Recursion

Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2017/index.html

Definition

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  }
```

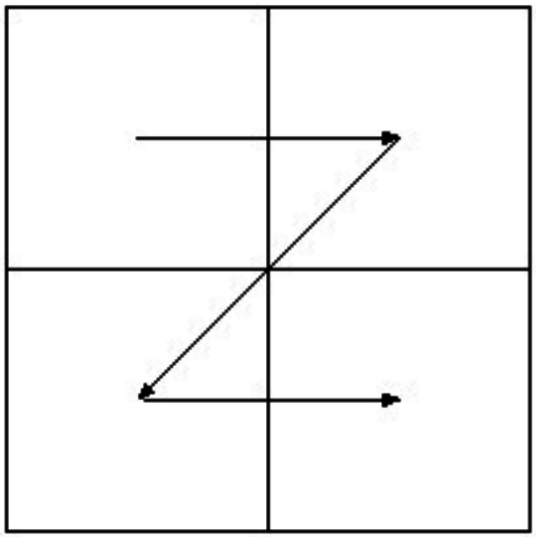
Recursive vs. iterative implementations

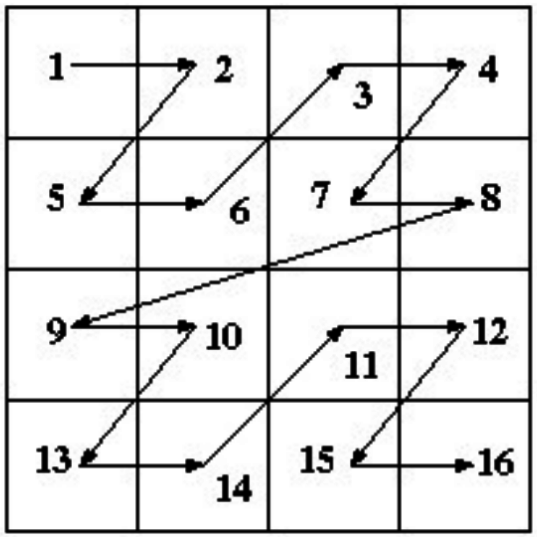
```
1    int factorial ( int n ) /* Recursive implementation */
2    {
3       if (n < 0) return (-1); /* Error condition */
4       if (n == 0) return (1); /* Base case */
5       return(n * factorial(n-1)); /* Recursive call */
6    }
```

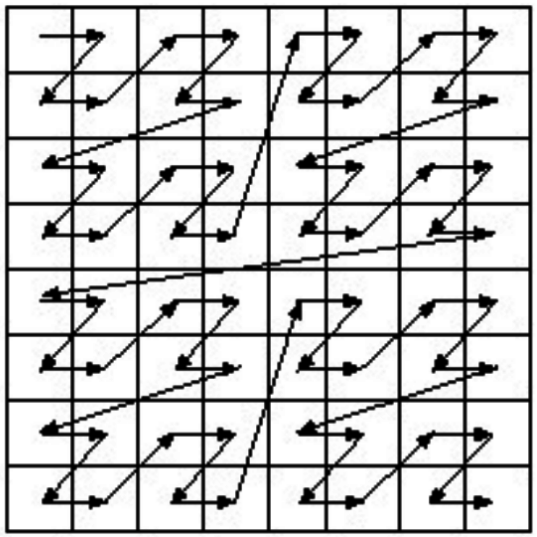
Z curve

Problem statement

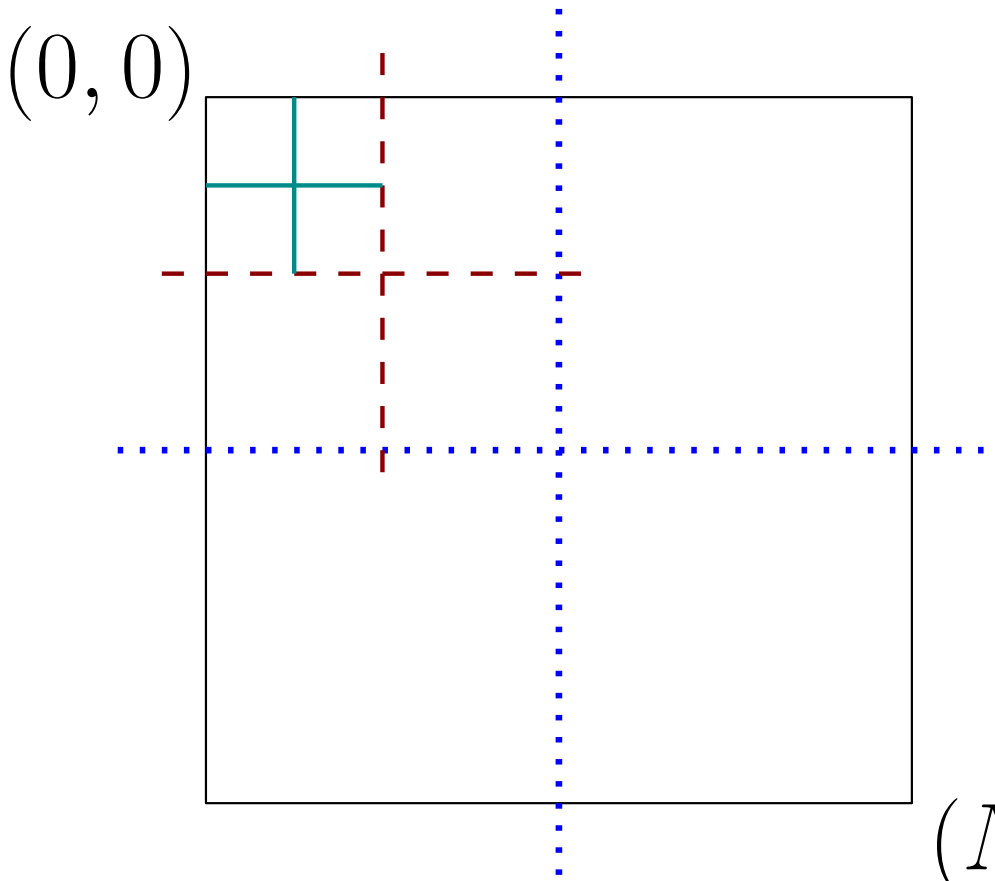
Consider a 2-D matrix of size $2^m \times 2^m$. The entries of the matrix are, in row-major order, $1, 2, 3, \ldots, 2^{2m}$. Print the entries of the matrix in Z-curve order (as shown in the picture below).







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;
}
```

