江西理工大学 Jiangxi University of Science and Technology
信息工程学院 School of information engineering

高级算法分析与设计
Advanced Algorithm Analysis and Design

Lecture 04:
Time complexity(B)

Dr Ata Jahangir Moshayedi

EMAIL: ajm@jxust.edu.cn

Prof Associate ,
School of information engineering Jiangxi university of science and technology, China

LIVE Lecture series

Autumn _2022

高级算法分析与设计

Advanced Algorithm Analysis and Design

LECTURE 04: Time complexity_B

时间复杂度 _B

Agenda

目录
CONTENTS

江西理工大学
JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY

01

Constant complexity - O( )
恒定复杂度- O( )

Big-O notation:

Def

f(n) = O(g(n)) if and only if ∃ 2 positive constants c and n_0,

such that

$$|f(n)| \leq c \bullet |g(n)| \; \forall \; n \geq n_0.$$

So, g(n) actually is the upper bound of f(n).

Def

f(n) = O(g(n))当且仅当存在2个正常数c和n0

因此

$$|f(n)| \leq c \bullet |g(n)| \; \forall \; n \geq n_0.$$

g(n)实际上是f(n)的上界。

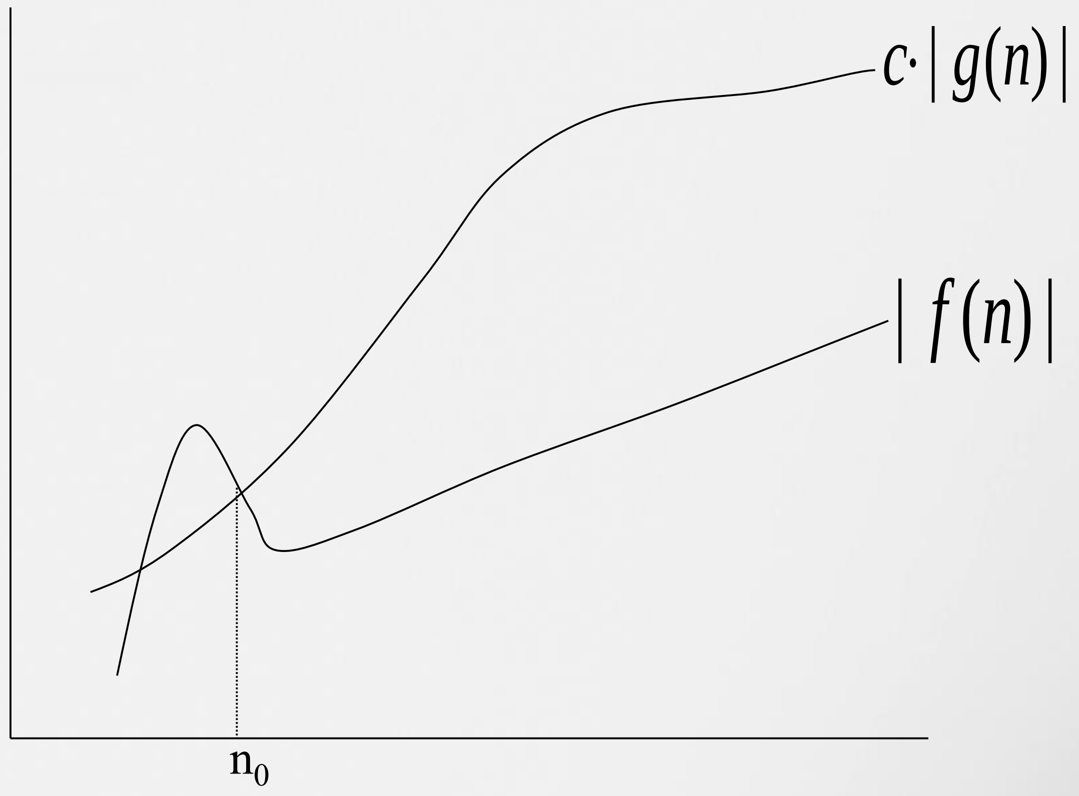


Figure 1. Illustrating “big O”

Constant complexity - O(1)

- The most efficient algorithm, in theory, runs in constant time and consumes a constant amount of memory.
- An example of an algorithm with constant complexity:No matter how you increase the amount of incoming data, the complexity of the algorithm will not increase. The efficiency (or complexity) of such an algorithm is called constant and is written as **O(1)**.

```
const getArrayElement = (arr, i) => arr [i];
```

- As an input, we get the array arr and the index i. We return the array element at position i.
- The speed of execution and the amount of required memory of this algorithm doesn’t depend on the length of the input array, so the complexity is considered constant.

恒定复杂度- O(1)

- 理论上，最有效的算法在恒定时间内运行，消耗恒定数量的内存。
- 一个复杂度恒定的算法示例:无论你如何增加输入数据的数量，算法的复杂度都不会增加。这种算法的效率(或复杂度)称为常数，并写为O(1)。

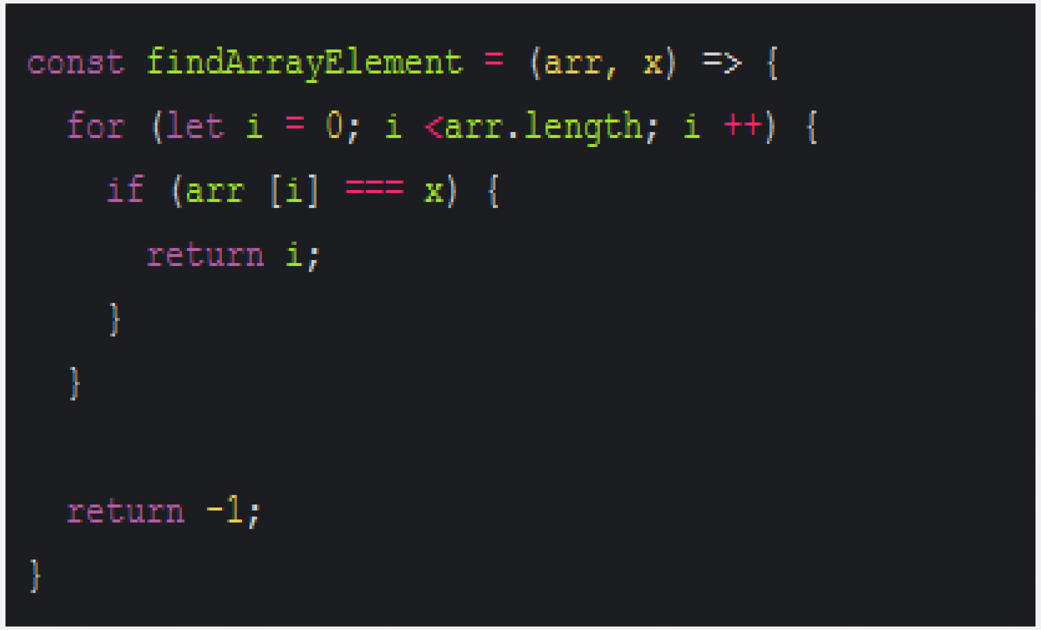
```
const getArrayElement = (arr, i) => arr [i];
```

- 作为一个输入，我们得到数组arr和下标i，我们返回位置i的数组元素。
- 该算法的执行速度和所需的内存量不依赖于输入数组的长度，因此复杂度被认为是恒定的。

Linear complexity - O(n)
线性复杂度- O(n)

- The complexity of many algorithms grows in direct proportion to the amount of incoming data.
- A good example is a linear search:

- 许多算法的复杂度与输入数据量成正比。
- 线性搜索就是一个很好的例子:



```
const findArrayElement = (arr, x) => {
  for (let i = 0; i <arr.length; i ++) {
    if (arr [i] === x) {
      return i;
    }
  }

  return -1;
}
```

- If a search in an array of 10 elements takes 5 seconds, then in an array of 100 elements we will search for 100 seconds. 10 times longer.
- This complexity (efficiency) is called linear and is written as **O(n)**.
- But, note that the linear search algorithm does not need additional memory. Therefore, for this characteristic, it gets the highest score - **O(1)**.

