算法的时间复杂度和空间复杂度分别是什么?

时间复杂度用于描述算法执行时间与输入规模之间的关系。它通常表示为大 O 符号，例如 O(n)、O(n^2)、O(log n) 等，其中 n 是输入数据的规模。更加关注当 n 趋于无穷时，算法整体的运行效率。
空间复杂度与时间复杂度相对，描述算法执行过程中所需的存储空间与输入规模之间的关系。它同样使用大 O 符号表示，例如 O(n)、O(n^2)、O(1) 等。
在算法当中，往往会有算法使用更多的空间复杂度，以达到更少的时间复杂度。

算法是什么？有什么作用?

算法（Algorithm）是解决特定问题的一系列定义清晰的计算步骤，它可以用来完成一个计算任务或者解决一个计算问题。他可以用于将抽象的问题简单化，从而化繁为简；同时也能提供某类特定情景的合乎逻辑的解决方案。

算法分析的方法是多种多样的。常用的评判算法效率的方法有哪些？请举例。

首先可以使用**时间复杂度**。正如上文所说，它衡量了算法执行时间随着输入规模增长的变化趋势。O 代表运行时间的上界，即最坏情况。时间复杂度越低，通常意味着算法的执行效率越高。
其次可以使用 **RAM 随机访问机**测试实际运行时间。通过给予完全随机的输入进行时间的观测。这通常需要多次测试以获得平均值。该方法考虑硬件性能、操作系统、编译器优化等因素。
最后可以使用公式，如**主方法**，**递推法**进行**渐近分析**。这是一种理论估算方法，它使用大 O 表示法来描述算法的时间复杂度和空间复杂度。它关注的是算法在最坏情况下的性能，而不是具体实现的细节。

如何去评判一个算法的复杂度?

使用渐近记号。通常使用的最多的是类似 O(n)、O(n^2)、O(1)等，代表输入之后，假设输入的 n 规模非常大，考虑其运行的时间上界。以下列举了一些从小到大排列的时间复杂度：O(log n)、O(n)、O(n log n)、O(n^2)、O(2^n)

算法在一般情况下被认为有五个基本属性。它们分别是什么？请简要说明

他们分别是：输入，输出，确定性，有限性，可行性。
输入：算法需要有明确的输入，这些输入是算法处理的基础。输入可以是数据、信号或其他形式的信息。
输出：算法执行完成后应该有明确的输出，输出是算法处理结果的展示。输出应该能解决提出的问题或满足特定需求。
确定性：算法中的每一步操作都应该是明确的，对于相同的输入，算法应该在每次执行时都产生相同的输出，不存在随机性。
有限性：算法必须在有限的步骤之后结束，不能包含无限循环。这意味着算法在执行过程中，每一步都应该在有限时间内完成。

可行性：算法中的操作都应该是可执行的，即在当前的技术和资源条件下，算法的每一步都能够被准确无误地执行。

1，6，7，8，题代码如下：

```python
def isprime(n):
    flag=True
    if n == 2:
        flag=True
    elif n < 2:
        flag=False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            flag=False
    if flag:
        print("is prime")
    else:
        print("not prime")
    return

isprime(int(input("输入要检测的正整数："))
```

```python
######################################################################
# 实现选择排序,并尝试对不同长度的随机数组排序,并计算出程序执行时间
import timeit

def selection_sort(arr):

    for i in range(len(arr)):
        min_index = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

    print(arr)

arr=list(map(int,input().split(" ")))
selection_sort(arr)

elapsed_time = timeit.timeit(stmt="selection_sort(arr)", setup="from __main__ import selection_sort, arr"
print(f"运行时间：{elapsed_time}秒")
```

```python
#汉诺塔

def tower(n, a, b, c):
    if n==1:
        print(f"从{a}移动到{b}")
    else:
        tower(n-1,a,c,b)
        print(f"从{a}移动到{b}")
        tower(n-1,c,b,a)
tower(3,'A','B','C')
```

8

```python
##########################################################################
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def arrayToBST(nums):
    if not nums:
        return None

    mid = len(nums) // 2
    root = TreeNode(nums[mid])

    root.left = arrayToBST(nums[:mid])
    root.right = arrayToBST(nums[mid+1:])

    return root

def inorderTraversal(root, result=None):
    if result is None:
        result = []
    if root:
        inorderTraversal(root.left, result)
        result.append(root.val)
        inorderTraversal(root.right, result)
    return result

def left_depth_first_traversal_iterative(root):
    if root is None:
        return

    stack = [root]
    while stack:
        node = stack.pop()
        print(node.val)  # 访问当前节点

        # 先左后右入栈，确保左子树先处理
        if node.left:
            stack.append(node.left)
        if node.right:
            stack.append(node.right)
```

```
# 示例数组
nums = [1,5,2,8,6,9,1]

# 将数组转换为BST
bst_root = arrayToBST(nums)


leftorder = left_depth_first_traversal_iterative(bst_root)#左序遍历
inorder=inorderTraversal(bst_root)
print(inorder)#排序后数组
print(leftorder)
#########################################################################
```