编程题：

1.带字母的 ()(()) 字符串，对每个()内的内容进行反转。力扣 1190. 反转每对括号间的子串

遍历过程中遇到正括号无需考虑，在遇到反括号的时候将栈中的字符一直弹出，直到遇到正括号，因为栈是后进先出，相当于翻转了一遍，重新压回去即可。

也是因为栈是后进先出，遍历完一遍最后里面放的字符顺序是反过来的，所以用另外一个栈辅助一下，再把字符顺序反转一遍，就是答案了。

O(n)

```
public class Solution
{
    public string ReverseParentheses(string s)
    {
        Stack<char> stack = new Stack<char>();
        for(int i = 0; i < s.Length; i++)
        {
            if(s[i] != ')')
            {
                stack.Push(s[i]);
            }
            else
            {
                StringBuilder sb = new StringBuilder();
                while(stack.Peek() != '(')
                {
                    sb.Append(stack.Pop());
                }
                stack.Pop();
                for (int j = 0; j < sb.Length; j++)
                {
                    stack.Push(sb[j]);
                }
            }
        }
        StringBuilder result = new StringBuilder();
        Stack<char> stack2 = new Stack<char>();

        while(stack.Count > 0)
        {
            stack2.Push(stack.Pop());
        }
        while (stack2.Count > 0)
        {
            result.Append(stack2.Pop());
        }

        return result.ToString();
```

```
    }
}
```

2.给定一个仅由字符 ‘a’, ‘b’, ‘c’ 组成的字符串，找到其中最长的子串，使得该子串中 ‘a’, ‘b’, ‘c’ 的数量相同。

> 时间复杂度： O(n)
>
> ### 1. 状态表示
>
> - 用 (a-b, a-c) 这个二元组来表示当前位置的“状态”。
> - 如果两个位置的 (a-b, a-c) 状态相同，说明这两个位置之间的子串中 'a'、'b'、'c' 的数量是相等的。
>
> ### 2. 哈希表记录状态
>
> - 用一个字典记录每种状态第一次出现的位置。
> - 初始时，状态 (0, 0) 在位置 -1
>
> ### 3. 遍历字符串
>
> - 更新 a、b、c 的计数。
> - 计算当前的状态 (a-b, a-c)。
> - 如果这个状态之前出现过，说明从上次出现的位置到当前位置的子串，'a'、'b'、'c' 的数量相等。
> - 用当前位置减去上次出现的位置，得到子串长度，更新最大长度。
> - 如果这个状态没出现过，就把当前位置记录到字典里。

```
public class Solution
{
    public int LongestEqualABC(string s)
    {
        int n = s.Length;
        int maxLen = 0;

        // 记录每种状态第一次出现的位置
        var dict = new Dictionary<(int, int), int>();
        dict[(0, 0)] = -1; // 初始状态

        int a = 0, b = 0, c = 0;
        for (int i = 0; i < n; i++)
        {
            // 统计 a、b、c 的数量
            if (s[i] == 'a') a++;
            else if (s[i] == 'b') b++;
            else if (s[i] == 'c') c++;

            // 当前状态
            var key = (a - b, a - c);
```

```
            // 如果之前出现过这个状态
            if (dict.ContainsKey(key))
            {
                // 更新最大长度
                maxLen = Math.Max(maxLen, i - dict[key]);
            }
            else
            {
                // 记录第一次出现的位置
                dict[key] = i;
            }
        }
        return maxLen;
    }
}
```

3.整形数组，有正数负数，找和值最大的区间，时间要求O(n)

暴力：

```
public class Solution
{
    public int MaxSubArray(int[] nums)
    {
        int max = nums[0];
        int sum = nums[0];

        for(int i = 1; i < nums.Length; ++i)
        {
            if(sum < 0) //防止了负数累加导致的结果变小
            {
                sum = 0;
            }
            sum = sum + nums[i];
            max = Math.Max(sum, max);
        }

        return max;
    }
}
```

DP：

```
public class Solution
{
    public int MaxSubArray(int[] nums)
    {
        int[] dp = new int[nums.Length + 1];
        dp[0] = nums[0];
```

```
        int max = nums[0];
        for (int i = 1; i < nums.Length; ++i)
        {
            dp[i] = Math.Max(dp[i - 1] + nums[i], nums[i]);
            max = Math.Max(dp[i], max);
        }
        return max;
    }
}
```

