

Time Analysis: Examples

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:

Solution 1: Iterative \rightarrow O(n)

Solution 2: Recursion

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

Test for n=6 and n=40

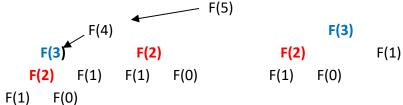
Why recursive solution is taking much time?

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)
```



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

- Calculate operation costs:
 - If statement takes 1 unit of time
 - Multiplication (*) takes 1 unit of time
 - Subtraction (-) takes 1 unit of time
 - o 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. This called **recurrence equation**¹ analysis that describes the running time of an algorithm.

¹ A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).



```
T(n) = T(n-1) + 3 \rightarrow T(n-1) = T(n-2)+3
= T(n-2) + 6 \rightarrow T(n-2) = T(n-3)+3
= T(n-3) + 9
```

For
$$T(0) \rightarrow n-k = 0 \rightarrow n = k$$

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

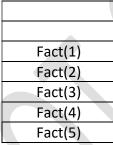
= 1 + 3n \rightarrow $O(n)$

Space analysis:

Recursive Tree

$$Fact(5) \rightarrow Fact(4) \rightarrow Fact(3) \rightarrow Fact(2) \rightarrow Fact(1) \rightarrow Fact(0)$$

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



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

So needed space is proportional to $n \rightarrow O(n)$

Fibonacci sequence time complexity analysis

- Calculate operation costs:
 - If statement takes 1 unit of time
 - 2 subtractions (-) takes 2 unit of time
 - o 1 addition (+) takes 1 unit of time
 - o 2 recursive function calls

• So
$$T(0) = T(1) = 1$$

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

While for iterative solution \rightarrow O(n)

Recursion with memorization

T(n) is proportional to 2ⁿ

Solution: Do not calculate something already has been calculated.

 \rightarrow $O(2^n)$

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]
}</pre>
```

Time complexity \rightarrow O(n)



upper bound analysis → worst case analysis

Calculate Xⁿ using recursion

Iterative solution: O(n) X ⁿ = X * X * X * X * * X n-1 multiplication	Recursive solution 1: $O(n)$ $X^n = X * X^{n-1} \text{ if } n > 0$ $X^0 = 1 \text{ if } n = 0$	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	<pre>pow(x, n){ if n==0 return 1 return x * pow(x, n-1) }</pre>	<pre>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) }</pre>

Recursive solution 1: Time analysis

T(1) = 1
T(n) = T(n-1) + c
=
$$(T(n-2) + c) + c \rightarrow T(n-2) + 2c$$

= $T(n-3) + 3c$
:
= $T(n-k) + kc$
For T(0) \rightarrow $n-k = 0 \rightarrow n = k$
T(n) = T(0) + nc \rightarrow 1 + nc \rightarrow 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
$$\rightarrow$$
 T(n) = T(n/2) + c₁

If odd
$$\rightarrow$$
 T(n) = T(n-1) + c₂

If
$$0 \rightarrow T(0) = 1$$

If
$$1 \rightarrow T(1) = c_3$$

If odd, next call will become even:

$$T(n) = T((n-1)/2) + c_1 + c_2$$

The Data Structure: Lectures Note

If even

T(n) = T(n/2) + c
= T(n/4) + 2c
= T(n/8) + 3c = T(n/2³) + 3c
:
= T(n/2^k) + k c
For T(1)
$$\rightarrow$$
 T(0) + c \rightarrow 1
n/2^k = 1 \rightarrow n = 2^k \rightarrow k = log n
= c3 + c log n \rightarrow O(log n)

35

Big-O Exercises

Exercise 1:

```
void fun(int n, int[] arr) {
    int i = 0, j = 0;
    for( ; i < n; ++i)
        while(j < n && arr[i] < arr[j])
        j++;
}</pre>
```

Solution 1:

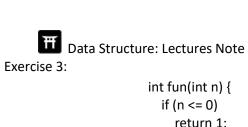
T(n) = O(n), since the inner while loop will run n times during the entire outer for loop. Outer loop will run n times as well. So T(n) = O(2n), 2 is constant.

Exercise 2:

```
int fun(int n) {
      if (n <= 1)
          return n;
      return fun (n-1) + fun (n-1); // 2 f(n-1)
}</pre>
```

Solution 2:

```
T(n) =  2T(n-1) + c \qquad n > 1 
T(n) = 2T(n-1) + c \qquad r(n-1) = 2T(n-2) + c 
T(n) = 2^{2}T(n-2) + 2c \qquad r(n-2) = T(n-3) + c 
T(n) = 2^{3}T(n-3) + 3c \qquad \vdots 
T(n) = 2^{k}T(n-k) + kc 
 Let n-k = 0 \Rightarrow n = k 
T(n) = 2^{n}T(0) + nc \Rightarrow T(n) = 2^{n} d + nc \Rightarrow T(n) = O(2^{n})
```



```
if (n <= 0)

return 1;

return 1 + fun(n-5);

}
```

Solution 3:

$$T(n) = T(n-5) + c$$

$$T(n-5) = T(n-10) + c$$

$$T(n) = T(n-10) + 2c$$

$$T(n-10) = T(n-15) + c$$

$$T(n) = T(n-15) + 3c$$

$$T(n-15) = T(n-20) + c$$

$$T(n) = T(n-20) + 4c$$

$$T(n-20) = T(n-25) + c$$
...
$$T(n-(n-5)) = T(0) + c$$

$$T(n) = T(0) + n/5 c \rightarrow T(n) = d + n/5 c \rightarrow T(n) = O(n)$$

Exercise 4:

```
int fun(int n) {
   if (n <= 0)
      return 1
   return 1 + fun(n/5);
}</pre>
```

Solution 4:

$$T(n) = 0$$
 $T(n/5)+c$
 $n <= 0$

$$T(n) = T(1) + \log_5(n) c \rightarrow T(n) = d + \log_5(n) c \rightarrow T(n) = O(\log_5 n)$$

```
Ħ
```

```
Data Structure: Lectures Note
```

```
Exercise 5:
```

```
int isIn(int A[], int k, int x) {
                                    if (k >= A.length)
                                             return 0;
                                    if (A[k] == x)
                                             return 1;
                                    return isIn(A, k+1, x)
                           }
Solution 5:
                           d
                                             n <= 1
        T(n) =
                           T(n-1)+c
                                             n > 1
         T(n) = T(n-1) + c
                  T(n-1) = T(n-2)+2c
                 T(n-(n-1)) = T(1)+nc
         T(n) = T(1) + nc
         T(n) = d + nc \rightarrow T(n) = O(n)
```

Exercise 6:

Solution 6:

For input integer n, the innermost statement of fun() is executed following times.

```
n + n/2 + n/4 + ... + 1
```

So time complexity T(n) can be written as

```
T(n) = O(n + n/2 + n/4 + ... 1) \rightarrow O(n)
```

The value of count is also n + n/2 + n/4 + .. + 1

$$T(n) = n + T(n/2) \rightarrow n + n/2 + T(n/4) \rightarrow n + n/2 + n/4 + T(n/8) + ... + 1 \rightarrow n + n-1 \rightarrow 2n -1$$