Basics of programming

Balázs Goldschmidt

Recursive functions

Recursive functions

- Problem
  - Let's calculate the n^{th} factorial!

```
int fact(int x) {
   int i, y;
   y = 1;
   for (i = 2; i <= x; i++) {
       y *= x;
   }
   return y;
}
```

Recursive functions

- Problem
  - Let's calculate the n^{th} factorial!
  - Without loops: n! = n*(n-1)!

```
int fact(int x) {
   if (x == 0) return 1;
   else return x*fact(x-1);
}
```


Recursive functions

- Sometimes recursion is simpler
- Problem
  - calculate n^{th} fibonacci number!
  - definition: $a_1 = 1$, $a_2 = 1$, $a_n = a_{n-1}+a_{n-2}$

```
int fibo(int x) {
   if (x==1 || x==2) return 1;
   else return fibo(x-1)+fibo(x-2);
}
```


Recursive functions

- Try fibonacci with loops!
  - definition: $a_1 = 1$, $a_2 = 1$, $a_n = a_{n-1}+a_{n-2}$

```
int fibo(int x) {
   int i, a1, a2, a3;
   a1 = a2 = a3 = 1;
   for (i = 3; i <= x; i++) {
       a2 = a3;
       a1 = a2;
       a3 = a1+a2;
   }
   return a3;
}
```

Towers of hanoi

- Problem:
  - we have three rods and n disks
  - move the disks on first rod to the second
  - only 1 piece a time and always put smaller on bigger

Towers of hanoi solution

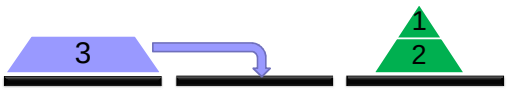
- Print out each move

```
void hanoi(int n, int from, int to, int help) {
   if (n == 0) return;
   hanoi(n-1, from, help, to);
   printf("disk %d, %d -> %d\n", n, from, to);
   hanoi(n-1, help, to, from);
}
```

Towers of hanoi solution

- Print out each move

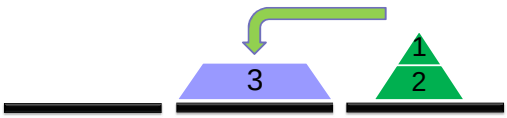
```
void hanoi(int n, int from, int to, int help) {
   if (n == 0) return;
   hanoi(n-1, from, help, to);
   printf("disk %d, %d -> %d\n", n, from, to);
   hanoi(n-1, help, to, from);
}
```



Towers of hanoi solution

- Print out each move

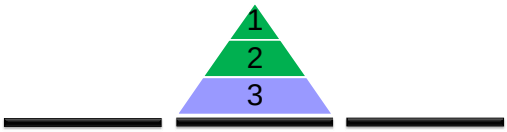
```
void hanoi(int n, int from, int to, int help) {
   if (n == 0) return;
   hanoi(n-1, from, help, to);
   printf("disk %d, %d -> %d\n", n, from, to);
   hanoi(n-1, help, to, from);
}
```



Towers of hanoi solution

- Print out each move

```
void hanoi(int n, int from, int to, int help) {
   if (n == 0) return;
   hanoi(n-1, from, help, to);
   printf("disk %d, %d -> %d\n", n, from, to);
   hanoi(n-1, help, to, from);
}
```



Rules of recursion

- Only one step is to be solved
  - e.g.: moving a single disk
- Rest is done by a new call
  - e.g.: moving n-1 disks
- Always have an exit
  - e.g.: 0 disks needs no move
- Local variables are unique in each call
- Too deep recursion should be avoided

Exercise

- Solve factorial problem
- Print out in each function
  - the depth of recursion
  - the address of parameter x
  - use %p in *printf*: `printf("%p\n", &x);`

Recursive data structures

Store data in order

- Let's store data in order
  - array
    - with each insertion *n/2* elements should be moved on average
  - binary tree
    - stores elements in nodes of a tree
    - each element is usually at most $log_2 n$ far from root

Binary tree

- Data structure

```
typedef struct btree {
   int n;
   struct btree *left, *right;
} btree;
```

- Rules
  - leaves have null pointers for children
  - for each node x
    - `x->left->n  <  x->n`
    - `x->right->n  >=  x->n`

Binary tree: empty tree and creation

- An empty tree is a NULL pointer

```
btree* tree = NULL;
```

- Create one element tree

```
btree* create(int n) {
   bintree* t;
   t = (btree*)malloc(sizeof(btree));
   t->left = t->right = NULL;
   t->n = n;
   return t;
}
```


Binary tree: insertion

- Algorithm
- Insert number *n* into tree *t*

if t is empty
    n is new tree
otherwise
    if *n* < (*t->n)*, insert into *t->left*
    otherwise insert into *t->right*

Binary tree: insertion

- Insert number *n* into tree *t*

```
btree* insert(btree* t, int n) {
  if (t == NULL) {
      t = create(n);
  } else {
      if (n < t->n)
            n->left =   insert(n->left, n);
      else
            n-> right = insert(n->right, n);
  }
  return t;
}
```


Binary tree: contains

- Contains: return true if *tree* contains *n*

```
int contains(btree* t, int n) {
  if (t == NULL) {
      return 0;
  } else {
      if (t->n == n)
            return 1;
      else if (n < t->n)
            return contains(t->left, n);
      else
            return contains(t->right, n);
  }
}
```


Binary tree traversal

- How can we traverse a binary tree?
  - inorder
    - left subtree, root, right subtree
    - e.g. printing data in increasing order
  - preorder
    - root, left subtree, right subtree
  - postorder
    - left subtree, right subtree, root

Binary tree algorithms

- Most algorithms are recursive
  - empty tree
  - root node
  - left or right subtree
- Algorithms for exercise
  - depth of tree
  - number of nodes, leaves, parents
  - printing out: inorder, preorder, postorder
  - deleting a tree