Functions and recursion

Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2019/index.html

Function calls

```
1   void main(void)
2   {  ...
3      u = f(x, y*z);
4      ...
5   }
6
7   int f(int a, int b)
8   {  ...
9      if (a > 0)
10        p = g(b);
11     else
12        p = h(b / 2);
13     return p;
14  }
15
16  int g(int m)
17  { ... }
18
19  int h(int n)
20  { ... }
```

Function calls

```
1   void main(void)
2   {  ...
3      u = f(x, y*z);
4      ...
5   }
6
7   int f(int a, int b)
8   {  ...
9      if (a > 0)
10        p = g(b);
11     else
12        p = h(b / 2);
13     return p;
14  }
15
16  int g(int m)
17  { ... }
18
19  int h(int n)
20  { ... }
```

1. Let `a = x`, `b = y*z`.
2. Execute the statements in `f()`.
   (a) If `a` is positive, let `m = b`. Execute the statements in `g()`, and store the obtained value in `p`.
   (b) Otherwise, let `n = b/2`. Execute the statements in `h()`, and store the obtained value in `p`.
   (c) In either case, return value of `p` to calling function.
3. Store the value returned by `f` in `u`.
4. Continue from line 4.

Function calls

```
1   void main(void)
2   {  ...
3      u = f(x, y*z);
4      ...
5   }
6
7   int f(int a, int b)
8   {  ...
9      if (a > 0)
10        p = g(b);
11     else
12        p = h(b / 2);
13     return p;
14  }
15
16  int g(int m)
17  { ... }
18
19  int h(int n)
20  { ... }
```

- **Caller:** calling function (e.g., `main`)
- **Callee:** called function (e.g., `f`)
- **Formal parameters** (or simply *parameters*)

  Example: `a`, `b` – formal parameters for `f`
- **Actual parameters** (or actuals / arguments)

  Example: `x`, `y*z` – arguments passed to `f` on line 3

Parameter passing

- Call by value
- Call by reference
- Call by name

Call by value

- Arguments evaluated, copied into local storage area of called function
- Changes made to parameters in called function **not** reflected in caller
- C — call by value,

Call by value

- Arguments evaluated, copied into local storage area of called function
- Changes made to parameters in called function **not** reflected in caller
- C — call by value, but arrays are interpreted as pointers

Call by value

- Arguments evaluated, copied into local storage area of called function
- Changes made to parameters in called function **not** reflected in caller
- C — call by value, but arrays are interpreted as pointers

Example:

```
void swap(int x, int y) { int t; t = x; x = y; y = t; }

int main(void) {
  int a = 10, b = 20;
  printf("a = %d, b = %d\n", a, b);
  swap(a,b);
  printf("a = %d, b = %d\n", a, b);
  return;
}
```

Call by value

- Arguments evaluated, copied into local storage area of called function
- Changes made to parameters in called function **not** reflected in caller
- C — call by value, but arrays are interpreted as pointers

Example:

```
void swap_elements(int *A, int i, int j)
{ int t; t = A[i]; A[i] = A[j]; A[j] = t; }

int main(void) {
  int a[] = {10, 20, 30, 40};
  printf("1st element = %d, 2nd element = %d\n", a[0], a[1]);
  swap_elements(a, 0, 1);
  printf("1st element = %d, 2nd element = %d\n", a[0], a[1]);
  return;
}
```

Other parameter passing mechanisms

- Call by reference: changes made to argument variables in callee are reflected in caller
- Call by name: parameters are *literally* replaced in body of callee by arguments (like string replacement)
  - C macros use call by name

Passing parameters using pointers in C

- For arrays
- To “simulate” call by reference
- For efficiency

Passing parameters using pointers in C

- For arrays
- To "simulate" call by reference
- For efficiency

Example:

```
void swap(int *x, int *y) { int t; t = *x; *x = *y; *y = t; }

int main(void) {
  int a = 10, b = 20;
  printf("a = %d, b = %d\n", a, b);
  swap(&a, &b);
  printf("a = %d, b = %d\n", a, b);
  return;
}
```

Passing parameters using pointers in C

- For arrays
- To "simulate" call by reference
- For efficiency

Example:

```
void swap(int *x, int *y) { int t; t = *x; *x = *y; *y = t; }

int main(void) {
  int a = 10, b = 20;
  printf("a = %d, b = %d\n", a, b);
  swap(&a, &b);
  printf("a = %d, b = %d\n", a, b);
  return;
}
```

`&a` = address of `a`
`x` = copy of address of `a`
Both point to same location
⇒ Changing `*(&a)` and `*x` are equivalent.
⇒ Changing `a` and `*x` are equivalent.