Binary trees

(Recursive) Definition

A *binary tree* over a domain D is either:

- the empty set (called an *empty binary tree*); or
- a 3-tuple $\langle S_1, S_2, S_3 \rangle$ where
  - $S_1 \in D$, (called the *root*) and
  - S_2 and S_3 are *binary trees* over D (called the *left* and *right* subtree resp.)

Binary tree properties

- *Height:* h
- *Number of nodes:* n
- *Number of leaves:* l

Binary tree traversals

- *Preorder*
- *Inorder*
- *Postorder*

Binary tree implementation

Conventional implementation:

```
typedef struct tnode {
    DATA d;
    struct tnode *left, *right;
    struct tnode *parent; // optional
} TNODE;
```

- One malloc per node
- Nodes may be scattered all over the heap area

Binary tree implementation

Alternative implementation:

```
typedef struct tnode {
    DATA d;
    int left, right;
    int parent; //optional
} TNODE;
```

- One initial malloc and reallocs as needed
- All nodes located within the same array

Binary tree implementation

Alternative implementation:

```
typedef struct tnode {
    DATA d;
    int left, right;
    int parent; //optional
} TNODE;
```

- One initial malloc and reallocs as needed
- All nodes located within the same array

Initially:

root = -1

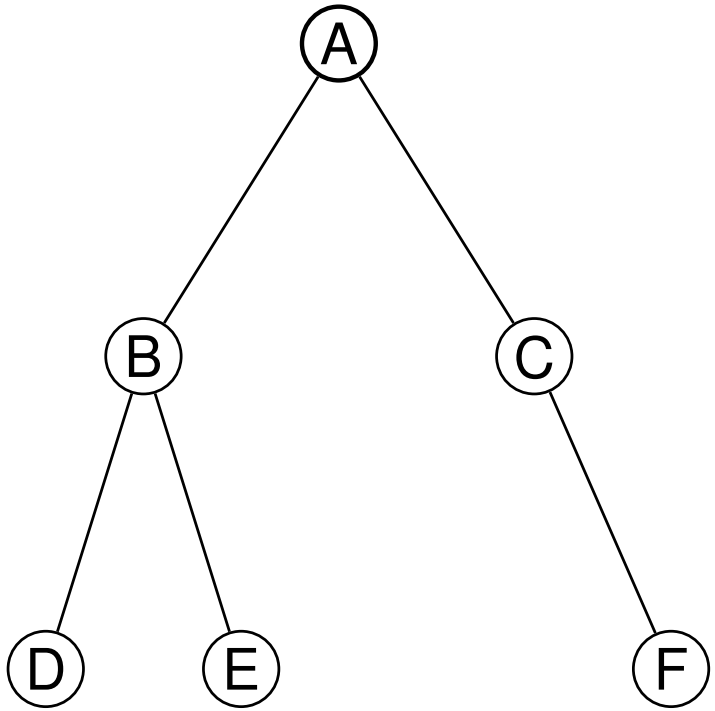
	DATA	left	right
free ⟶ 0	—	1	-1
1	—	2	-1
2	—	3	-1
3	—	4	-1
⋮		⋮	
n-1	—	-1	-1

Binary tree implementation

Alternative implementation:

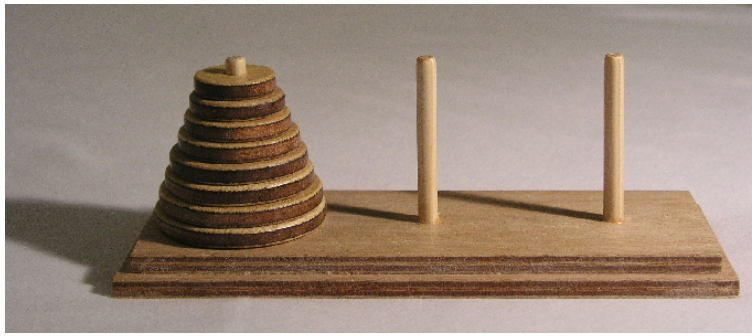
root = 0

	DATA	left	right
0	A	1	2
1	B	3	4
2	C	-1	5
3	D	-1	-1
4	E	-1	-1
5	F	-1	-1
free ⟶ 6	—	7	-1
⋮	⋮		
n-1	—	-1	-1



Problems – Day – 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 – Day – 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 – Day – 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 – Day – IV

For the following problems, some test cases (example trees) are available from the course home page. You may adapt the following code snippet to read in a tree and store it in an array (using the Alternative Implementation).

```
scanf("%u", &numNodes);
if (NULL == (tree = (NODE *) malloc(numNodes * sizeof(NODE)))) {
    fprintf(stderr, "Out of memory\n");
    exit(1);
}
for (node = tree, i = 0; i < numNodes; node++, i++)
    scanf("%d %d %d", &(node->data), &(node->left), &(node->right));
```

6. Given a binary tree with integer-valued nodes, and a *target* value, determine whether there exists a root-to-leaf path in the tree such that the sum of all node values along that path equals the target.
   Modify your program to consider *all* paths, not just root-to-leaf paths.

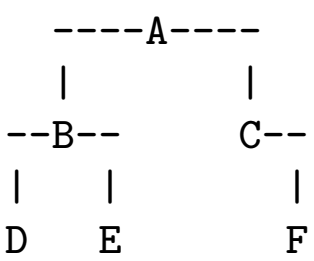
Problems – Day – V

7. Given a binary tree stored in an array (as in the Alternative Implementation), and the indices of two nodes in the tree, find the index of the node that is the lowest common ancestor of the given nodes.
8. Write a program to print a binary tree, rotated anti-clockwise by 90° on the screen. For example, for the tree on slide 16, your output should like something like:

```
        F
    C
A
        E
    B
        D
```

Now, try to write a program that will print the same tree in the following format:

Problems – Day – VI



Appendix: storage issues, activation records

How does the computer handle function calls?

```
1   void main(void)
2   { ...
3      m = 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  { ... }
```

How does the computer handle function calls?

```
1   void main(void)
2   { ...
3      m = 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()`.
   (b) Otherwise, let `n = b/2`.
   Execute the statements in `h()`.
   (c) In either case, return the value stored in `p` (obtained from `g()` or `h()`).
3. Continue from line 4.

Storage for local variables

- Local variables stored in a region of memory called an *activation record*
- Need to keep track of *nested* calls

Storage for local variables

- Local variables stored in a region of memory called an *activation record*
- Need to keep track of *nested* calls
- Where are the activation records stored?

Storage for local variables

- Local variables stored in a region of memory called an *activation record*
- Need to keep track of *nested* calls
- Where are the activation records stored?
  - simple solution: one block of memory per function
    does not work for recursive functions

Storage for local variables

- Local variables stored in a region of memory called an *activation record*
- Need to keep track of *nested* calls
- Where are the activation records stored?
  - simple solution: one block of memory per function
    does not work for recursive functions
  - better solution: one block of memory per *function call*
    - activation records stored in a chunk of memory called *activation stack*
    - when a function is called, its activation record is added to the end of the activation stack;
    - when function returns, its activation record is removed.
    - works for recursive functions

Review question – slide I

Which of `read_data1`, `read_data2`, `read_data3` is best?

```
1  typedef struct {
2      char name[64];
3      int roll, rank;
4      float percent;
5  } STUDENT;
6
7  STUDENT *read_data1(void)
8  { STUDENT s;
9    scanf("%s %d %d %f",
10     &(s.name[0]), &(s.roll),
11     &(s.rank), &(s.percent));
12   return &s;
13 }
```

Review question – slide II

```
14
15  STUDENT read_data2(void)
16  { STUDENT s;
17    scanf("%s %d %d %f",
18      &(s.name[0]), &(s.roll),
19      &(s.rank), &(s.percent));
20    return s;
21  }
22
23  STUDENT *read_data3(STUDENT *s)
24  { scanf("%s %d %d %f",
25      &(s->name[0]), &(s->roll),
26      &(s->rank), &(s->percent));
27    return s;
28  }
```