Computing Laboratory

Functions and Recursion

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

November, 2021

1 Functions

2 Parameter Passing

3 Recursion

Functions – Prototype declaration, definition, call

```
int max(int, int, int); // Prototype declaration

int max(int a, int b, int c){ // Definition starts
    return (a>b)?((a>c)?a:c):((b>c)?b:c);
} // Definition ends

int main(){
    int x = 7, y = 13, z = 11, maximum;
    maximum = max(x, y, z); // Call
    return 0;
}
```

Functions – Prototype declaration, definition, call

```
int max(int, int, int); // Prototype declaration

int max(int a, int b, int c){ // Definition starts
    return (a>b)?((a>c)?a:c):((b>c)?b:c);
} // Definition ends

int main(){
    int x = 7, y = 13, z = 11, maximum;
    maximum = max(x, y, z); // Call
    return 0;
}
```

- Calling function: `main()`
- Called function: `max()`
- Parameters: a, b, c (declared in `max()`)
- Arguments: x, y, z (passed to `max()`)

Functions – Efficient use

```
int max(int a, int b){
    if(a > b) {return a;}
    else {return b;}
}
```

Functions – Efficient use

```
int max(int a, int b){
    if(a > b) {return a;}
    else {return b;}
}

int max(int a, int b, int c){
    return max(max(a, b), c);
}
```

Inline functions

```
int main(){
    int x = 7, y = 13, maximum;
    inline int max(int x, int y) {return x > y ? x : y;}
    maximum = max(2, 3);
    return 0;
}
```

Inline functions

```
int main(){
    int x = 7, y = 13, maximum;
    inline int max(int x, int y) {return x > y ? x : y;}
    maximum = max(2, 3);
    return 0;
}
```

Note: Inline functions are compiled just as a regular code, but can be then inserted into compiled code and optimized as needed.

Different mechanisms of parameter passing in C

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

Call by value

- Arguments are evaluated and then copied into local storage area of the `called` function.
- Changes made to parameters in the called function do not get reflected in the `calling` function.

Call by value

- Arguments are evaluated and then copied into local storage area of the called function.
- Changes made to parameters in the called function do not get reflected in the calling function.

```
void SWAP(int a, int b){
    int t; t = a; a = b; b = t;
}
int main(){
    int m = 5, n = 20;
    printf("(%d, %d)", 2*m, n); // Prints (10, 20)
    SWAP(m, n);
    printf("(%d, %d)", 2*m, n); // Prints (10, 20)
    return 0;
}
```

Call by value

- C uses call by value in general, but arrays are interpreted as pointers.

Call by value

- C uses call by value in general, but arrays are interpreted as pointers.

```
void SWAP_FirstPair(int *A, int i, int j){
    int t; t = A[i]; A[i] = A[j]; A[j] = t;
}
int main() {
    int m[] = {10, 20, 30, 40};
    printf("(%d, %d)", m[0], m[1]); // Prints (10, 20)
    SWAP_FirstPair(m, 0, 1);
    printf("(%d, %d)", m[0], m[1]); // Prints (20, 10)
    return 0;
}
```

Call by reference

- Changes made to parameters in the `called` function get reflected in the `calling` function.
- C simulates call by reference for efficiency.

Call by reference

- Changes made to parameters in the called function get reflected in the calling function.
- C simulates call by reference for efficiency.

```
void SWAP(int *a, int *b){
    int t; t = *a; *a = *b; *b = t;
}
int main(){
    int m = 10, n = 20;
    printf("(%d, %d)", m, n); // Prints (10, 20)
    SWAP(&m, &n);
    printf("(%d, %d)", m, n); // Prints (20, 10)
    return 0;
}
```

Note: `&m` and a point to the same location. Therefore, changing `*(&m)` (= `m`) and `*a` are equivalent.