Recursion

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

```
1   int factorial ( int n ) {
2      int prod;
3
4      if (n < 0) return (-1);
5      prod = 1;
6      while (n > 0) {
7         prod *= n;
8         n = n - 1;
9      }
10     return (prod);
11  }
```

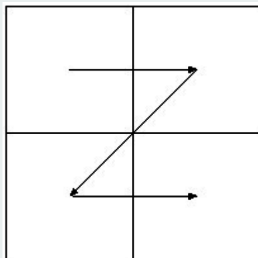
Recursive vs. iterative implementations

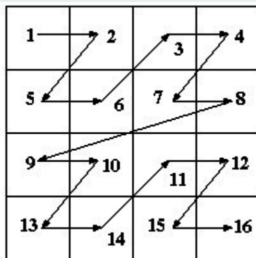
```
1    int factorial ( int n )          /* Recursive implementation */
2    {
3       if (n < 0) return (-1);       /* Error condition */
4       if (n == 0) return (1);       /* Base case */
5       return(n * factorial(n-1));   /* Recursive call */
6    }
```

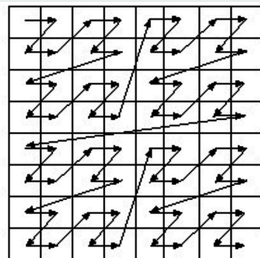
Z curve

Problem statement

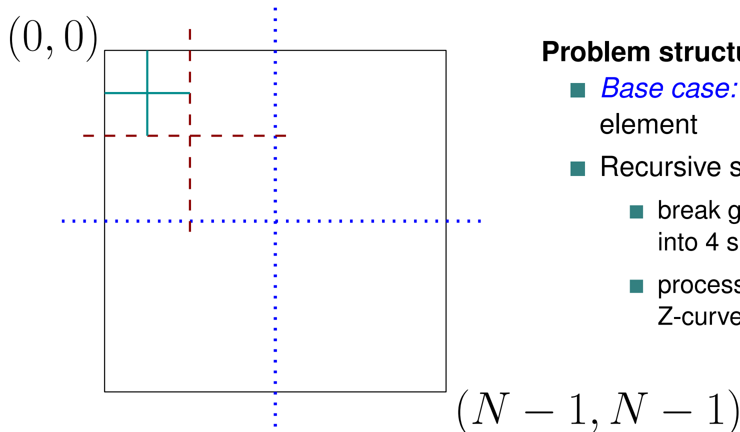
Consider a 2-D matrix of size $2^m \times 2^m$. The entries of the matrix are, in row-major order, $1, 2, 3, \ldots, 2^{2m}$. Print the entries of the matrix in Z-curve order (as shown in the picture below).







Z curve: structure



Problem structure:

- *Base case:* single element
- Recursive structure:
  - break given square into 4 sub-squares
  - process squares in Z-curve order

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);
    return;
}
```

Permutations

Algorithm

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

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

   $\ldots$

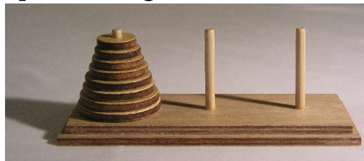
n. Generate all permutations of $1, 2, \ldots, n-1$ and add n to the beginning.

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;
    }

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

    return;
}
```

Problems – I

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.

Problems – II

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 $G(n)$.

Problems – III

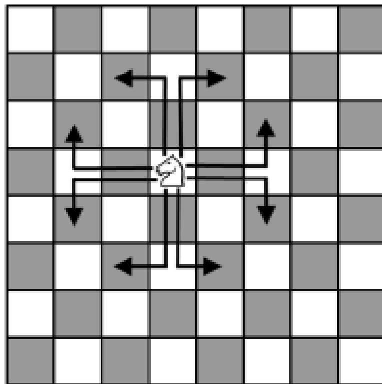
4. What does the following function compute?

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

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

Problems – IV

6. 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. 6). Thus, eight moves are possible for a knight in general.



Problems – V

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, at each step, all eight moves are equally likely. Note that, once the knight moves off the board, it can never return to the board.

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

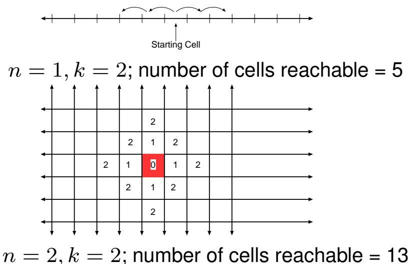
Sample Output 3
0.125000
Explanation for the above

	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 $8 \times 8 = 64$.

Problems – VII

7. 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.



$n = 1, k = 2$; 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.