- 如果在一个包含10个元素的数组中搜索需要5秒，那么在一个包含100个元素的数组中搜索需要100秒。10倍的时间。
- 这种复杂度(效率)被称为线性的，并被写成O(n)。
- 但是，注意线性搜索算法不需要额外的内存。因此，对于这个特征，它得到了最高的分数- O(1)。

二次复杂度- O(n²^)
Quadratic complexity - O(n^2^)

- Simple sorting algorithms like bubble sort or selection sort have **O(n^2^)** complexity.
- It is not very efficient. If the length of the array grows 10 times, the execution time of the algorithm will increase 10^2^ (100) times!
- This happens because the algorithm uses a nested loop. We estimate the complexity of one pass as **O(n)**. But we go through it not once, but **n** times. Therefore, the total execution time is estimated as **O(n*n) = O(n^2^)**.

- 简单的排序算法如冒泡排序或选择排序有O(n^2^)的复杂度。
- 这不是很有效。如果数组的长度增加10倍，算法的执行时间将增加10^2^(100)倍!
- 这是因为算法使用了嵌套循环。我们估计一次通过的复杂度为O(n)。但我们不止经历一次，而是n次。因此，总执行时间估计为O(n*n) = O(n²^)。

Logarithmic complexity - O(log n)

- logarithms that can reduce the amount of analyzed data at each iteration have a logarithmic complexity - **O(log n)**.
- Binary search is a good example of an algorithm with logarithmic complexity. At each iteration, we discard half of the data and work only with the other half.

- 能够在每次迭代中减少分析数据量的对数具有对数复杂度- O(log n)。
- 二分搜索是具有对数复杂度的算法的一个很好的例子。在每次迭代中，我们丢弃一半的数据，只处理另一半数据。

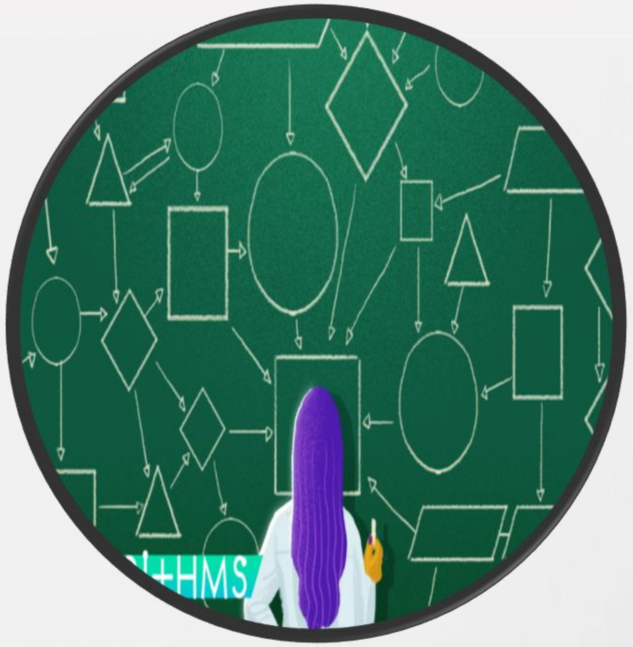
02

Let us review

让我们回顾一下

Efficiency 效率

- Computer Scientists don’t just write programs.
- They also ***analyze*** them.
- How efficient is a program?
  - How much time does it take program to complete?
  - How much memory does a program use?
  - How do these change as the amount of data changes?
  - What is the difference between the average case and worst-case efficiency if any?



- 计算机科学家不只是写程序。
- 他们也会分析。
  - 一个程序的效率如何?
  - 完成一个程序需要多少时间?
  - 一个程序使用多少内存?
  - 这些是如何随着数据量的变化而变化的呢?
  - 平均情况下的效率和最坏情况下的效率有什么区别?

Sequential Statements

顺序语句

- If we have statements with basic operations like comparisons, assignments, reading a variable. We can assume they take constant time each O(1).

- 如果我们有包含比较，赋值，读取变量等基本操作的语句。我们可以假设它们每个O(1)花费的时间是常数。

1. statement1;
2. statement2;
3. ...
N. statementN;

1. 语句1;
2. 语句2;
3. ...
N. 语句N;

- If we calculate the total time complexity, it would be something like this:

```
total =
 time(statement1) + time(statement2) + ... time (statementN)
```

- 如果我们计算总的时间复杂度，它会是这样的:

- Let’s use T(n) as the total time in function of the input size n, and t as the time complexity taken by a statement or group of statements.

- 让我们用T(n)作为输入大小n的总时间函数，用T作为一条或一组语句的时间复杂度。

```
T(n) = t(statement1) + t(statement2) + ... + t(statementN);
```

- If each statement executes a basic operation, we can say it takes constant time O(1).
- As long as you have a fixed number of operations, it will be constant time, even if we have 1 or 100 of these statements.

- 如果每个语句执行一个基本操作，我们可以说它需要常数时间O(1)。
- 只要你有固定数量的操作，它将是恒定的时间，即使我们有1个或100个这样的语句。

Example: 例如:

- Let’s say we can compute the square sum of 3 numbers.
  - 假设我们可以计算三个数的平方和。

```
function squareSum(a, b, c) {
  const sa = a * a;
  const sb = b * b;
  const sc = c * c;
  const sum = sa + sb + sc;
  return sum;
}
```

- As you can see, each statement is a basic operation (math and assignment).
- Each line takes constant time O(1).
- If we add up all statements’ time it will still be O(1).
- It doesn’t matter if the numbers are 0 or 9,007,199,254,740,991, it will perform the same number of operations.
- ⚠ Be careful with function calls. You will have to go to the implementation and check their run time. More on that later.

- 如您所见，每个语句都是一个基本操作(数学和赋值)。
- 每条直线花费恒定的时间O(1)。
- 如果我们把所有语句的时间加起来，它仍然是O(1)。
- 不管数字是0还是9,007,199,254,740,991，它都将执行相同数量的操作。
- ⚠注意函数调用。您必须转到实现并检查它们的运行时。稍后再详细介绍。

Conditional Statements

- Very rarely, you have a code without any conditional statement. How do you calculate the time complexity?
- Remember that we care about the worst-case with Big O so that we will take the maximum possible runtime.

