Recursions

October 25, 2019 3:06 PM

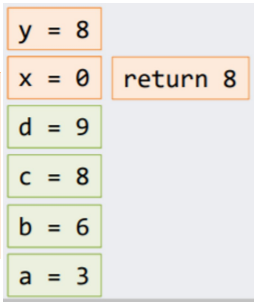
- a function or method call is an interruption or aside in the execution flow of a program
- stop, memorize where you were in the task, handle interruption, go back to what you were doing

Activation records on a computer

- computer handles function/method calls in same way

```
...
int a, b, c, d;
a = 3;
b = 6;
c = foo(a, b);
d = 9;
...
```

```
int foo(int x, int y) {
  while (x > 0) {
    y++;
    x >>= 1; // bitwise right shift by 1
  }
  return y;
}
```



Calculating Fibonacci series

```
int fib(int n)
{
  if (n <= 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

- fib(n) = 0 if n = 0, 1 if n = 1,
- otherwise fib(n) = fib(n – 1) + fib(n – 2)
- Fibonacci function is recursive
- a recursive function calls itself
- each cal to a recursive method results in a separate call to the method, with its own input
- do not use loops to repeat instructions, but use recursive calls in if statements
- recursive functions consist of two or more cases, there must be a base case and recursive case
- **Base case**
  - smaller problem with a known solution, not recursive
  - can be more than 1 base case
- **Recursive case** is the same problem with a smaller input
  - must include a recursive function call
  - can be more than one recursive case
  - if problem is small enough to be solved directly, solve it, otherwise:
    - recursively apply the algorithm to one or more smaller instances
    - use solutions from smaller instances to solve problem
- ex. fib(5)

```
int fib(int n)
{
  if (n <= 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

- ex. multi wrapped package, op

```
openPresent(pkg) {
  if you can see the actual gift
    say "thank you"
  else
    open the box to reveal spkg
    openPresent(spkg)
}
```

- ex. factorials
  - 6! = 6 · 5 · 4 · 3 · 2 · 1
  - 5! = 5 · 4 · 3 · 2 · 1
  - 6! = 6 · 5!
  - 0! = 1
  - $n! = n \cdot n - 1 \cdot n - 2 \cdot \cdots$

```
int factorial(int n) {
  if (n <= 0)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Designing recursive functions

- ex. max value in an array
  - max value is either current element or largest value in rest of array

- know we have largest element when subarray contains single element

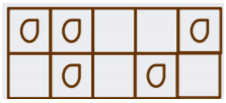
```
int arrayMax(int arr[], int size, int start) {
  if (start == size - 1)
    return arr[start];
  else
    return max(arr[start], arrayMax(arr, size, start+1));
}
```

- ex. summation

```
int summation(int n) {
  if (n <= 0)
    return 0;
  else
    return (n + summation(n-1));
}
```

$$\sum_{i=0}^{n} i = n + (n-1) + (n-2) + \cdots + 2 + 1 + 0$$
$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$$
$$\sum_{i=0}^{0} i = 0$$

- more than 1 recursive call
  - split into single squares



```
eatChocolateBar(b)
{
  if (b is a single square)
    if (b does not contain a nut)
      eat it
  else
    break the bar into two pieces
    eatChocolateBar(piece1)
    eatChocolateBar(piece2)
}
```

Stack overflow

- stack space limited, if many function invocations placed on stack without returning, stackoverflow can occur
- ex. if *summation(1000000)*
- for factorial example, may hit stack overflow before full products completed

Tail recursion

- function tail-recursive if recursive call i the absolute last thing function needs before returning
- no need to wait for a return from deeper recursive call to compute result