COMP242

Data Structure

Lectures Note: Recursion

Prepared by: **Dr. Mamoun Nawahdah**

2016/2017

Recursion (Time Analysis Revision)

Example 1: Write a recursive method to calculate the sum of squares of the first ***n*** natural numbers. ***n*** is to be given as an input.

```
public int sumOfSquares(int n) {
    if (n==1)
        return 1;
    return   (n*n) + sumOfSquares(n-1);
}
```

Recursion may sometimes be very intuitive and simple, but it may not be the best thing to do.

Example 2: **Fibonacci sequence**:

F(n) = n if n=0, 1 ; F(n) = F(n-1) + F(n-2) if n > 1

0	1	1	2	3	5	8	13	..
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	..

Solution 1: **Iterative**

```
public static int fib1(int n){
    if(n<=1) return n;
    int f1 = 0,   f2 = 1,   res=0;
    for(int i=2; i<=n; i++){
        res =f1+f2;
        f1=f2;
        f2=res;
    }
    return res;
}
```

Solution 2: **Recursion**

```
public static int fib2(int n){
    if(n<=1) return n;
    return (fib2(n-1)+fib2(n-2));
}
```

Test for n=6 and n=40

Why recursive solution is taking much time?

Do analyze the 2 algorithms in term of calculating F(n)

In **Solution 1**:

We have **F(0)** and **F(1)** given

Then we calculate F(2) using F(1) and F(0)

F(3) using F(2) and F(1)

F(4) using F(3) and F(2)

:

F(n) using F(n-1) and F(n-2)

In **Solution 2**:

F(5)

F(4) **F(3)**

F(3) **F(2)** **F(2)** F(1)

F(2) F(1) F(1) F(0) F(1) F(0)

F(1) F(0)

Note: we are calculating the same value multiple times!!

n	F(2)	F(3)	..
5	3	2	
6	5		
8	13		
:			
40	**63245986**		

Exponential growth

Time and Space complexity Analysis of recursion

Example: recursive factorial

```
fact(n){
        If (n==0) return 1;
        Return n *  fact(n-1);
}
```

- Calculate operation costs:
  - **If** statement takes 1 unit of time
  - Multiplication (*) takes 1 unit of time
  - Subtraction (-) takes 1 unit of time
  - Function call
- So $T(0) = 1$
  $T(n) = 3 + T(n-1)$ for $n > 0$

To solve this equation, reduce **T(n)** in term of its base conditions.

$T(n) = T(n-1) + 3$

$= T(n-2) + 6$

$= T(n-3) + 9$

:

$= T(n-k) + 3k$

For $T(0)$ ➔ $n-k = 0$ ➔ $n = k$

Therefore **T(n)** $= T(0) + 3n$

$= 1 + 3n$ ➔ **O(n)**

Space analysis:

Recursive Tree

Fact(5) → Fact(4) → Fact(3) → Fact(2) → Fact(1) → Fact(0)

Each function call will cause to save current function state into memory (call stack, push):

Fact(1)
Fact(2)
Fact(3)
Fact(4)
Fact(5)

Each return statement will retrieve previous saved function state from memory (pop):

So needed space is proportional to n ➔ **O(n)**

Fibonacci sequence time complexity analysis

```
public static int fib2(int n){
     if(n<=1) return n;
     return (fib2(n-1)+fib2(n-2));
}
```

- Calculate operation costs:
  - **If** statement takes 1 unit of time
  - 2 subtractions (**-**) takes 2 unit of time
  - 1 addition (**+**) takes 1 unit of time
  - 2 function calls
- So $T(0) = T(1) = 1$

  $T(n) = T(n-1) + T(n-2) + 4$ for $n > 1$

To solve this equation, reduce T(n) in term of its base conditions.