- 很少有代码没有任何条件语句。如何计算时间复杂度?
- 记住，我们关心的是大O的最坏情况，这样我们就可以得到最大的可能运行时间。

```
1. if (isValid) {
2.   statement1;
3.   statement2;
4. } else {
5.   statement3;
6. }
```

- Since we are after the worst-case we take whichever is larger:
  - 因为我们在寻找最坏情况，所以取较大的值:

T(n) = Math.max([t(statement1) + t(statement2)], [time(statement3)])

```
if (isValid) {
  array.sort();
  return true;
} else {
  return false;
}
```

- What’s the runtime? The if block has a runtime of O(n log n) ,that’s common runtime for efficient sorting algorithms. The else block has a runtime of O(1). 运行时是什么?if块的运行时为O(n log n)(这是高效排序算法的常见运行时)。else块的运行时为O(1)。
- So we have the following: 所以我们有以下内容:

O([n log n] + [n]) => O(n log n)

Since n log n has a higher order than n, we can express the time complexity as O(n log n).

由于n log n的阶数比n高，我们可以将时间复杂度表示为O(n log n)。

Loop Statements

- **Linear Time Loops**
- For any loop, we find out the runtime of the block inside them and multiply it by the number of times the program will repeat the loop.

```
1. for (let i = 0; i < array.length; i++) {
2.   statement1;
3.   statement2;
4. }
```

For this example, the loop is executed array.length, assuming n is the length of the array, we get the following: `T(n) = n * [ t(statement1) + t(statement2) ]`

- All loops that grow proportionally to the input size have a linear time complexity O(n).
- If you loop through only half of the array, that's still O(n).
- Remember that we drop the constants so 1/2 n => O(n).

循环语句

- **线性时间循环**
- 对于一些循环，我们找出其中块的运行时，并将其乘以程序重复循环的次数。

```
for (let i = 0; i < array.length; i++) {
  statement1;
  statement2;
}
```

对于本例，循环执行为array。长度，假设n是数组的长度，我们得到如下:

```
T(n) = n * [ t(statement1) + t(statement2) ]
```

- 所有与输入大小成比例增长的循环都具有线性时间复杂度O(n)。
- 如果只遍历数组的一半，仍然是O(n)
- 记住，我们去掉常数1/2 n => O(n)

Constant-Time Loops
常量时间的循环

- However, if a constant number bounds the loop, let’s say 4 (or even 400).
  但是，如果循环的边界是一个常量，我们设为4(甚至400)。
- Then, the runtime is constant O(4) -> O(1).
- 那么，运行时为常数O(4) -> O(1)。

```
for (let i = 0; i < 4; i++) {
statement1;
  statement2;
}
```

- That code is O(1) because it no longer depends on the input size.
- It will always run statement 1 and 2 four times.
- 该代码是O(1)，因为它不再依赖于输入大小。
- 它总是运行语句1和语句2四次。

Logarithmic Time Loops

- Consider the following code, where we divide an array in half on each iteration (binary search):
- 考虑下面的代码，我们在每次迭代(二进制搜索)时将一个数组分成两半:

```
function fn1(array, target, low = 0, high = array.length - 1) {
 let mid;
  while ( low <= high ) {
    mid = ( low + high ) / 2;
    if ( target < array[mid] )
      high = mid - 1;
    else if ( target > array[mid] )
      low = mid + 1;
    else break;
  }
  return mid;
}
```

- This function divides the array by its middle point on each iteration.
- The while loop will execute the amount of times that we can divide array.length in half.
- We can calculate this using the log function. E.g. If the array's length is 8, then we the while loop will execute 3 times because $\log_2(8) = 3$.

- 这个函数在每次迭代时都除以数组的中点。
- while循环将执行数组可以除的次数。长度的一半。
- 我们可以用log函数来计算。例:如果数组的长度是8，那么 while循环将执行3次，因为log2(8) = 3。

Nested loops statements

- Sometimes you might need to visit all the elements on a 2D array (grid/table).
- For such cases, you might find yourself with two nested loops.
- 有时你可能需要访问2D数组(网格/表)中的所有元素。
- 对于这种情况，你可能会发现自己有两个嵌套循环。

```
for (let i = 0; i < n; i++) {
  statement1;
  for (let j = 0; j < m; j++) {
    statement2;
    statement3;
  }
}
```

- For this case, you would have something like this:
- 在这种情况下，你会得到这样的东西:

$$T(n) = n * [t(statement1) + m * t(statement2...3)]$$

- Assuming the statements from 1 to 3 are O(1), we would have a runtime of O(n * m).
- If instead of m, you had to iterate on n again, then it would be O(n^2).
- Another typical case is having a function inside a loop.

- 假设从1到3的语句是O(1)，我们将有O(n * m)的运行时。
- 如果不是m，而是n，那么它就是O(n²)
- 另一种典型情况是在循环中有一个函数。

Function call statements
函数调用语句

- When you calculate your programs' time complexity and invoke a function, you need to be aware of its runtime. If you created the function, that might be a simple inspection of the implementation. If you are using a library function, you might need to check out the language/library documentation or source code

- 在计算程序的时间复杂度并调用函数时，需要了解它的运行时。如果您创建了函数，那么这可能是对实现的简单检查。如果使用库函数，可能需要查看语言/库文档或源代码

- Let's say you have the following program:
- 假设你有以下程序:

```
for (let i = 0; i < n; i++) {
fn1();
 for (let j = 0; j < n; j++) {
    fn2();
    for (let k = 0; k < n; k++) {
      fn3();
    }
 }
 }
```

Depending on the runtime of fn1, fn2, and fn3, you would have different runtimes.

- •If they all are constant O(1), then the final runtime would be O(n^3).
- •However, if only fn1 and fn2 are constant and fn3 has a runtime of O(n^2), this program will have a runtime of O(n^5).
- •Another way to look at it is, if fn3 has two nested and you replace the invocation with the actual implementation, you would have five nested loops.

In general, you will have something like this: T(n) = n * [t(fn1()) + n * [t(fn2()) + n * [t(fn3())]]]

根据fn1、fn2和fn3的运行时，您将拥有不同的运行时。

- 如果它们都是常数O(1)，那么最终的运行时间将是$O(n^3)$。
- 然而，如果只有fn1和fn2是常数，而fn3的运行时为O(n^2)，则该程序的运行时为O(n^5)。
- 另一种看待它的方法是，如果fn3有两个嵌套，并且您用实际的实现替换调用，那么您将有五个嵌套循环。

一般来说，你会得到这样的结果: T(n) = n * [t(fn1()) + n * [t(fn2()) + n * [t(fn3())]]]

Recursive Functions Statements

Analyzing the runtime of recursive functions might get a little tricky. There are different ways to do it. One intuitive way is to explore the recursion tree.

Let’s say that we have the following program:

```
function fn(n) {
  if (n < 0) return 0;
  if (n < 2) return n;
  return fn(n - 1) + fn(n - 2);
}
```

You can represent each function invocation as a bubble (or node).

