Computer Programming
Laboratory 11

Recursion

GENERAL STRUCTURE

```
int f(n1) {
  if (is_base(n1) == true) {
    return value;
  }
  else {
     f(g(n1));
  }
}
```

EXAMPLE (FEASIBLE INDUCTION)

INDUCTION: (f(n) and f(n) => f(n+1)) => forall x. f(x)

```
void reverse_induction(int n1) {
  if (n1 == 0) {
    cout << “done!”;
  }
  else {
     cout << “Working on: “ << n1 << endl;
     f(n1 - 1);
  }
}
```

Recursion

```
void reverse_induction(int n, bool true_for_others) {
  if (n1 == 0) {
    cout << “done!”;
    return check_prop_for(0) && true_for_others
 }
  else {
     cout << “Working on: “ << n << endl;
     true_for_n = check_prop_for(n)
     reverse_induction(n1 - 1, true_for_n && true_for_others);
  }
}
```

Do and donts

- Find the pattern:
  - What is the base case (when do I stop?)
  - How do I move from “current” to “next”, so that “next” has the same structure of “current” or it is the base case
- Use a wrapper function, if you need to add arguments

- Don’t do side effects
- Don’t use global variables
- Don’t nest conditions

Level 0: no brainers

```
recursion_is_difficult = true
while(recursion_is_difficult) {
  Implement problems on the right
  If (ah_ah_moment) {
    recursion_is_difficult = false
  }
}
```

- Factorial
- Product of first N integers
- Sum of first N integers
- Division as consecutive subtractions
- Sum of x+y as adding 1 to x for y times

Remember the pattern

```
void recursive(data) {
   if (base_case)
      Return value_of_base_case

   data1 ← set to something closer to base case
   recursive(data1)
}
```

Level 1: Arrays and Matrices

```
arrays_are_difficult = true
while(arrays_are_difficult) {
  Implement problems on the right
  If (ah_ah_moment) {
    arrays_are_difficult = false
  }
}
```

- print a vector in reverse order
- sum the elements of a vector
- product of two vectors
- Check if a vector is palindrome
- Let the user enter a matrix (array of arrays) and print values
- matricial product

Level 2: Arrays and Recursion

```
if (recursion_is_difficult || arrays_are_difficult) {
  goto slide 1; // remark goto is a bad practice
}
implement a binary tree using an array
draw a binary tree
implement it
implement the in_order_visit to a given tree
If (ah_ah_moment) {
   recursion_is_difficult = false
}
```

- A node is stored as four consecutive cells in the array:
  - Allocated = 0 || 1
  - Value of the node
  - Index in the array where I find the left child (-1 if no left child)
  - Index in the array where I find the right child (-1 if no right child)
- We implement the functions in the next slide

implement a binary tree using an array

```
// find index of first free four cells in the array
int free(int memory[])
// add a node to the tree
int add_node(int value, int left_child, int right_child, int memory[])
// return true if the four cells in the array starting at index_of_node are marked as allocated
bool is_allocated(int index_of_node)
// getters
int get_node_value(int index_of_node)
// returns the index in the array where I can find the left (and right) child
int get_left_child(int index_of_node)
int get_right_child(int index_of_node)
// return true if node is a leaf
bool is_leaf(int index_of_node)
```

Implement a binary tree using an array

- Use the API above to build a tree, for instance:

```
int memory[4 * 1000]; // space for 1000 nodes.
int leaf1 = add_node(10, 0, 0, memory)
int leaf2 = add_node(30, 0, 0, memory)
int leaf3 = add_node(30, leaf1, leaf2, memory)
int leaf4 = add_node(40, leaf3, 0, memory)
```

… and back to recursion

```
in_order_visit(int node) {
   if (is_leaf(node))
     return;
   else {
     in_order_visit(left_child)
     cout << node value is: get_value(node)
     in_order_visit(right_child)
  }
}
```

Tic Tac Toe: Specification

```
// output the board to screen
void print_board(const int board[][BOARD_SIZE])
// given [row, col] (move), move a piece of color to board (return false if already taken)
bool move(int move[2], int color, int board[][BOARD_SIZE])
// has color won?
void has_won(int board[][BOARD_SIZE], color)
// given a board, return the list of cells which are free; store them in cells,
// found stores the number of free cells found
void free_cells(int cells[BOARD_SIZE * BOARD_SIZE][2], int &found, const inst board[][BOARD_SIZE])
// choose an element at random from free_cells and store it into cell (that is, [row, col]).
void computer_move(int cell[2], const int free_cells_size, const inst free_cells[BOARD_SIZE * BOARD_SIZE][2])
// ask row, cell, return them in cell
void user_move(int cell[2])
```

Improving Tic Tac Toe

- Let the computer play by itself
- Increase the board size