1 休闲游戏排行榜

代码实现

```
using System;
using System.Collections.Generic;
using System.Linq;

public class LeaderboardSystem
{
    public static List<int> GetTopScores(int[] scores, int m)
    {
        // 处理边界情况
        if (scores == null || scores.Length == 0 || m <= 0)
```

```
        {
            return new List<int>();
        }

        // 确保 m 不超过数组长度
        m = Math.Min(m, scores.Length);

        // 使用 LINQ 获取前 m 个最高分数并按降序排序
        return scores.OrderByDescending(score => score).Take(m).ToList();
    }
}
```

单元测试

```
using NUnit.Framework;
using System.Collections.Generic;

public class LeaderboardSystemTests
{
    [Test]
    public void TestGetTopScores()
    {
        // 测试用例 1
        int[] scores1 = { 100, 50, 75, 80, 65 };
        int m1 = 3;
        List<int> expected1 = new List<int> { 100, 80, 75 };
        Assert.AreEqual(expected1, LeaderboardSystem.GetTopScores(scores1, m1));

        // 测试用例 2
        int[] scores2 = { 90, 85, 95 };
        int m2 = 5; // m 大于数组长度
        List<int> expected2 = new List<int> { 95, 90, 85 };
        Assert.AreEqual(expected2, LeaderboardSystem.GetTopScores(scores2, m2));

        // 测试用例 3
        int[] scores3 = { };
        int m3 = 2; // 空数组
        List<int> expected3 = new List<int> { };
        Assert.AreEqual(expected3, LeaderboardSystem.GetTopScores(scores3, m3));

        // 测试用例 4
        int[] scores4 = { 100 };
        int m4 = 1; // 仅一个分数
        List<int> expected4 = new List<int> { 100 };
        Assert.AreEqual(expected4, LeaderboardSystem.GetTopScores(scores4, m4));

        // 测试用例 5
        int[] scores5 = { 10, 20, 30, 40, 50 };
        int m5 = 0; // m 为 0
        List<int> expected5 = new List<int> { };
        Assert.AreEqual(expected5, LeaderboardSystem.GetTopScores(scores5, m5));
```

```
    }
}
```

分析

时间复杂度:

排序操作的时间复杂度是 O(nlogn)，其中 n 是 scores 数组的长度。取前 m 名的操作是O(m)，但由于排序的时间复杂度更高，因此总体时间复杂度为 O(nlogn)。

空间复杂度:

存储排序后的数组需要 O(n) 的空间复杂度。 返回的列表空间复杂度是O(m)，但通常 O(n) 是主导的。

进阶思考

如果玩家数量达到数百万，单纯的排序可能会造成性能瓶颈。在这种情况下，可以考虑以下优化方案：

使用最小堆: 维护一个大小为 m 的最小堆，只保留前 m 个最高分数。这可以将时间复杂度降低到O(nlogm)，适合处理大量数据。

选择算法: 使用快速选择算法（QuickSelect）来找到第 m 大的元素，然后进行一次遍历，将所有大于等于该元素的分数收集起来，这样可以将时间复杂度优化到O(n)的平均情况。

2 魔法能量场

代码实现

```
using System;

public class EnergyFieldSystem
{
    public static float MaxEnergyField(int[] heights)
    {
```

```
        // 处理边界情况
        if (heights == null || heights.Length < 2)
        {
            return 0f; // 至少需要两个塔才能形成面积
        }

        int left = 0;
        int right = heights.Length - 1;
        float maxArea = 0;

        while (left < right)
        {
            // 计算当前梯形的面积
            float height = Math.Min(heights[left], heights[right]);
            float width = right - left;
            float area = (height + height) * width / 2; // 梯形面积计算公式

            maxArea = Math.Max(maxArea, area);

            // 移动指针
            if (heights[left] < heights[right])
            {
                left++; // 移动左指针
            }
            else
            {
                right--; // 移动右指针
            }
        }

        return maxArea;
    }
}
```

单元测试

```
using NUnit.Framework;

public class EnergyFieldSystemTests
{
    [Test]
    public void TestMaxEnergyField()
    {
        // 测试用例 1
        int[] heights1 = { 1, 8, 6, 2, 5, 4, 8, 3, 7 };
        float expected1 = 52.5f;
        Assert.AreEqual(expected1, EnergyFieldSystem.MaxEnergyField(heights1),
0.001f);

        // 测试用例 2
        int[] heights2 = { 1, 1 };
        float expected2 = 0.0f; // 只有两个高度相同的塔，无法形成面积
```

```
        Assert.AreEqual(expected2, EnergyFieldSystem.MaxEnergyField(heights2),
0.001f);

        // 测试用例 3
        int[] heights3 = { 5, 5, 5, 5, 5 };
        float expected3 = 20.0f; // 所有塔高度相同
        Assert.AreEqual(expected3, EnergyFieldSystem.MaxEnergyField(heights3),
0.001f);

        // 测试用例 4
        int[] heights4 = { 0, 0, 0, 0 };
        float expected4 = 0.0f; // 所有塔高度为0
        Assert.AreEqual(expected4, EnergyFieldSystem.MaxEnergyField(heights4),
0.001f);

        // 测试用例 5
        int[] heights5 = { 1, 2, 3, 4, 5 };
        float expected5 = 6.0f; // 最高的面积由1和5形成
        Assert.AreEqual(expected5, EnergyFieldSystem.MaxEnergyField(heights5),
0.001f);
    }
}
```

