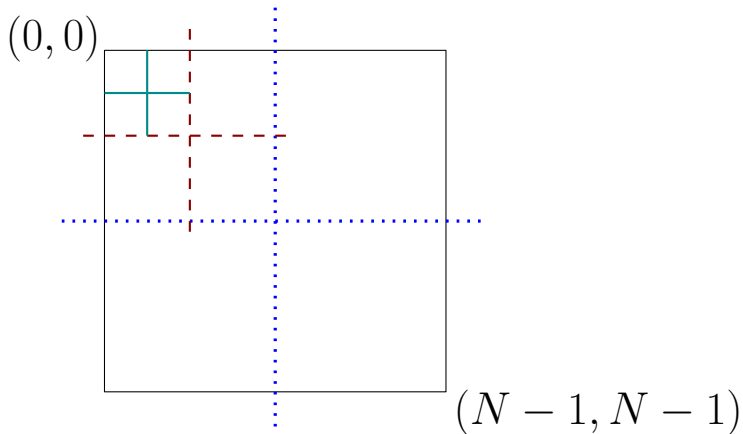
Z curve

Problem statement

Consider a 2-D matrix of size $2^m \times 2^m$. The entries of the matrix are, in row-major order, $1, 2, 3, \dots, 2^{2m}$. Print the entries of the matrix in Z-curve order (as shown in the picture below).



Z curve: structure



Z curve: code I

```
void z_curve(int top_left_row, int top_left_column,
             int bottom_right_row, int bottom_right_column,
             int **matrix)
{
    /* Base case */
    if (top_left_row == bottom_right_row &&
        top_left_column == bottom_right_column) {
        printf("%d ", matrix[top_left_row][top_left_column]);
        return;
    }

    /* Recurse */
    /* upper-left sub-square */
    z_curve(top_left_row,
            top_left_column,
            (top_left_row + bottom_right_row)/2,
            (top_left_column + bottom_right_column)/2,
            matrix);
}
```

Z curve: code II

```
/* upper-right sub-square */
z_curve(top_left_row,
        (top_left_column + bottom_right_column)/2 + 1,
        (top_left_row + bottom_right_row)/2,
        bottom_right_column,
        matrix);

/* lower-left sub-square */
z_curve((top_left_row + bottom_right_row)/2 + 1,
        top_left_column,
        bottom_right_row,
        (top_left_column + bottom_right_column)/2,
        matrix);

/* lower-right sub-square */
z_curve((top_left_row + bottom_right_row)/2 + 1,
        (top_left_column + bottom_right_column)/2 + 1,
        bottom_right_row, bottom_right_column,
        matrix);
```

Z curve: code III

```
    return;  
}
```

Permutations

Algorithm

To generate all permutations of $1, 2, 3, \dots, n$, do the following:

1. Generate all permutations of $2, 3, \dots, n$, and add 1 to the beginning.
2. Generate all permutations of $1, 3, 4, \dots, n$ and add 2 to the beginning.
- ...
- n . Generate all permutations of $1, 2, \dots, n - 1$ and add n to the beginning.

Permutations: code

```
void permute(int *A, int k, int n) {
    int i;

    if (k==n) {
        for (i = 0; i < n; i++) {
            printf("%d ", A[i]);
        }
        putchar('\n');
        return;
    }

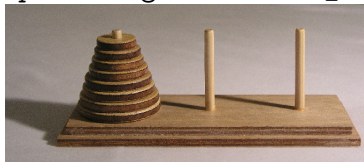
    permute(A, k+1, n);
    for(i = k+1; i < n; i++){
        SWAP(A, i, k);
        permute(A, k+1, n);
        SWAP(A, k, i);
    }

    return;
}
```

Top level call:
permute(A,0,n);

1. Towers of Hanoi: see

https://en.wikipedia.org/wiki/Tower_of_Hanoi



CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=228623>

2. Write a recursive function with prototype `int C(int n, int r)`; to compute the binomial coefficient using the following definition:

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

Supply appropriate boundary conditions.

3. Define a function $G(n)$ as:

$$G(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ G(n-1) + G(n-2) + G(n-3) & \text{if } n \geq 3 \end{cases}$$

Write recursive **and** iterative (i.e., non-recursive) functions to compute $G(n)$.

