Note Title 12-11-2018

COMP 100
NOV 12, 2018

Recursion

Computing Cartesian Product:

$a = \{"A", "B"\}$ → m = size of first set

$b = \{"1", "2"\}$ → n = size of second set

$O(mn)$

$a \times b = \{(A,1), (A,2), (B,1), (B,2)\}$

$\{A1, A2, B1, B2\}$

Base Case:- case · at most one element in each set

↓ Recursive Case

$a = \{"A"\} \cup a'$ where $a' = a - \{"A"\}$

$b = b$

$a \times b = \{"A"\} \times b \cup a' \times b$

$a = a_1 \cup a_2$

b

$a \times b = (a_1 \times b) \cup (a_2 \times b)$

if (a1.size() == 1)

$a_1 \times b = (a_1 \times b_1) \cup (a_1 \times b_2)$

$m = 1$ n otherwise $b = b_1 \cup b_2$

$m = 1$ $n = 1$ $m - 1$ $n - 1$

```
set<string> cartesianProduct(set<string> a, set<string> b){
  //Base Case 1
  if((a.size()==0) || (b.size()==0)){
      set<string> ans;
      return ans;
  }
  //Base Case 2
  if((a.size()==1) && (b.size()==1)){
      string e1 = a.first(); // → *a.begin();
      string e2 = b.first(); // * b.begin();
      set<string> ret;
      ret.insert(e1+e2);
      return ret;
  }
}
```

C++

// Recursive call 1:-

// Recursive call 2:-

// Recursive call 1:-

// Recursive call 2:-

```
if (a.size() > 1) {
    string e1 = a.first();
    set<string> a1, a2;
    a1.insert(e1)
    a2 = a;
    a2.remove(e1);
    return setUnion(cartesianProduct(a1,b), cartesianProduct(a2,b));
} else {
    string e2 = b.first();
    set<string> b1, b2;
    b1.insert(e2);
    b2 = b;
    b.remove(e2);
    return setUnion(cartesianProduct(a,b1), cartesianProduct(a,b2));
}
```

{A1, A2, B1, B2}

O(mn)

O(nlogn)

→ Base Cond:- n≥0 or n=2

// fib(n) = fib(n-1) + fib(n-2)

Fibonacci:-

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 - -

0th 1st

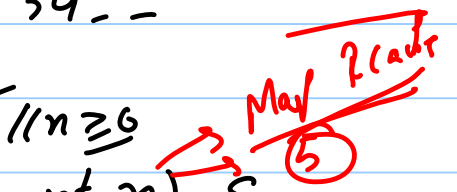
```
int fib(int n) {
    if (n==0)
        return 0;
    if (n==1)
        return 1;
    int prev1=0, prev2=1;
    for (int i=2; i<=n; i++) {
        res = prev1 + prev2;
        prev1 = prev2;
        prev2 = res;
    }
    return res;
}
```

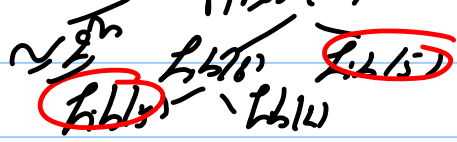
Time = O(n)