分析

时间复杂度:

双指针法的时间复杂度是 O(n)，其中n是heights数组的长度，因为我们最多遍历数组一次。

空间复杂度:

空间复杂度是O(1)，我们只使用了常数空间来存储指针和面积等变量。

进阶挑战

允许玩家使用魔法道具:

如果允许临时增加某个位置的塔的高度，我们可以在计算面积时，对于每一个位置考虑其高度加上道具的高度，并与其他塔组合计算，得到更大的面积。 这种情况下，可以使用类似于动态规划的方法，来评估不同位置上使用道具的影响。

建筑限制:

对于某些位置有建筑限制（高度为0），我们在计算面积时需要忽略这些位置。 在双指针遍历过程中，可以在遇到高度为0的塔时，直接跳过这些位置，不进行面积计算。

创意思考

能量场机制可以影响玩家的策略选择，例如：

玩家可能会选择在特定位置建造更高的塔以获得更大的面积。 玩家可以分析地图上的塔的位置，选择最佳组合以最大化能量场。 游戏中可以设计一些任务或挑战，要求玩家在有限时间内构建最大面积的能量场。

此外，还可以考虑引入时间限制、资源消耗等因素，来增加游戏的复杂度和趣味性。

魔法宝箱探险

代码实现

```
using System;

public class TreasureHuntSystem
{
    public static int MaxTreasureValue(int[] treasures)
    {
        // 处理边界情况
        if (treasures == null || treasures.Length == 0) return 0;
        if (treasures.Length == 1) return treasures[0];
        if (treasures.Length == 2) return Math.Max(treasures[0], treasures[1]);

        int n = treasures.Length;
        int[] dp = new int[n];

        // 初始化 dp 数组
        dp[0] = treasures[0];
        dp[1] = Math.Max(treasures[0], treasures[1]);

        // 动态规划填充 dp 数组
        for (int i = 2; i < n; i++)
        {
            dp[i] = Math.Max(dp[i-1], treasures[i] + dp[i-2]);
        }

        return dp[n-1];
```

```
    }
}
```

单元测试

```
using NUnit.Framework;

public class TreasureHuntSystemTests
{
    [Test]
    public void TestMaxTreasureValue()
    {
        // 测试用例 1
        int[] treasures1 = { 3, 1, 5, 2, 4 };
        int expected1 = 12; // 选择第1, 第3, 第5个宝箱
        Assert.AreEqual(expected1,
TreasureHuntSystem.MaxTreasureValue(treasures1));

        // 测试用例 2
        int[] treasures2 = { 2, 7, 9, 3, 1 };
        int expected2 = 12; // 选择第2, 第4个宝箱
        Assert.AreEqual(expected2,
TreasureHuntSystem.MaxTreasureValue(treasures2));

        // 测试用例 3: 只有一个宝箱
        int[] treasures3 = { 5 };
        int expected3 = 5;
        Assert.AreEqual(expected3,
TreasureHuntSystem.MaxTreasureValue(treasures3));

        // 测试用例 4: 两个宝箱
        int[] treasures4 = { 10, 15 };
        int expected4 = 15; // 选择第二个宝箱
        Assert.AreEqual(expected4,
TreasureHuntSystem.MaxTreasureValue(treasures4));

        // 测试用例 5: 全是负数
        int[] treasures5 = { -1, -2, -3, -4 };
        int expected5 = 0; // 什么也不选
        Assert.AreEqual(expected5,
TreasureHuntSystem.MaxTreasureValue(treasures5));
    }
}
```

分析

时间复杂度：

该算法的时间复杂度是 O(n)，其中 n 是宝箱的数量，因为我们只需要遍历数组一次。

空间复杂度：

该算法的空间复杂度是 O(n)，因为我们使用了一个大小为 n 的 dp 数组。如果想进一步优化空间复杂度，可以用两个变量代替 dp 数组，只保存前两个状态。

进阶挑战

使用"魔法钥匙"：

如果允许使用一次“魔法钥匙”来打开两个相邻的宝箱，可以在计算时考虑两种情况：不使用魔法钥匙的情况，按当前算法处理。 使用魔法钥匙的情况，计算打开相邻宝箱的收益。 最终取两种情况下的最大值。

高级关卡：包含负值宝箱:

对于负值的宝箱，如果该宝箱的价值为负，则应该尽量避免打开它。动态规划中的转移方程依旧适用，算法将会自动忽略负收益的宝箱。

创意思考

这个机制带来了一些有趣的策略选择：

玩家需要在每个宝箱之间做出取舍，选择性的跳过一些宝箱，权衡每次选择的收益和损失。 如果允许使用道具（如魔法钥匙），玩家可以保存道具在关键时刻使用。 在设计游戏关卡时，可以增加复杂性，比如让部分宝箱隐藏价值，或者有时间限制，迫使玩家快速决策。