Let’s do some examples:

- When you n = 2, you have 3 function calls. First fn(2) which in turn calls fn(1) and fn(0).
- For n = 3, you have 5 function calls. First fn(3), which in turn calls fn(2) and fn(1) and so on.
- For n = 4, you have 9 function calls. First fn(4), which in turn calls fn(3) and fn(2) and so on.

Since it’s a binary tree, we can sense that every time n increases by one, we would have to perform at most the double of operations.

递归函数语句

分析递归函数的运行时可能有点棘手。做这件事有不同的方法。一种直观的方法是探索递归树。

假设我们有以下程序:

```
function fn(n) {
  if (n < 0) return 0;
  if (n < 2) return n;
  return fn(n - 1) + fn(n - 2);
}
```

可以将每个函数调用表示为气泡(或节点)。

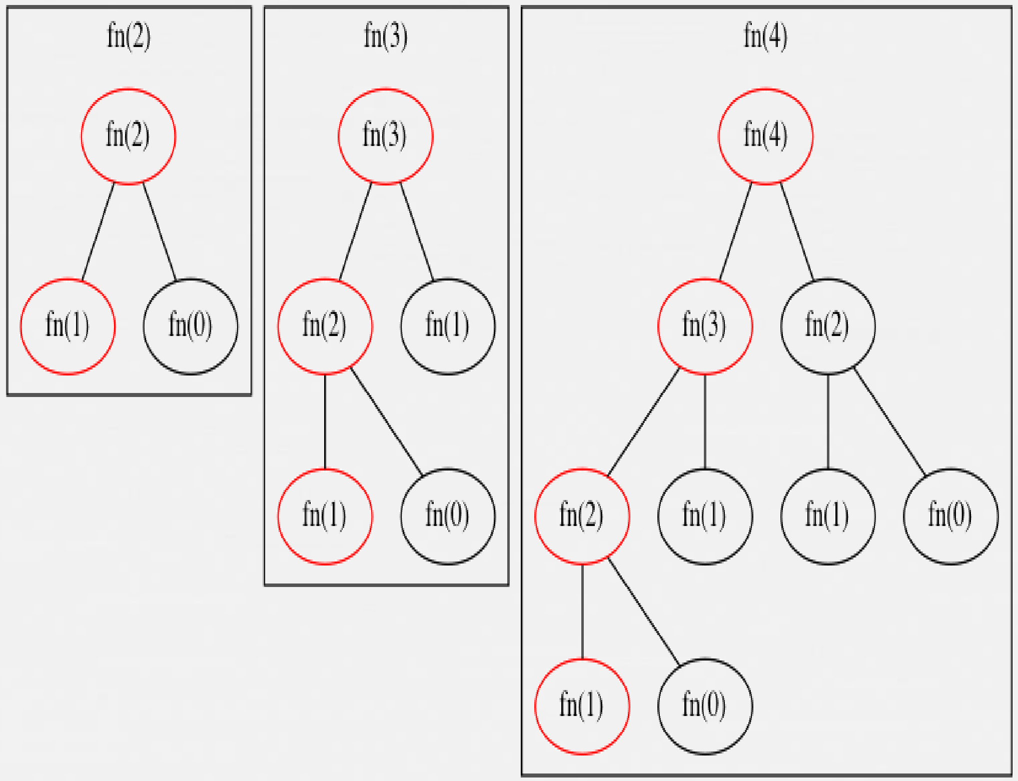
我们来做一些例子:

当n = 2时，有3个函数调用。第一个fn(2)依次调用fn(1)和fn(0)。

对于n = 3，有5个函数调用。首先fn(3)，然后调用fn(2)和fn(1)，以此类推。

对于n = 4，有9个函数调用。首先fn(4)，然后调用fn(3)和fn(2)，以此类推。

因为它是二叉树，我们可以感觉到，每当n增加1，我们最多只能执行2倍的运算。



If you take a look at the generated tree calls, the leftmost nodes go down in descending order: fn(4), fn(3), fn(2), fn(1), which means that the height of the tree (or the number of levels) on the tree will be n.

The total number of calls, in a complete binary tree, is 2^n - 1.

As you can see in fn(4), the tree is not complete. The last level will only have two nodes, fn(1) and fn(0), while a complete tree would have 8 nodes. But still, we can say the runtime would be exponential O(2^n). It won't get any worst because 2^n is the upper bound.

如果查看生成的树调用，最左边的节点按降序排列:fn(4)， fn(3)， fn(2)， fn(1)，这意味着树的高度(或层数)将为n。

在一个完整的二叉树中，调用的总数是2^n - 1。在fn(4)中可以看到，树是不完整的。最后一层只有两个节点，fn(1)和fn(0)，而完整的树有8个节点。但我们仍然可以说运行时间是O(2^n)的指数。情况不会更糟，因为2^n是上界。

Technique

- Informal approach for this class
    - more formal techniques in theory classes
- **How many computations will this program (method, algorithm) perform to get the answer?**
- Many simplifications
    - view algorithms as Java programs
    - **determine by analysis the total number executable statements (computations) in program or method as a function of the amount of data**
    - focus on the *dominant term* in the function

T(N) = $17.5N^3 + 25N^2 + 35N + 251$ ***IS ORDER*** N^3

- 这门课的非正式方法
- 理论课上有更多的形式技巧
- 这个程序(方法、算法)要执行多少次计算才能得到答案?
- 许多简化
- 将算法视为Java程序
- 通过分析确定程序或方法中可执行语句(计算)的总数作为数据量的函数

关注函数中的主导项T(N) = $17.5N^3 + 25N^2 + 35N + 251$ ***IS ORDER*** N^3

Counting Statements
统计状态

```
int x;                 // one statement

x = 12;                // one statement

int y = z * x + 3 % 5 * x / i;          // 1

x++;                   // one statement

boolean p = x < y && y % 2 == 0 || z >= y * x;     // 1

int[] data = new int[100];         // 100

data[50] = x * x + y * y;          // 1
```

Counting Up Statements

- `int result = 0;` 1
- `int i = 0;` 1
- `i < values.length;` N + 1
- `i++` N
- `result += values[i];` N
- `return total;` 1

$T(N) = 3N + 4$
T(N) is the number of executable statements in method total as function of values.length