4.字符串只包含 0 1，要找出最长连续字串并且0，1个数相同。时间复杂度O(n)

```
public class Solution {
    public int FindMaxLength(int[] nums) {
        int n = nums.Length;
        int maxLen = 0;

        // 记录每种状态第一次出现的位置
        var dict = new Dictionary<(int, int), int>();
        dict[(0, 0)] = -1; // 初始状态

        int ones = 0, zeros = 0;
        for (int i = 0; i < n; i++) {
            // 统计 0 和 1 的数量
            if (nums[i] == 1) ones++;
            else zeros++;

            // 当前状态
            var key = (ones - zeros, zeros);

            // 如果之前出现过这个状态
            if (dict.ContainsKey(key)) {
                // 更新最大长度
                maxLen = Math.Max(maxLen, i - dict[key]);
            } else {
                // 记录第一次出现的位置
                dict[key] = i;
            }
        }
        return maxLen;
    }
}
```

一、自我介绍

1.介绍多态，C#虚函数实现原理、虚函数表、虚析构函数（主要就是考查内存布局）、析构函数和虚函数有啥区别，构造函数能否为虚函数？为什么？

1. 多态

多态是面向对象编程的三大特性(封装继承多态)之一，指的是**同一个接口，使用不同的实例而表现出不同的行为。**

- **编译时多态**（如方法重载、运算符重载）
- **运行时多态**（如虚函数、接口）

---

2. C#虚函数实现原理

2.1 虚函数

在C#中，父类的方法如果被 `virtual` 修饰，子类可以用 `override` 重写。

2.2 虚函数表

每个包含虚函数的类都有一张虚函数表，表中存放着该类虚函数的实际地址，程序在运行时可以找到正确的虚函数实现。

3. 析构函数 `~ClassName()`

作用：在GC确定对象不可达、准备回收时，释放这些非托管资源，防止资源泄漏。

- 析构函数没有访问修饰符、没有参数、不能被手动调用、不能重载。
- 只能在类中定义，不能用于结构体。

---

4. 析构函数和虚函数的区别

	析构函数	**虚函数**
作用	释放资源	支持多态
关键字	`~ClassName()`	`virtual` / `override`
是否可重写	只能重写 `Object.Finalize()`	可以被子类重写
是否自动虚拟	是（C#中自动虚）	需显式声明 `virtual`
调用时机	垃圾回收时	方法被调用时

---

5. 构造函数能否为虚函数？为什么?

构造函数不能是虚函数。

原因:

- 构造函数的作用是初始化对象的内存和状态，在对象创建时调用。

- 虚函数依赖于虚函数表，而虚函数表是在构造函数执行期间才初始化的。在构造函数执行时，对象还未完全构造好，虚表还未建立，无法实现多态。
- 没有意义：构造函数是用来创建对象的，不需要多态。

2.C# 的接口多继承是怎么实现的?

接口的分派是通过接口表实现的，每个接口有自己的方法表。

当类实现多个接口时，编译器会在类的Type信息中记录所有接口的方法，并在运行时通过接口表进行方法分派。

3.C#类型转换操作符，引用值会变化吗?

Parse、ToString、TryParse、Convert 类，is、as

引用类型转换时，引用的“地址”不会变化

4.在基类的析构方法调用虚函数，会呈现多态吗

- 在基类析构函数中调用虚函数，不会发生多态，只会调用基类的实现。
- 这是为了防止访问已销毁的派生类成员，保证程序安全。
- 析构函数的调用顺序：子类自身的析构函数、成员对象的析构函数、基类的析构函数，这个顺序与构造函数相反

5.哈希表

- 通过哈希函数将键映射到数组的某个位置（桶）。
- 如果发生哈希冲突（不同的键映射到同一个桶），通过链表或开放寻址法解决。
- 查找、插入、删除操作的平均时间复杂度为 O(1)。

Hashtable（非泛型）

键和值都是 object 类型

底层是数组+链表的结构，通过哈希函数定位桶，通过链表解决冲突。

SortedDictionary<TKey, TValue>

- **底层实现**：红黑树（自平衡二叉搜索树）
- **有序性**：有序，按 key 排序
- **增删改查**：O(log n)（查找、插入、删除都是对数级别）
- **适用场景**：需要有序字典，且频繁增删改查

Dictionary<TKey, TValue>

- **底层实现**：哈希表
- **有序性**：无序
- **增删改查**：平均 O(1)，最坏 O(n)（极少数情况下哈希冲突严重时）
- **适用场景**：不关心顺序，追求极致的查找/插入/删除性能

6.介绍一下红黑树

红黑树的五大性质:

- 每个节点要么是红色，要么是黑色。
- 根节点是黑色。（根黑）
- 每个叶子节点是黑色。（叶黑）
- 红色节点的子节点必须是黑色。（不红红，不能有两个连续的红色节点）
- 从任一节点到其每个叶子节点的所有路径都包含相同数目的黑色节点。（黑路同，黑色节点数相同）

红黑树的作用：

保证最长路径不会超过最短路径的2倍，因此树的高度始终是 O(log n)。这样可以保证查找、插入、删除操作的时间复杂度都是 O(log n)。

总结口诀
根叶黑：根节点和叶子节点都是黑色
不红红：红色节点不能连续
黑路同：任意一条路径黑色节点数相同

7.内存泄漏是什么，如何避免、检查、解决（例如析构函数调用前发生程序异常，造成内存泄漏等等，代码逻辑考虑准确）

C# 用 GC 自动管理内存，大部分情况下能自动回收不再使用的对象，极大减少了内存泄漏的风险。
常见的内存泄漏场景主要是“**托管内存泄漏**”，对象虽然不再需要，但依然被引用，GC 无法回收:

- 事件未注销：对象+=订阅了事件，但未及时-=注销，导致对象一直被事件源引用，无法被GC回收。
- 静态变量持有引用：静态变量生命周期贯穿整个程序，若持有对象引用，对象不会被回收。
- 集合类未清理：如 List、Dictionary 等集合中存放了大量对象，未及时清理。

8.单链表可以使用快速排序吗?

可以，每一趟把大的数往右放，维护最右边节点就行

时间复杂度平均 O(n log n)，最坏 $O(n^2)$每次分区选到的基准值总是极值。

1. 选择一个基准节点（pivot），通常选链表头节点。
2. 遍历链表，将小于pivot的节点放到左链表，大于等于pivot的节点放到右链表。
3. 递归地对左右链表分别排序。
4. 将排好序的左链表、pivot节点、右链表连接起来。

```
public class ListNode
{
    public int val;
    public ListNode next;
    public ListNode(int x) { val = x; }
}
public ListNode QuickSort(ListNode head)
{
    if (head == null || head.next == null)
        return head;

    // 选第一个节点为pivot
    ListNode pivot = head;
    ListNode leftDummy = new ListNode(0), leftTail = leftDummy;
    ListNode rightDummy = new ListNode(0), rightTail = rightDummy;
    ListNode curr = head.next;

    // 分割链表
    while (curr != null)
    {
        if (curr.val < pivot.val)
        {
            leftTail.next = curr;
            leftTail = leftTail.next;
        }
        else
        {
            rightTail.next = curr;
            rightTail = rightTail.next;
        }
        curr = curr.next;
    }
    leftTail.next = null;
    rightTail.next = null;

    // 递归排序左右链表
    ListNode leftSorted = QuickSort(leftDummy.next);
    ListNode rightSorted = QuickSort(rightDummy.next);

    // 拼接：左链表 + pivot + 右链表
    return Concat(leftSorted, pivot, rightSorted);
}
private ListNode Concat(ListNode left, ListNode pivot, ListNode right)
{
    ListNode head = left ?? pivot;
    ListNode tail = head;

    // 找到左链表的尾部
    if (left != null)
    {
        while (tail.next != null)
            tail = tail.next;
```

```
            tail.next = pivot;
        }
        pivot.next = right;
        return head;
    }
```

9.如何判断单链表中是否存在环？如果存在环，如何找入口节点。（快慢指针）

单链表中是否存在环：

- 定义两个指针：slow（每次走一步），fast（每次走两步）。
- 无环：fast会先到达链表末尾（指向null）。
- 有环：相遇

```
public bool HasCycle(ListNode head)
{
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null)
    {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast)
            return true; // 有环
    }
    return false; // 无环
}
```

如何找环的入口节点：

1. 当slow和fast第一次相遇时，让fast回到链表头，slow留在相遇点。
2. 两个指针每次走一步，再次相遇的节点就是环的入口。

```
public ListNode DetectCycle(ListNode head)
{
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null)
    {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast)
        {
            ListNode ptr = head;
            while (ptr != slow)
            {
                ptr = ptr.next;
                slow = slow.next;
            }
            return ptr; // 环的入口
```

```
            }
        }
        return null; // 无环
    }
```

10.接口（interface）

- 只包含方法签名、属性、事件、索引器等的声明，没有任何实现
- 不能包含字段（变量）。
- 不能包含构造函数、析构函数。
- 类或结构体实现接口时，必须实现接口中所有成员。
- 支持多继承（一个类可以实现多个接口）。