COL100 Lecture 24

Note Title 29-10-2018

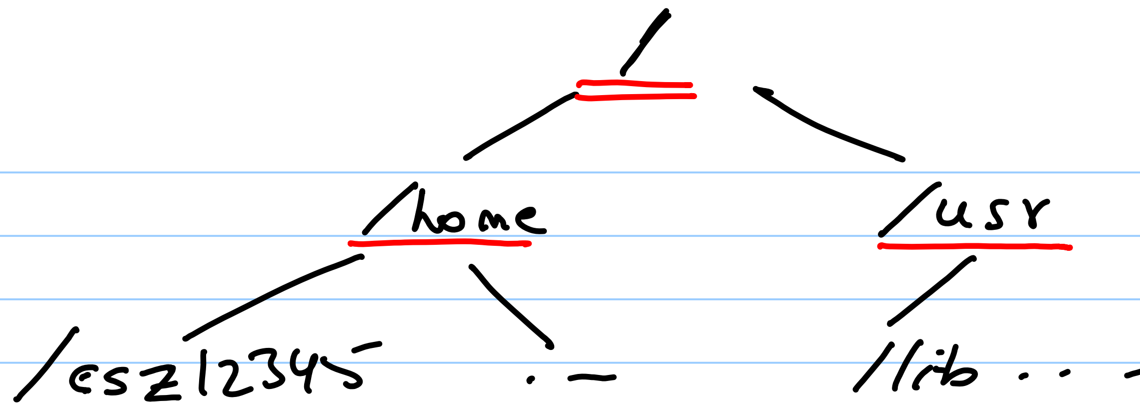
Review: Sets, Maps

Today: RECURSION

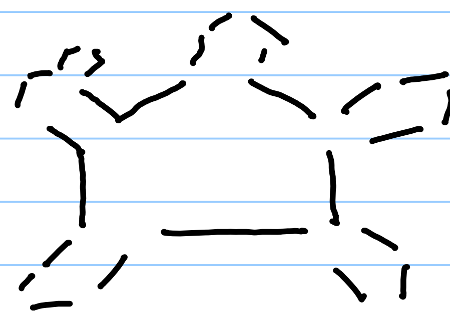
- Exploit <u>self-similarity</u> in problems
- Learn recursive problem solving

```
output f ( .... inputs )
{

    s_outputs = f ( ... smaller inputs )

}
```

How is the problem <u>self-similar</u>?



Fractals



Recursive programming

writing <u>functions that call themselves</u> to solve problems that are recursive (or self-similar) in nature.

Blue-shirt example

Base case (easy case)

Recursive case

- assume that smaller has been solved
- easy to solve the bigger problem with this assumption

Every recursion has at least two cases:

base case:

a simple occurrence of the problem that can be solved directly

recursive case:

a more complex occurrence of the problem that cannot be answered directly.
But it can be instead described in terms of <u>smaller</u> occurrences of the same problem.

Ask yourself :

1. " How is the task <u>self-similar</u> ? "
2. ~~Ab~~ Find the <u>minimum amount</u> of work that can be done to obtain a solution for the problem, given solutions of smaller problems.
3. Make the problem simpler by doing this minimum amount of work.
4. Trust the recursion
5. There must be a stopping point

<u>3 rules of recursion:</u>

1. Every input must have a <u>case</u> (either base or recursive)

2. The <u>must</u> be a base case that makes no recursive calls, i.e. on some inputs, the code should not make any recursive calls.

3. The recursive case must make the problem simpler and make forward progress to the base case.

Recursive program structure

...

```
recursiveFunc (...)
{
    if ( input is base case) {
        compute the solution without
        recursion and return it
    } else { // recursive case
        1. Break the problem into
           subproblems of the same form
        2. Call recursiveFunc() on each
           self-similar problem
        3. Reassemble results of subproblems
    }
```

factorial

$$n! = 1 \times 2 \times 3 \ldots \times (n-1) \times n$$

```
int factorial (int n)
{
    int total = 1;
    for (int i = 1; i <= n; i++)
    {
        total = total * i;
    }
    return total;
}
```

self-similarity

$n! = n * (n-1)!$

$0! = 1$

$4! = 4 * 3 * 2 * 1$

$5! = 5 * \underline{(4 * 3 * 2 * 1)}$

$= 5 * 4!$

Recursive factorial

```
  int factorial (int n)
  {
1     if ( n == 0 )   { // base case
2         return 1;
3     } else {    // recursive case
4         int total = factorial(n - 2);
5         total = total * n;
6         return total;
7     }
  }
```

will it reach here ever?

factorial (2) ?

$\underbrace{1\quad 3\quad 4}_{\text{input } n=2}\quad \underbrace{1\quad 3\quad 4}_{n=1}\quad \underbrace{1.\quad 2}_{n=0}\quad \underbrace{5\quad 6}_{n=1}\quad \underbrace{5\quad 6}_{n=2}$ end what happens if I call factorial (2)

Using a stack

fact (0) 1 2 return 1.

fact (1) 1 3 4

fact (2) 1 3 4

fact(1)

fact(2)

5 6 return 1 * 1.

fact(2) 5 6 return 2 * (1 * 1)

end