3 算法的时间复杂度和空间复杂度分别是什么 ？

时间复杂度是关于输入大小 n 的函数,关注的是算法执行时间随输入规模增长而增长的趋势,一般用大 O 表示

空间复杂度是衡量算法执行过程中所需存储空间的量度。它同样使用大 O 表示法来描述，但关注的是算法运行过程中所需的最大存储空间

4 算法是什么 ？有什么作用 ？

算法是操作用数据模型抽象,数据结构等形式表示的数据,从而获取解决方案的技术

算法能够:

提高计算效率：通过设计高效的算法，可以显著减少计算所需的时间和资源，提高计算效率。

解决复杂问题:算法能够将复杂的问题分解为一系列简单的步骤,使得问题变得可解决。

5 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 ？请举例说明 ？

时间复杂度分析：通过分析算法执行过程中基本操作（如比较、赋值、算术运算等）的次数，来评估算法执行时间随输入规模增长而变化的趋势。常用大 O 表示法来描述时间复杂度，如 O(n)、O(n^2)、O(log n)等。

空间复杂度分析：评估算法执行过程中所需额外存储空间的大小。同样使用大 O 表示法，如 O(1)、O(n)等。

6 如何去评判一个算法的复杂度 ？

时间复杂度:

估算这些基本操作在算法执行过程中被执行的次数，这通常与输入数据的规模（通常用 n 表示）有关。

使用大 O 表示法来描述这种增长趋势，忽略常数因子和低阶项。

空间复杂度:

确定算法中所有临时占用存储空间的总量，包括局部变量、辅助数据结构等。

估算这个总量随着输入数据规模 n 的变化趋势。

7 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 ？

输入：算法具有零个或多个输入。这些输入是算法开始执行前所必需的，它们描述了算法需要处理的数据或问题的初始状态。

输出：算法至少具有一个或多个输出。输出是算法执行结束后产生的结果，反映了算法对输入数据的处理结果或问题的答案。

有限性：算法在执行有限的步骤后必须自动结束，不能出现无限循环。这意味着算法的执行时间是有限的，并且每个步骤都必须在可接受的时间内完成。

确定性：算法的每个步骤都必须有明确的定义，即算法的每一步都应该是准确无误的，不能存在二义性。这样才能保证算法在相同的输入下总能产生相同的输出。

有效性：算法的每一步都必须是可行的，即算法中的每个操作都可以通过已经实现的基本运

算在有限的时间内完成。这要求算法的设计必须考虑到实际执行时的可行性和可操作性。

1

```python
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True

print(is_prime(7))
```

6

```python
def select_sort(num):
    for i in range(len(num)):
        min_index = i
        for j in range(i + 1, len(num)):
            if num[j] < num[min_index]:
                min_index = j
        num[i], num[min_index] = num[min_index], num[i]
    return num

def generate_random_list(size):
    return [random.randint(1, 1000) for _ in range(size)]


start_time = time.time()
random_list = generate_random_list(1024)
select_sort(random_list)
print("sorted list:", random_list)
end_time = time.time()
duration = end_time - start_time
print("it cost {} seconds".format(duration))
```

7

```python
def hannuota(num,start,end,mid):
    if num == 1:
        print("{}-->{}".format(start,end))
    else:
        hannuota(num-1,start,mid,end)
        print("{}-->{}".format(start,end))
        hannuota(num-1,mid,end,start)

print("汉诺塔问题：")
hannuota(3,'A','C','B')
```

有些底层的问题算了很多遍，比如说汉诺塔三层，我们需要增添一个记忆序列，再次碰到这种东西的时候，我们就可以直接输出，而不是进行计算。

8

```python
import random

class TreeNode:   ···

def insert(root, key):   ···

def inorder_traversal(root):   ···

def generate_random_list(size): ···

keys = generate_random_list(10)
root = None
for key in keys:
    root = insert(root, key)

sorted_keys = inorder_traversal(root)
print(sorted_keys)
```