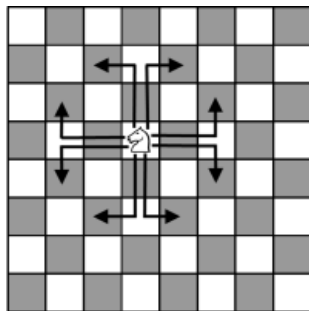
4. What does the following function compute?

```
int f ( int n )
{
    int s = 0;
    while (n-->0) s += 1 + f(n);
    return s;
}
```

5. Complete the skeletons provided for recursive implementations of the *mergesort* and *quicksort* algorithms.

Problems – I

1. In chess, a knight can move to a cell that is horizontally 2 cells and vertically 1 cell away, or vertically 2 cells and horizontally 1 cell away (see Fig. 1). Thus, any knight on the board can attempt exactly eight moves. Some of these moves will result in the knight moving off the board.



Problems – II

Given an $m \times n$ chess-like board, and a knight at position (i, j) (where $0 \leq i < m$ and $0 \leq j < n$), compute the probability that the knight remains on the board after a given number (say k) of steps, assuming that, for a knight on the board, all eight moves are equally likely, but once the knight moves off the board, it has *no valid* moves.

Input Format

$m\ n\ i\ j\ k$

Output Format

The probability value rounded up to 6 decimal places.

Sample Input 0

1 2 0 1 1

Sample Output 0

0.000000

Sample Input 1

8 8 3 3 1

Sample Output 1

1.000000

Sample Input 2

6 6 0 0 1

Sample Output 2

0.250000

Sample Input 3

4 4 0 0 2

Sample Output 3

0.125000

Explanation for the above

	0	1	2	3
0	0/2/2		2	
1			1	2
2	2	1		
3		2		2/2

The knight starts at the top-left corner (0,0). After 1 move, it may either move off the board (permanently), or go to the squares marked 1 (in red / blue). After another move from the square marked with a red (respectively, blue) 1, it lands on one of the squares marked with a red (respectively, blue) 2, or moves off the board. The knight stays on the board for 8 distinct sequences of moves (of 2 steps each). The total number of such moves of 2 steps each is $8 \times 8 = 64$.

The following simplified version of the above problem may help understand how the probability is to be calculated.

Consider a 2×1 grid, with an **X** on the left square. Now toss a fair coin: if it comes up heads, move **X** one square to the right; if it comes up tails, move **X** one square to the left. What is the probability that **X** stays within the grid after k coin tosses?

Solution: The only sequence that permits **X** to stay on the board is HTHTHT... Thus, the required probability is $\frac{1}{2^k}$.

If it helps your understanding, you could calculate the probability of **X** moving off the board. This corresponds to the following mutually exclusive sequences:

T, **HH**, **HTT**, **HTHH**, **HTHTT**, **HTHTHH**, ...

The prob. of getting one of these sequences is $\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots + \frac{1}{2^k} = 1 - \frac{1}{2^k}$.

Note that, for this problem, the sequences **T** and **TY₂Y₃...Y_k** (where Y_i is either H or T) *have the same probability*. Thus, subsequent coin tosses do not affect the probability of the event that moves **X** off the grid.

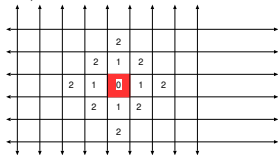
2. Write a program that takes a positive integer $n \leq 5$, and a non-negative integer $k \leq 100$ as command-line arguments, and computes the total number of cells reachable from any starting cell within an infinite, n -dimensional grid in k steps or less. See the examples below. You are only permitted to travel in a direction that is parallel to one of the grid lines; diagonal movement is not permitted.

NOTE: If you are interested, you may compute a closed-form expression for the required number, but **DO NOT** use the closed-form expression to compute the answer to this problem.

Problems – VII



$n = 1, k = 2$; number of cells reachable = 5



$n = 2, k = 2$; number of cells reachable = 13

For $n = 3$, your output for $k = 0, 1$ and 2 should be 1, 7, and 25, respectively.

3. <http://cse.iitkgp.ac.in/~abhij/course/lab/PDS/Spring15/A4.pdf>