For approximation assume **$T(n-1) \approx T(n-2)$** ➔ in reality **$T(n-1) > T(n-2)$**

$T(n) = 2\,T(n-2) + 4$ → $c = 4$

$= 2\,T(n-2) + c$ → $T(n-2) = 2\,T(n-4) + c$

$= 2\,\{ 2\,T(n-4) + c \} + c$

$= 4\,T(n-4) + 3c$

$= 8\,T(n-6) + 7c$

$= 16\,T(n-8) + 15c$

:

$= 2^k\,T(n-2k) + (2^k-1)c$

For **T(0)** ➔ $n-2k = 0$ ➔ $k = n/2$

Therefore $T(n) = 2^{n/2}\,T(0) + (2^{n/2} - 1)\,c$ ➔ $2^{n/2}\,(1+c) - c$

T(n) is proportional to $2^{n/2}$ ➔ **$O(2^{n/2})$** ⬅ lower bound analysis

Similarly, for approximation assume $T(n-2) \approx T(n-1)$ → in reality $T(n-2) < T(n-1)$

$$
\begin{aligned}
T(n) &= 2\,T(n-1) + c \quad \rightarrow \quad T(n-1) = 2\,T(n-2) + c \\
&= 2\,\{ 2\,T(n-2) + c \} + c \\
&= 4\,T(n-2) + 3c \\
&= 8\,T(n-3) + 7c \\
&= 16\,T(n-4) + 15c \\
&\;\vdots \\
&= 2^k\,T(n-k) + (2^k-1)c
\end{aligned}
$$

For $T(0)$ → $n-k = 0$ → $k = n$

Therefore $T(n) = 2^n\,T(0) + (2^n - 1)\,c$ → $2^n\,(1+c) - c$

$T(n)$ is proportional to 2^n → **$O(2^n)$** ← upper bound analysis → worst case analysis

While for iterative solution → **O(n)**

Recursion with memorization

Solution: don't calculate something already has been calculated.

Algorithm:

```
fib(n){
        If (n<=1)   return n
        If(F[n] is in memory)   return F[n]
        F[n] =  fib(n-1) + fib(n-2)
        Return F[n]
}
```

Time complexity → **O(n)**

Calculate X^n using recursion

Iterative solution: **O(n)** $X^n = X * X * X * X * \ldots * X$ n-1 multiplication	Recursive solution 1: **O(n)** $X^n = X * X^{n-1}$ if $n > 0$ $X^0 = 1$ if $n > 0$	Recursive solution 2: **O(log n)** $X^n = X^{n/2} * X^{n/2}$ if n is even $X^n = X * X^{n-1}$ if n is odd $X^0 = 1$ if $n > 0$

res = 1
for i←1 to n
  res ← res * x
	pow(x, n){
	  if n==0 return 1
	  return x * pow(x, n-1)
	}
		pow(x, n){
		  if n==0 return 1
		  if n%2 == 0 {
		    y ← pow(x, n/2)
		    return y * y
		  }
		  return x * pow(x, n-1)
		}

Recursive solution 1: Time analysis

T(1) = 1
T(n) = T(n-1) + c
= (T(n-2) + c) + c → T(n-2) + 2c
= T(n-3) + 3c
:
= T(n-k) + kc
For T(0) → n-k = 0 → n = k
T(n) = T(0) + nc → 1 + nc → **O(n)**

Recursive solution 2: Time analysis

- $X^n = X^{n/2} * X^{n/2}$ if n is even
- $X^n = X * X^{n-1}$ if n is odd
- $X^n = 1$ if n == 0
- $X^n = X * 1$ if n == 1

If even → T(n) = T(n/2) + c1
If odd → T(n) = T(n-1) + c2
If 0 → T(0) = 1
If 1 → T(1) = c3

If odd, next call will become even:
T(n) = T((n-1)/2) + c1 + c2
If even
T(n) = T(n/2) + c
= T(n/4) + 2c
= T(n/8) + 3c
:
= $T(n/2^k)$ + k c
For T(1) → T(0) + c → 1
$n/2^k = 1$ → $n = 2^k$ → k = log n
= c3 + c log n → **O(log n)**