$T(N) = 3N + 4$
T(N)是方法total中可执行语句的数量，作为values.length的函数

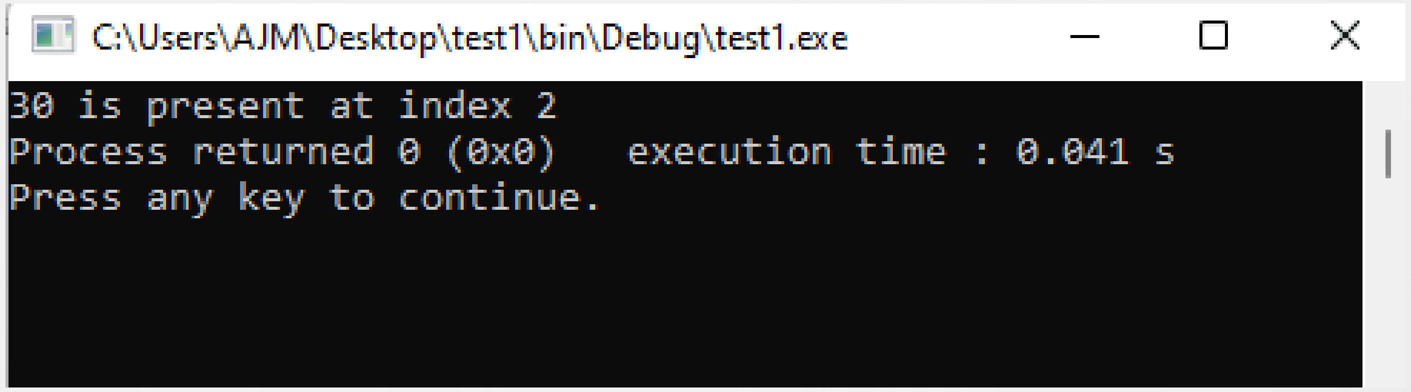
高级算法分析与设计

Advanced Algorithm Analysis and Design

Example　例如

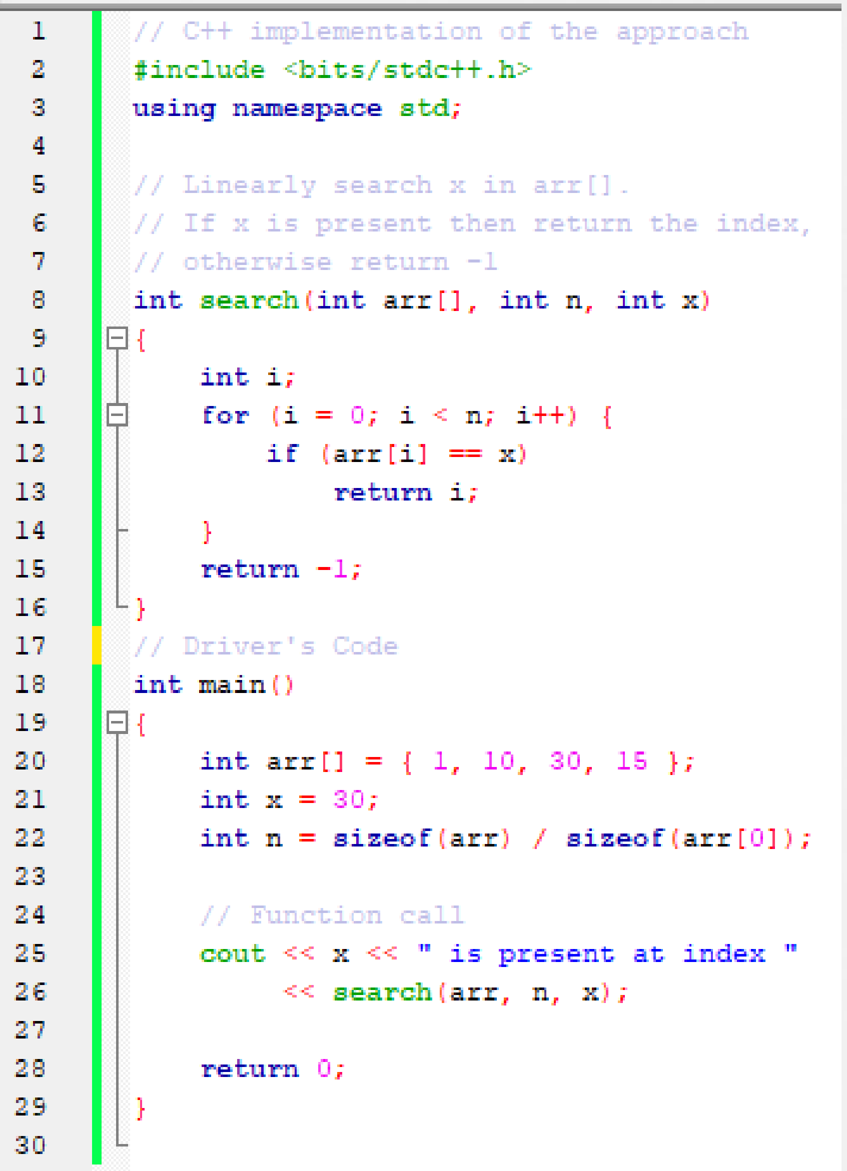
Example 1.
Linear search algorithm: 例1 线性搜索算法:

```
#include <bits/stdc++.h>
using namespace std;

  // Linearly search x in arr[].
// If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] == x)
            return i;
    }
    return -1;
}
// Driver's Code
int main()
{
    int arr[] = { 1, 10, 30, 15 };
    int x = 30;
    int n = sizeof(arr) / sizeof(arr[0]);


    // Function call
    cout << x << " is present at index "
         << search(arr, n, x);
    return 0;
}
```



```
30 is present at index 2
Process returned 0 (0x0)   execution time : 0.041 s
Press any key to continue.
```

Time Complexity Analysis: (In Big-O notation)



```cpp
// C++ implementation of the approach
#include <bits/stdc++.h>
using namespace std;

// Linearly search x in arr[].
// If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] == x)
            return i;
    }
    return -1;
}
// Driver's Code
int main()
{
    int arr[] = { 1, 10, 30, 15 };
    int x = 30;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    cout << x << " is present at index "
         << search(arr, n, x);

    return 0;
}
```

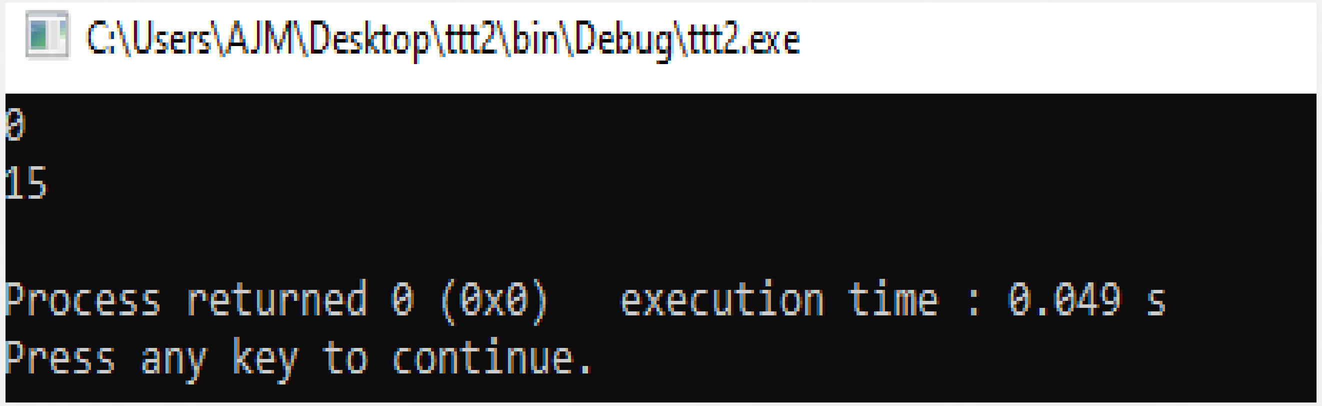
- **Best Case:** O(1), This will take place if the element to be searched is on the first index of the given list. So, the number of comparisons, in this case, is 1.
- **Average Case:** O(n), This will take place if the element to be searched is on the middle index of the given list.
- **Worst Case:** O(n), This will take place if:
  - The element to be searched is on the last index
  - The element to be searched is not present on the list

