第二次实验

1. 问题描述

> 这个问题描述了一个关于切割钢条的优化问题。给定一根长度为L的钢条和n个在钢条上标注的位置点，需要将钢条切割为n+1段。切割的代价与切割时钢条的长度成正比。问题的目标是找到一种切割方案，使得总的切割代价达到最小。
>
> 输入部分包括钢条的总长度L，位置点的个数n，以及一个长度为n的数组p，其中存储了所有位置点的坐标（这些坐标可能是乱序的）。我们需要在钢条的两端添加两个虚拟的位置点0和L，以便于处理边界情况。
>
> 输出是一个整数，表示最小的切割总代价。
>
> 这个问题要求我们编写一个函数`MinCost`，实现上述功能，并在`main`函数中调用这个函数，输入钢条长度，位置点个数和位置点坐标，然后输出最小的切割代价。

2. 问题分析

> ## 2.1. 问题分解
>
> > 我们考虑每次切割一个小段钢条的代价是其长度，那么对于任意一个位置k，从位置i到j的切割代价就是长度加上左右两边的切割代价。我们可以考虑所有可能的k来找到最小代价。
>
> ## 2.2. 递归结构
>
> > 可以尝试定义一个递归函数`cost(i, j)`来表示从位置i到j的最小切割代价。对于每个k（i < k < j），尝试切割并加上左右两侧的切割代价。这种递归结构在没有优化的情况下会导致大量的重复计算。
>
> ## 2.3. 动态规划引入
>
> > 考虑到递归结构中的重复计算，可以引入动态规划来保存已经计算过的解。我们可以使用一个二维数组`cost`来保存从位置i到j的最小切割代价。
>
> ## 2.4. 状态转移方程的构建
>
> > 对于每个位置k（i < k < j），`cost(i, j)`可以表示为切割在k的代价（即`j - i`，也就是i到j之间的长度）加上`cost(i, k)`和`cost(k, j)`。我们需要找到这样的k，使得整体代价最小。
>
> ## 2.5. 迭代填表
>
> > 考虑到`cost(i, j)`依赖于`cost(i, k)`和`cost(k, j)`，我们可以从小到大的长度来填表。首先，我们可以计算所有长度为2的`cost(i, j)`，然后是长度为3的，依此类推，直到整个钢条的长度。
>
> ## 2.6. 初始化与边界处理
>
> > 对于长度为1或者相邻的切割点，切割代价为0，这是我们的初始化条件。为了处理整个钢条的两端，我们在位置点数组的开始和结束添加了两个位置0和L。

2.7. 最终解+优化

> 在完成上述所有步骤后，`cost[0][n+1]`即为整个钢条的最小切割代价。如果需要，可以考虑空间优化，如滚动数组等方法。

3. 算法设计

3.1. 排序切割点

```
vector<int> helper(p, p + n + 2);
sort(helper.begin(), helper.end());
```

这里，`helper`数组不仅仅包括了给出的切割点`p`，还有起始的0和结束的L。排序的目的是为了后续计算中方便确定区间长度，同时有序的数组对于后续的动态规划策略也是友好的。

3.2. 初始化动态规划数组

```
int size = helper.size();
vector<vector<int>> cost(size, vector<int>(size, 0));
```

- `cost[i][j]`用于存储从位置`helper[i]`到`helper[j]`之间的最小切割代价。
- `size`即为切割点数量加上起始和结束位置，总共为`n+2`。

3.3. 动态规划填表：

- 外层循环的目标是逐渐增加考虑的钢条的长度：

```
for (int len = 2; len < size; ++len) {
```

为什么从2开始？因为长度为1的情况是两个相邻的点，它们之间的切割代价是0，这个是基础情况。

- 中层循环确定了当前考虑的钢条段的起始点`i`和结束点`j`：

```
for (int i = 0; i < size - len; ++i) {
    int j = i + len;
    cost[i][j] = INT_MAX;
```

- `cost[i][j] = INT_MAX`是一个初始设置，确保后续在对比时能够找到一个更小的代价值。
- 内层循环则是核心部分，它对`i`和`j`之间所有可能的切割点进行迭代：

```
for (int k = i + 1; k < j; ++k) {
```