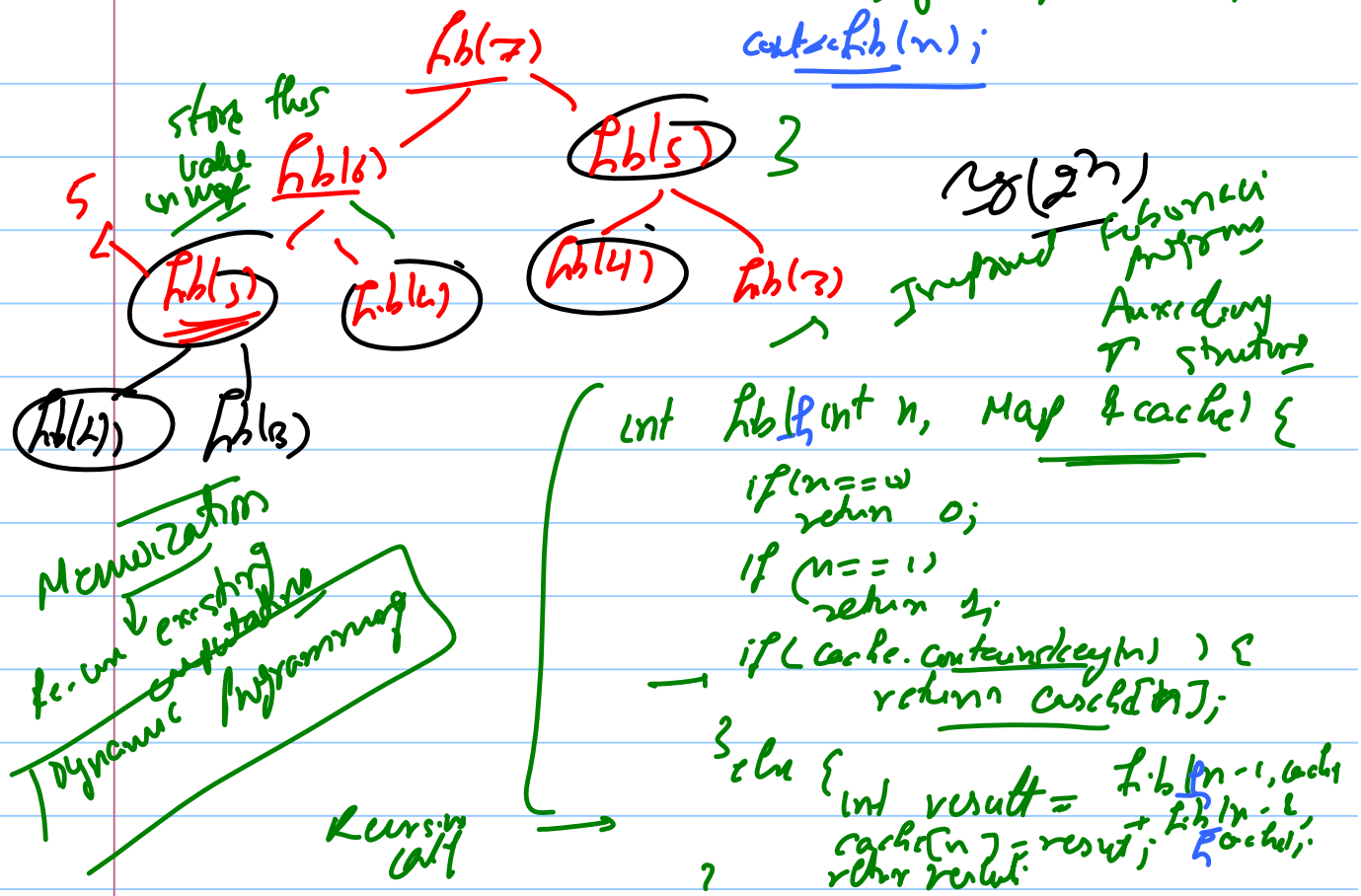
Recursive // n≥0

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

Max 2 calls






int main() {
n = getInteger("n:");
cout << fib(n);
fib(7)
store this value in map
fib(6)
fib(5)
fib(5)
fib(4)
fib(3)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
O(2^n)
Inefficient Fibonacci program
Auxiliary structure
int fib(int n, Map &cache) {
if (n == 0)
return 0;
if (n == 1)
return 1;
if (cache.containsKey(n)) {
return cache[n];
} else {
int result = fib(n-1, cache) + fib(n-2, cache);
cache[n] = result;
return result;
}
Memoization
Recursion + existing computation
Dynamic Programming
Recursive call

~~fib(n)~~ // wrapper function

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

Memoization:-

```
cache = {};  // empty
function f(args) {
    if I have computed f(args) before:
        Look for f(args) in cache
    else
        compute f(args);
        store result in cache
}
```

Factorial(n) Iterative Ingram

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

```
fact.helper(n, 1);
```

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

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

Tail recursion