- 最佳情况:O(1)，如果要搜索的元素位于给定列表的第一个索引上，就会发生这种情况。在这个例子中，比较的次数是1。
- 平均情况:O(n)，如果要搜索的元素位于给定列表的中间索引上，就会发生这种情况。
- 最坏情况:O(n)，如果:
- 要搜索的元素位于最后一个索引上
- 要搜索的元素不在列表中

Example 2.
odd and Even algorithm:

- we will take an array of length (n) and deals with the following cases :
  - **If (n) is even, then our output will be 0**
  - **If (n) is odd then our output will be the sum of the elements of the array.**

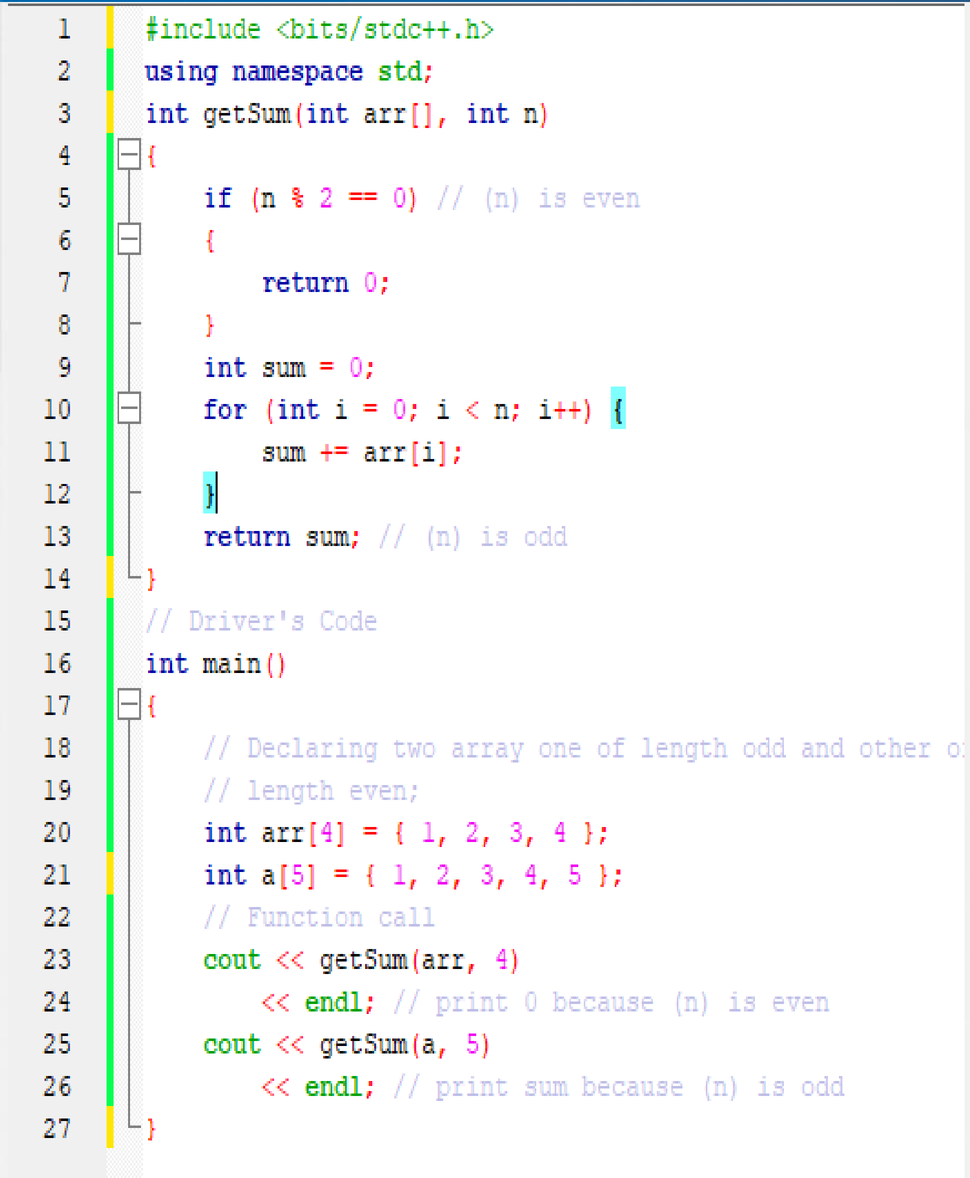
- 我们将取一个长度为(n)的数组，并处理以下情况:
- 如果(n)是偶数，那么输出为0
- 如果(n)是奇数，那么我们的输出将是数组元素的和。


C:\Users\AJM\Desktop\ttt2\bin\Debug\ttt2.exe

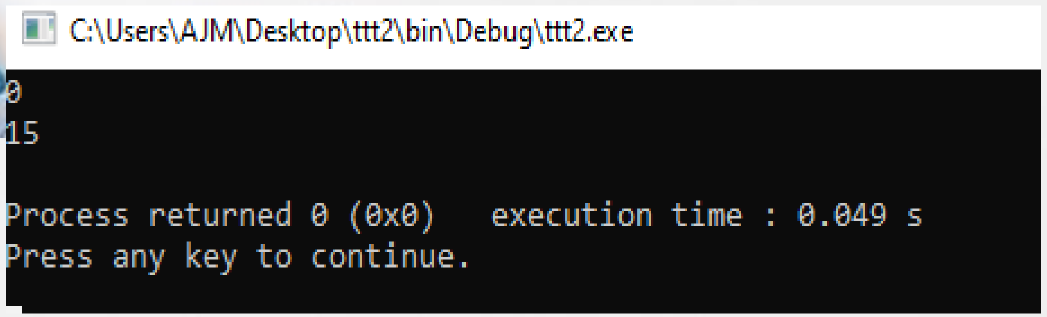
```
0
15

Process returned 0 (0x0)   execution time : 0.049 s
Press any key to continue.
```

```
#include <bits/stdc++.h>
using namespace std;
int getSum(int arr[], int n)
{
        if (n % 2 == 0) // (n) is even
        {
                return 0;
        }
        int sum = 0;
        for (int i = 0; i < n; i++) {
                sum += arr[i];
        }
        return sum; // (n) is odd
}
// Driver's Code
int main()
{
        // Declaring two array one of length odd and other of
        // length even;
        int arr[4] = { 1, 2, 3, 4 };
        int a[5] = { 1, 2, 3, 4, 5 };
        // Function call
        cout << getSum(arr, 4)
                << endl; // print 0 because (n) is even
        cout << getSum(a, 5)
                << endl; // print sum because (n) is odd
}
```