Call by name

- Parameters are literally replaced in body of the `calling` function by arguments (like string replacement)

Call by name

- Parameters are literally replaced in body of the calling function by arguments (like string replacement)

```
#define SWAP(a, b) { int t; t = a; a = b; b = t;}
int main(){
    int m = 10, n = 20;
    printf("(%d, %d)", m, n); // Prints (10, 20)
    SWAP(m, n);
    printf("(%d, %d)", m, n); // Prints (20, 10)
    return 0;
}
```

Basics of 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 versus iterative implementations

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

Recursive versus iterative implementations

Let's execute the following complete program:

```
#include <stdio.h>
int factorial(int);
void main(){
    factorial(5);
}
int factorial(int n){                    // Iterative version
    int prod = 1;
    if(n < 0) return (-1);               // Error condition
    while(n > 0){
        printf("\nValue of n (@address %u) = %d", &n, n);
        prod *= n--;
    }
    return (prod);
}
```

Recursive versus iterative implementations

Output:

```
Value of n (@address 1257313068) = 5
Value of n (@address 1257313068) = 4
Value of n (@address 1257313068) = 3
Value of n (@address 1257313068) = 2
Value of n (@address 1257313068) = 1
```

Recursive versus iterative implementations

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

Recursive versus iterative implementations

Let's execute the following complete program:

```
#include <stdio.h>
int factorial(int);
void main(){
    factorial(5);
}
int factorial(int n){                    // Recursive version
    if(n < 0) return (-1);               // Error condition
    printf("\nValue of n (@address %u) = %d", &n, n);
    if(n == 0) return (1);               // Base case
    printf("\nEntering into factorial(%d)", n-1);
    return(n * factorial(n-1));          // Recursive call
}
```

Recursive versus iterative implementations

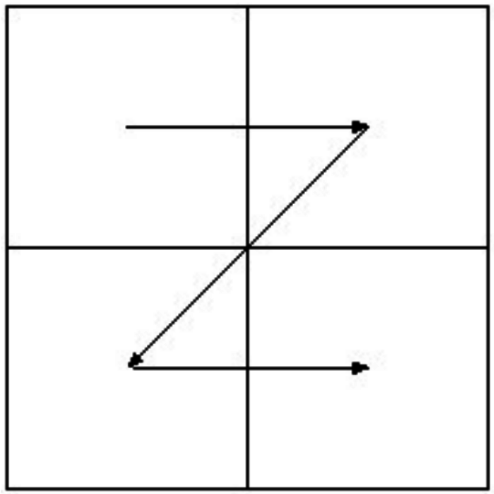
Output:

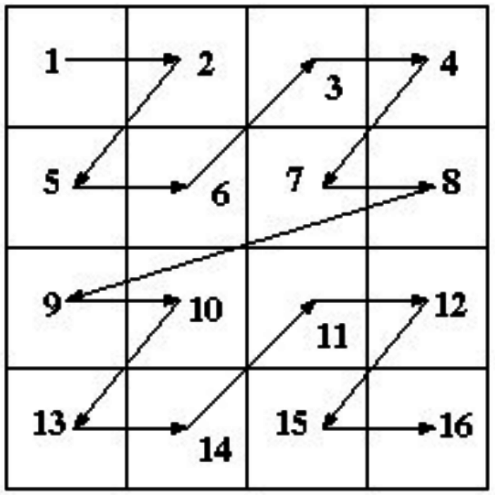
```
Value of n (@address 540067532) = 5
Entering into factorial(4)
Value of n (@address 540067500) = 4
Entering into factorial(3)
Value of n (@address 540067468) = 3
Entering into factorial(2)
Value of n (@address 540067436) = 2
Entering into factorial(1)
Value of n (@address 540067404) = 1
Entering into factorial(0)
Value of n (@address 540067372) = 0
```