这里`k`是实际的切割点位置，它始终在`i`和`j`之间。

- 对于每一个`k`，都会计算这样切割的代价，并与当前代价对比，取较小值：

```
int tempCost = helper[j] - helper[i] + cost[i][k] + cost[k][j];
cost[i][j] = min(cost[i][j], tempCost);
```

- `helper[j] - helper[i]`是因为题目中提到，切割的代价与钢条的长度成正比。
- `cost[i][k] + cost[k][j]`是左右两部分的切割代价。

3.4. 输出结果

```
cout << cost[0][size - 1] << endl;
```

- `cost[0][size - 1]`就是整个钢条从开始到结束的最小切割代价。此时我们已经考虑过了所有的钢条段和所有的可能切割位置，所以这个值是确保最小的。

这个动态规划方法的基础是对各种可能性进行完全的考察。每次当我们考虑一个更长的钢条段时，都依赖于更小的钢条段的最小代价，这正是动态规划的精髓所在

4. 算法实现

4.1. 实现1(提交版本)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <climits>
using namespace std;
void MinCost(int L, int n, int *p) {
    // 首先对切割点进行排序
    vector<int> helper(p, p + n + 2);
    sort(helper.begin(), helper.end());
    int size = helper.size();
    vector<vector<int>> cost(size, vector<int>(size, 0));
    // 计算不同长度的切割情况
    for (int len = 2; len < size; ++len) {
        for (int i = 0; i < size - len; ++i) {
            int j = i + len;
            cost[i][j] = INT_MAX;
            for (int k = i + 1; k < j; ++k) {
                // 每次尝试不同的切割点并更新代价
                int tempCost = helper[j] - helper[i] + cost[i][k] +
cost[k][j];
                cost[i][j] = min(cost[i][j], tempCost);
            }
        }
    }
    // 输出从第一个到最后一个切割点的最小代价
    cout << cost[0][size - 1] << endl;
}
```

4.2. 三行滚动数组优化版

空间复杂度从`O(n^2)`降低到了`O(n)`

```
void MinCost(int L, int n, int *p) {
    // 初始化
    vector<int> points(p, p + n + 2);
```

```
    sort(points.begin(), points.end());
    int size = points.size();

    // 使用三行的滚动数组
    vector<vector<int>> helper(3, vector<int>(size, 0));

    // 计算不同长度的切割情况
    for (int len = 2; len < size; ++len) {
        for (int i = 0; i < size - len; ++i) {
            int j = i + len;
            helper[2][j] = INT_MAX;

            for (int k = i + 1; k < j; ++k) {
                int tempCost = points[j] - points[i] + helper[0][k] +
helper[1][j];
                helper[2][j] = min(helper[2][j], tempCost);
            }
        }

        // 滚动数组
        for (int j = 0; j < size; ++j) {
            helper[0][j] = helper[1][j];
            helper[1][j] = helper[2][j];
        }
    }

    // 输出从第一个到最后一个切割点的最小代价
    cout << helper[2][size - 1] << endl;
}
```

5. 运行结果

```
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.90.1-microsoft-standard-WSL2
x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sat Oct 28 19:28:30 CST 2023

  System load:  0.23               Processes:             117
  Usage of /:   1.8% of 250.92GB   Users logged in:       0
  Memory usage: 4%                 IPv4 address for eth0: 172.17.211.221
  Swap usage:   0%

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how
MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster
deployment.

   https://ubuntu.com/engage/secure-kubernetes-at-the-edge

Expanded Security Maintenance for Applications is not enabled.
```

```
0 updates can be applied immediately.

30 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm



This message is shown once a day. To disable it please create the
/home/dann_hiroaki/.hushlogin file.
dann_hiroaki@DESKTOP-QANEDCT:~$ gedit a.cpp
^C
dann_hiroaki@DESKTOP-QANEDCT:~$ g++ a.cpp
dann_hiroaki@DESKTOP-QANEDCT:~$ ./a.out
7 4
1 3 4 5
16
```