Time Complexity Analysis:



```cpp
#include <bits/stdc++.h>
using namespace std;
int getSum(int arr[], int n)
{
    if (n % 2 == 0) // (n) is even
    {
        return 0;
    }
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum; // (n) is odd
}
// Driver's Code
int main()
{
    // Declaring two array one of length odd and other o
    // length even;
    int arr[4] = { 1, 2, 3, 4 };
    int a[5] = { 1, 2, 3, 4, 5 };
    // Function call
    cout << getSum(arr, 4)
        << endl; // print 0 because (n) is even
    cout << getSum(a, 5)
        << endl; // print sum because (n) is odd
}
```



C:\Users\AJM\Desktop\ttt2\bin\Debug\ttt2.exe

```
0
15

Process returned 0 (0x0)   execution time : 0.049 s
Press any key to continue.
```

- **Best Case:** The order of growth will be **constant** because in the best case we are assuming that (n) is even.
- **Average Case:** In this case, we will assume that even and odd are equally likely, therefore Order of growth will be **linear**
- **Worst Case:** The order of growth will be **linear** because in this case, we are assuming that (n) is always odd.

- 最佳情况:增长的顺序将是恒定的，因为在最佳情况下，我们假设(n)是偶数。
- 平均情况:在这种情况下，我们将假设偶数和奇数是等可能的，因此增长顺序将是线性的
- 最坏情况:增长的顺序将是线性的，因为在这种情况下，我们假设(n)总是奇数。

Example 3. 例3 两正相乘算法: multiplying two positive algorithm:

(当乘数下对应的数字为奇数时，选择第2列中的数字)

Example -乘以两个正整数A和B
– multiplying two positive integers A and B

For example: 45*19
Usually:

$$\begin{array}{r l} 45 & \\ 19 & (x \\ \hline 405 & \\ 45 & (+ \\ \hline 855 & \end{array}$$

Multiplier 乘数	Multiplicand 被乘数	Result 结果
(A/2)	(B*2)	(pick numbers in column 2 when the corresponding number under the multiplier is odd)
45	19	19
22	38	
11	76	76
5	152	152
2	304	
1	608	608 (+
		855

A different algorithm:
不同的算法:

- An instance of a problem is a specific assignment of values to the parameters.
- This algorithm can be used for multiplying any two positive integers, we say that (45, 19) is an **instance** of this problem. Most problems have infinite collection of instances.
- It’s ok to define the **domain** (i.e. the set of instances) to be considered, and the algorithm should work for all instances in that domain.
- Although the above algorithm will not work if the first operand is negative, this does not invalidate the algorithm since (-45, 19) is not an instance of the problem being considered.
- 问题的实例是对参数的特定赋值。
- 这个算法可以用于任意两个正整数的乘法，我们说(45,19)是这个问题的一个实例。大多数问题都有无限的实例集合。
- 定义要考虑的域(即实例集)是可以的，算法应该适用于该域中的所有实例。
- 尽管如果第一个操作数为负，上述算法将无法工作，但这并不会使算法无效，因为(- 45,19)不是正在考虑的问题的实例。

Order:

Usually we use the **<u>frequency count</u>** to compare algorithms.

Consider the following 3 programs:

```
(a)
x ← x + y

(b)
for i ← 1 to n do
    x ← x + y
end

(c)
for i ← 1 to n do
    for j ← 1 to n do
        x ← x + y
    end
end
```

The frequency count of stmt x ← x + y is 1, n, n^2.

No matter which machine we use to run these programs, we know that the execution time of (b) is n times the execution time of (a).

通常我们使用频率计数来比较算法。

考虑以下3个程序:

stmt x←x + y的频率计数为1,n, n2。

无论我们用哪台机器运行这些程序，我们都知道(b)的执行时间是(a)的执行时间的n倍。

Now you calculate the example:

例子:

- *Is 17 n² – 5 = O(n²)?*

$$\because \quad 17n^2 - 5 \le 17n^2 \qquad \forall n \ge 1$$
$$(c = 17, n_0 = 1)$$
$$\therefore \quad 17n^2 - 5 = O(n^2)$$

- *Is 35 n³ + 100 = O(n³)?*

$$\because \quad 35n^3 + 100 \le 36n^3 \qquad \forall n \ge 5$$
$$(c = 36, n_0 = 5)$$
$$\therefore \quad 35n^3 + 100 = O(n^3)$$

- *Is 6 • 2ⁿ + n² = O(2ⁿ)?*

$$\because \quad 6 \cdot 2^n + n^2 \le 7 \cdot 2^n \qquad \forall n \ge 5$$
$$(c = 7, n_0 = 5)$$
$$\therefore \quad 6 \cdot 2^n + n^2 = O(2^n)$$

Now you calculate example

```
for(int i = 0; i < 5; i++)
    cout << i;
```

$O(1)$:

```
for(int i = 0; i < n; i++)
    cout << i;
```

$O(n)$:

```
int i = n;
while(i >= 1)
{
    i /= 2;
    cout << i;
}
```

$O(\log_2 n)$:

```
for(int i = 0; i < n; i++)
{
    int j = n;
    while(j >= 1)
    {
        j /= 2;
        cout << j;
    }
}
```

$O(n \log_2 n)$:

$O(n^2)$:

```
for(int i = 0; i < n; i++)      // O(n)
    for(int j = 2; j < n; j++)  // O(n)
        cout << i * j;          // O(n) * O(n) = O(n^2)
```