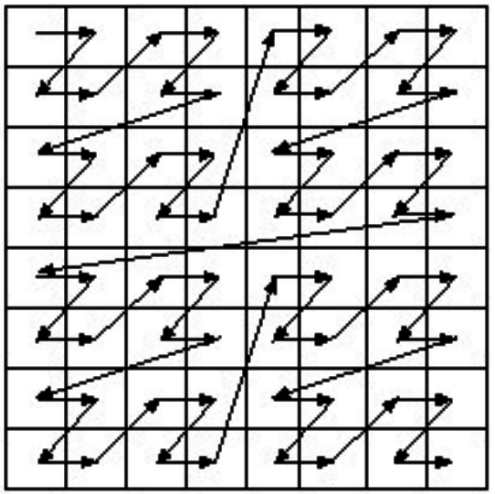
Z-curve

Problem statement

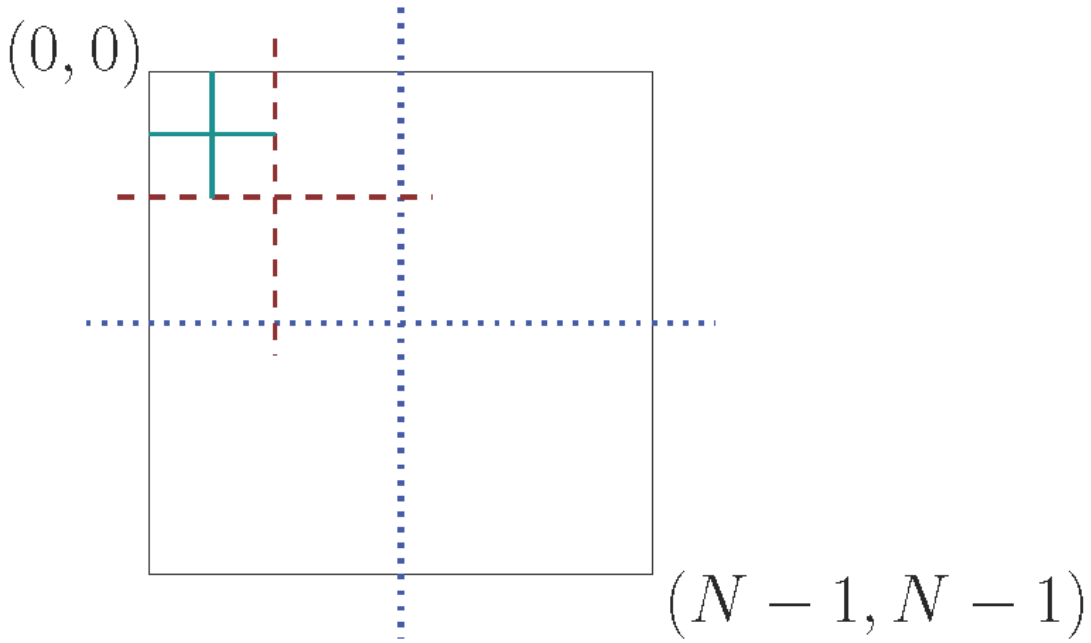
Consider a 2-D matrix of size $2^m \times 2^m$. The entries of the matrix are, in row-major order, $1, 2, 3, \ldots, 2^{2m}$. Print the entries of the matrix in Z-curve order (as shown in the picture below).







Structure of Z-curve



Decomposing the problem:

- *Base case:* A single element
- Recursive structure:
  - Break the given square into 4 sub-squares
  - Process the sub-squares following a Z-curve pattern

Z-curve – Writing the code

```
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;
    }

    /* Recursive structure */

    /* 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 – Writing the code

```
    /* 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.

   ...

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

Permutations – Writing the 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 – Day 8

1. What does the following function compute?

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

2. Write a recursive function with prototype

```
int C(int n, int r);
```

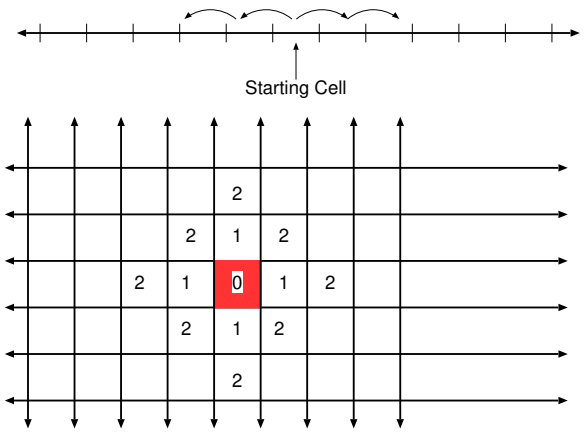
to compute the binomial coefficient using the following definition:

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

Supply appropriate boundary conditions.

Problems – Day 8

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



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

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

Problems – Day 8

4. 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 $G(n)$.

5. Solve the Towers of Hanoi problem. See further details at: `https://en.wikipedia.org/wiki/Tower_of_Hanoi`