这个概念可以进一步扩展，设计出多种不同的宝箱排列组合和关卡条件，增加游戏的策略性与挑战性。

魔法天赋评估系统

代码实现

```
using System;

public class TalentAssessmentSystem
{
    public static double FindMedianTalentIndex(int[] fireAbility, int[] iceAbility)
    {
        // 确保 fireAbility 是较短的数组
        if (fireAbility.Length > iceAbility.Length)
        {
            return FindMedianTalentIndex(iceAbility, fireAbility);
        }

        int m = fireAbility.Length;
        int n = iceAbility.Length;
        int totalLength = m + n;
        int halfLength = (m + n + 1) / 2;

        int low = 0, high = m;

        while (low <= high)
        {
            int partitionFire = (low + high) / 2;
            int partitionIce = halfLength - partitionFire;

            int maxLeftFire = (partitionFire == 0) ? int.MinValue :
fireAbility[partitionFire - 1];
            int minRightFire = (partitionFire == m) ? int.MaxValue :
fireAbility[partitionFire];

            int maxLeftIce = (partitionIce == 0) ? int.MinValue :
iceAbility[partitionIce - 1];
            int minRightIce = (partitionIce == n) ? int.MaxValue :
iceAbility[partitionIce];

            if (maxLeftFire <= minRightIce && maxLeftIce <= minRightFire)
            {
                // 找到合适的分割点
                if ((m + n) % 2 == 0)
                {
                    return (Math.Max(maxLeftFire, maxLeftIce) +
Math.Min(minRightFire, minRightIce)) / 2.0;
                }
                else
                {
                    return Math.Max(maxLeftFire, maxLeftIce);
                }
            }
            else if (maxLeftFire > minRightIce)
            {
                high = partitionFire - 1;
            }
            else
            {
                low = partitionFire + 1;
            }
        }
```

```
            throw new ArgumentException("输入数组不正确");
        }
    }
```

单元测试

```
using NUnit.Framework;

public class TalentAssessmentSystemTests
{
    [Test]
    public void TestFindMedianTalentIndex()
    {
        // 测试用例1
        int[] fireAbility1 = {1, 3, 7, 9, 11};
        int[] iceAbility1 = {2, 4, 8, 10, 12, 14};
        double expected1 = 8.0;
        Assert.AreEqual(expected1,
TalentAssessmentSystem.FindMedianTalentIndex(fireAbility1, iceAbility1), 1e-5);

        // 测试用例2: 两个数组长度相同
        int[] fireAbility2 = {1, 2, 5};
        int[] iceAbility2 = {3, 4, 6};
        double expected2 = 3.5;
        Assert.AreEqual(expected2,
TalentAssessmentSystem.FindMedianTalentIndex(fireAbility2, iceAbility2), 1e-5);

        // 测试用例3: 两个数组只有一个元素
        int[] fireAbility3 = {1};
        int[] iceAbility3 = {2};
        double expected3 = 1.5;
        Assert.AreEqual(expected3,
TalentAssessmentSystem.FindMedianTalentIndex(fireAbility3, iceAbility3), 1e-5);

        // 测试用例4: 两个数组一个为空
        int[] fireAbility4 = {};
        int[] iceAbility4 = {1, 2, 3};
        double expected4 = 2.0;
        Assert.AreEqual(expected4,
TalentAssessmentSystem.FindMedianTalentIndex(fireAbility4, iceAbility4), 1e-5);
    }
}
```

分析

时间复杂度:

算法的时间复杂度是 O(logmin(m,n))，其中 m 和 n 是两个数组的长度。我们每次通过二分查找在较小的数组上进行操作，从而在对数时间内找到合适的分割点。

空间复杂度：

算法的空间复杂度是 O(1)，因为我们只使用了常数空间来存储变量。

进阶挑战

实时更新大量学徒的天赋指数：

可以使用类似于“滑动窗口”的技术来处理动态更新的情况。在插入或删除时，只需要在已排序数组中找到相应位置插入或删除元素即可，这可以通过数据结构如平衡树或二分查找加以优化。

处理多个有序数组的中位数：

如果有多个有序数组，可以使用“归并排序”的思想将多个数组进行二分划分，进而找到合并后的中位数，或者使用一个小顶堆/大顶堆的组合来动态维护中位数。

创意思考

这个天赋评估系统可以扩展为玩家角色的技能成长系统。在角色的成长过程中，不同魔法属性（如火、冰、雷电等）的能力值变化会影响角色的技能学习、任务完成效率等。根据天赋评估系统，玩家可以通过合适的成长路径提升自己的综合能力。在任务系统或 PvP 对战中，这种天赋评估还可以作为决策依据，决定技能使用或战术策略，进而为游戏带来更多策略性玩法。