江西理工大學

JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Now you calculate example

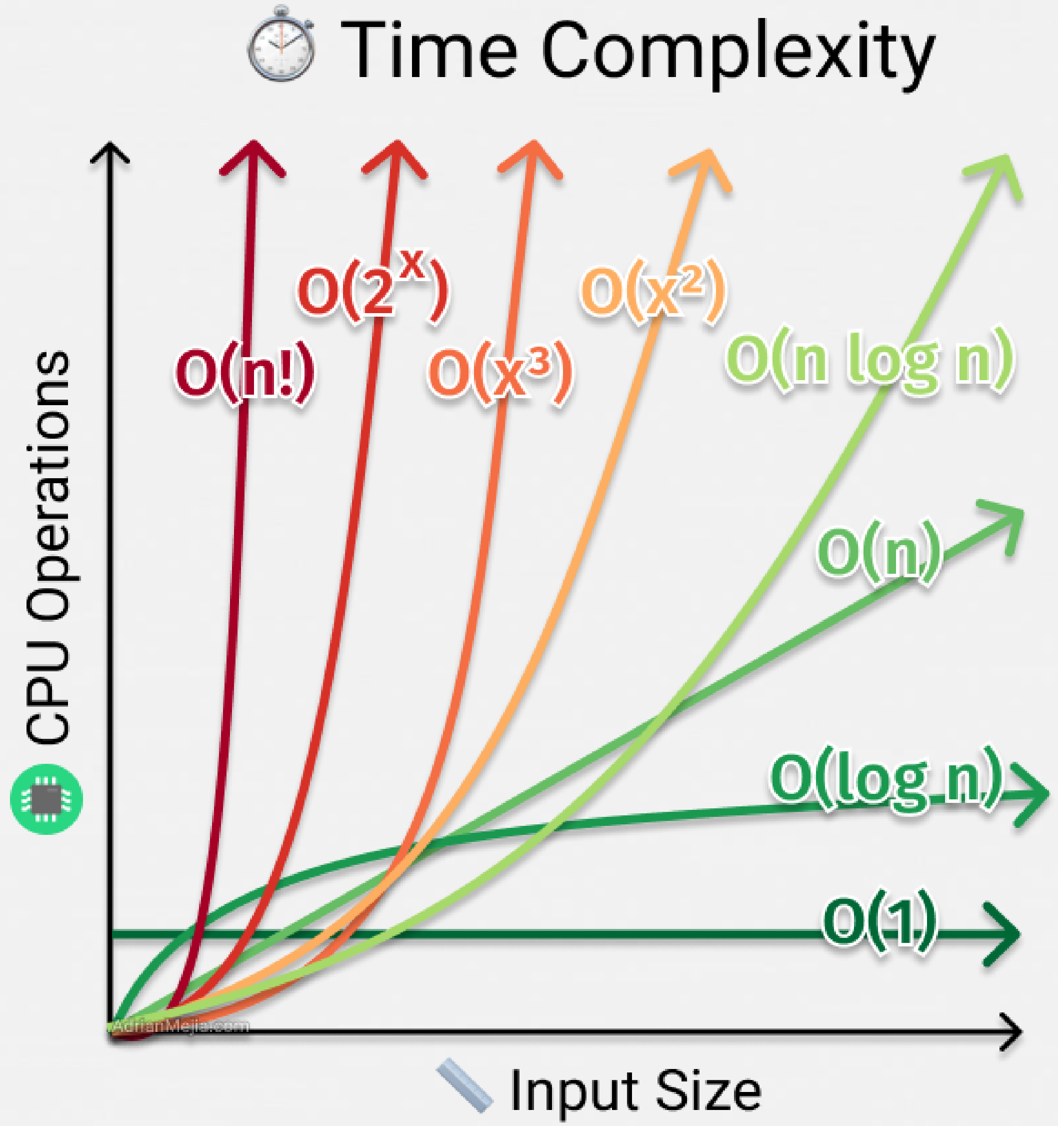
$$f(n) = n^m + n^{m-1} + \ldots + n^2 + n + c \Rightarrow f(n) = O(n^m)$$

$$\Downarrow$$

$$f(n) = 9n^2 - 4n + 2 \Rightarrow O(n^2)$$

Complexity classes

复杂的类别



Reference

- https://afteracademy.com/blog/time-and-space-complexity-analysis-of-algorithm

江西理工大学

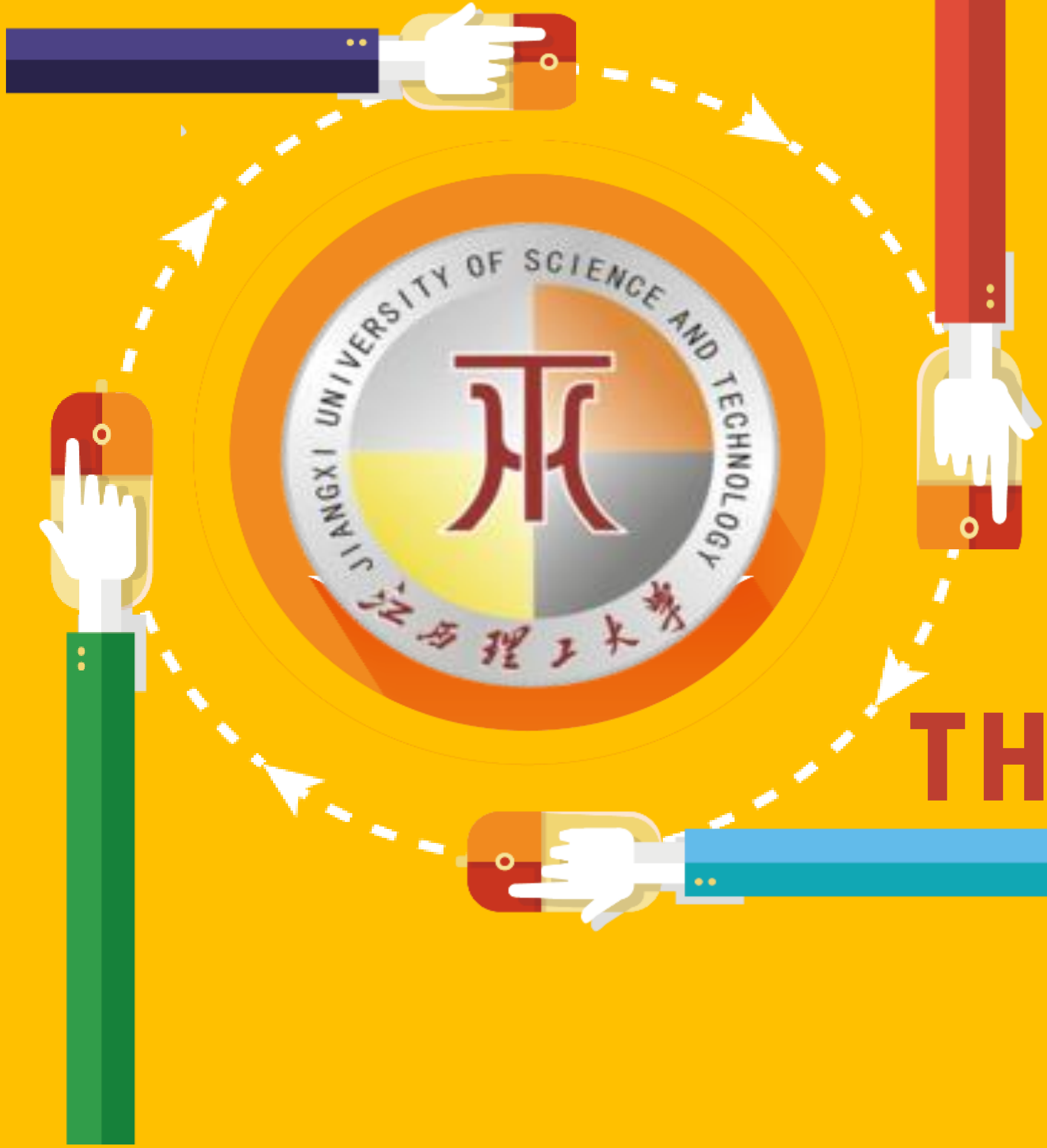
Jiangxi University of Science and Technology

信息工程学院

School of information engineering

高级算法分析与设计

Advanced Algorithm Analysis and Design



THANK YOU 谢谢你

"The beauty of research is that you never know where it's going to lead."

RICHARD ROBERTS
Nobel Prize in Physiology or Medicine 1993

"BE HUMBLE. BE HUNGRY. AND ALWAYS BE THE HARDEST WORKER IN THE ROOM."