COL100 Lecture 27

Note Title 12-11-2018

Review: recursion examples
3 rules of recursion

isDirectory
listDirectory
getTail

```
void crawl(string filename,
           string indent)
{
 cout << indent << getTail(filename)
      << endl;
 for ( ..... )
 {
    if (isDirectory(...))
       crawl(...);
 }

}
```

Fibonacci series

n	fib(n)
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

→ 1 + 1

→ 2 + 1

→ 3 + 2

→ 5 + 3

→ 8 + 5

.
.
.

```
// assume n >= 1
int fib ( int n)
{
  if (n==1 || n==2)
  {
    return 1;
  } else {
    return fib(n-1)
         + fib(n-2);
  }
}
```

fib(6)

fib(5) + 2. fib(4)

1. fib(4) + fib(3)

fib(3) + fib(2)

fib(2) + fib(1)

fib(3) + fib(2)

fib(6):

fib(2) gets called 5 times

---

fib(20):

6 − 2 = 4

20 − 16 = 4

fib(16) gets called 5 times

↳ fib(12) gets called 5 times

25 times

save mappings or associations of the type

$n \longrightarrow fib(n)$

cache will store mappings for already computed expressions

Pseudo-code template

```
cache = {}          // empty
function f(args):
    if I have computed f(args) before:
        Look up f(args) in cache
    else:
        Actually compute f(args) result
        Store args → result mapping in cache
    return result
```

MEMOIZATION

Memoization:

caching results of previous expensive function calls so they do not need to be recomputed

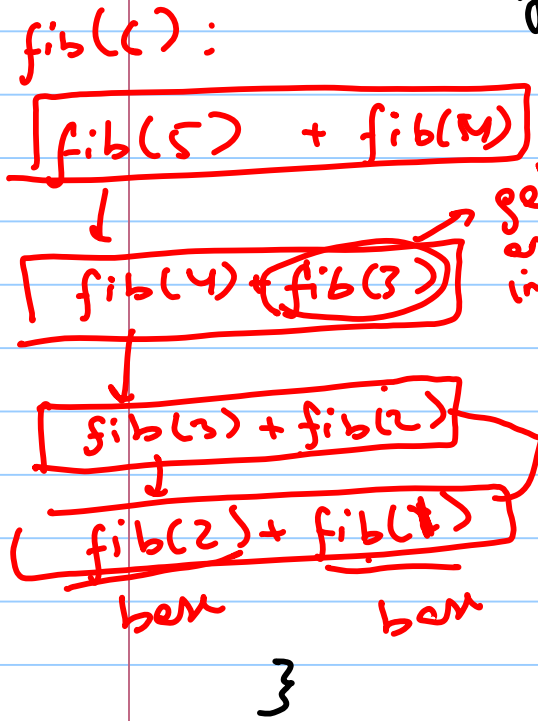
- Cache is often implemented by storing the call results in a collection

```
Map<int, int> cache;

int fib( int n )
{
    if (n == 1 || n == 2) { // base case
        return 1;
    } else if (cache.containsKey(n)) {
        return cache.at(n);
    } else {
        int result = fib(n-1) + fib(n-2);
        cache[n] = result;
        return result;
    }
}
```

// implement cache as pass-by-reference parameter to fib()

```
int fib_helper(int n, Map<int, int> & cache)
{
    if (n == 1 || n == 2) {
        return 1;
    } else if (cache.containsKey(n)) {
        return cache.at(n);
    } else {
        int result = fib_helper(n-1, cache)
                   + fib_helper(n-2, cache);
        cache[n] = result;
        return result;
    }
}
```



cache[3] → 2
cache[4] → 3
cache[5] → 5

```
int fib(int n)
{
    map<int, int> cache;
    return fib_helper(n, cache);
}
```

wrapper function

function that does some initial preparatory work, then calls a recursive function (or any other function)

Tail recursion

When a recursive call is made as the final action of a recursive function

```
int mystery ( int n )
{
    if (n < 10) {
        return n ;
    } else {
        int a = n / 10 ;
        int b = n % 10 ;
        return mystery ( a + b ) ;
    }
}
```

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

Not Tail recursive

Advantage of tail recursion :

e.g.

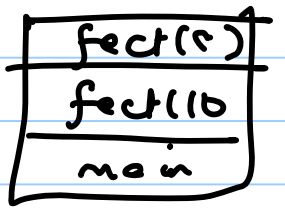
mystery (25?) :

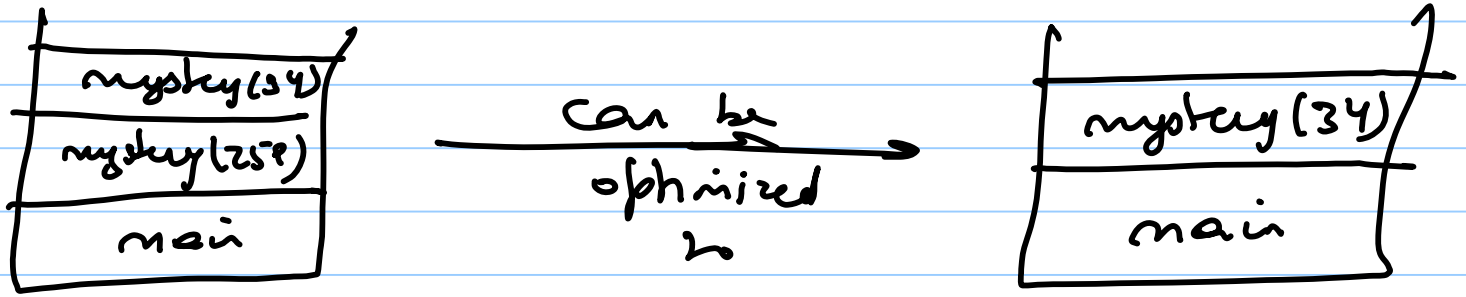
mystery (34)

eg. fact

fact (10)

fact (9)



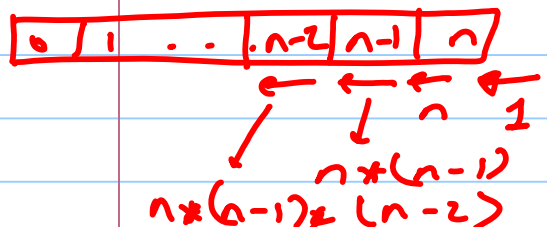


Often, I can convert non tail-recursive functions to tail-recursive:

result so far

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

int fact ( int n)
{
    return fact_helper(n, 1);
}
```



Tail recursion

fact (3)
↓
fact_helper(3,1)
↓
fact_helper (2, 3)
↓
fact_helper(1, 6)
← fact_helper(0, 6)
← return 6