3. 时间复杂度：定性描述算法运行时间的函数。
空间复杂度：对一个算法在运行过程中临时占用存储空间大小的量度。
4. 算法：解决问题的一个进程，可机械执行的一系列步骤精确而明确的规范。
算法的作用：解决问题的计算方法。
5. 评价算法效率的方法：时间复杂度、空间复杂度、实际运行时间、算法的适用场景等多个方面进行综合评估。
6. 评价一个算法的复杂度：从算法的时间复杂度和空间复杂度去评价。
7. 算法的五种基本属性：有限性：执行步骤有限
确定性：操作明确含义清晰
输入：有多个或 0 个输入
输出：有输出反馈
可行性：所有操作可以实现

实践题：1.

```
a=int(input())
count=0
for i in range(2,a//2):
    if(a%i==0):
        count+=1
        print("a is not prime")
        break
if(count==0):
    print("a is prime")
```

6.

```
#选择排序
def xuanze(a):
    p=[]
    while(len(a)!=0):
        j=min(a)
        p.append(j)
        a.remove(j)
    return p
li=input().split()
a=list(map(lambda x:int(x),li))
p=xuanze(a)
print(p)
```

时间复杂度：n**2

7.

```
def han_no_ta(n, i, j, k, moves):
    if n == 1:
        moves.append((i, k))
    else:
        han_no_ta(n - 1, i, k, j, moves)
        moves.append((i, k))
```

```
        han_no_ta(n - 1, j, i, k, moves)

def print_moves(moves):
    for move in moves:
        print(f"{move[0]} -> {move[1]}")
moves = []
han_no_ta(6, "1", "2", "3", moves)
print_moves(moves)
```

优化，迭代的方法：

```
def han_no_ta_iterative(n, i, j, k, moves):
    # 初始化栈
    stack = []

    def push(n, src, dest, aux):
        if n > 0:
            stack.append((n, src, dest, aux))
    def pop():
        return stack.pop()
    # 推送初始状态
    push(n, i, k, j)
    while stack:
        n, src, dest, aux = pop()
        if n == 1:
            moves.append((src, dest))
        else:
            push(n - 1, aux, src, dest)
            moves.append((src, dest))
            push(n - 1, src, dest, aux)
def print_moves(moves):
    for move in moves:
        print(f"{move[0]} -> {move[1]}")
moves = []
han_no_ta_iterative(6, "1", "2", "3", moves)
print_moves(moves)
```

8.

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

def insert_into_bst(root, value):
    if root is None:
```

```
        return TreeNode(value)
    else:
        if value <= root.val:
            root.left = insert_into_bst(root.left, value)
        else:
            root.right = insert_into_bst(root.right, value)
    return root
def preorder_traversal(root):
    result = []
    if root:
        result.append(root.val)
        result += preorder_traversal(root.left)
        result += preorder_traversal(root.right)
    return result
li = input().split()
a = list(map(lambda x: int(x), li))
root = None
for num in a:
    root = insert_into_bst(root, num)
sorted_array = preorder_traversal(root)
print( sorted_array)
```