# 《算法设计与分析》
# 课程实验报告


专业：计算机科学与技术
班级：2021211304
姓名：杨晨
学号：2021212171

# 1 概述

## 1.1 实验内容

- 编程实现下述 2 组算法之一（二选一），并利用给定的数据，验证算法正确性
  - 第一组
    - 最长公共子序列
    - 最大子段和
  - 第二组
    - 凸多边形最优三角剖分
  - 动态规划法求最优解，启发式/贪心策略求次优解

## 1.2 开发环境

- Windows10
- PyCharm 2023.2.4 (Professional Edition)
- Visual Studio Code 1.84.2

# 2 实验过程

## 2.1 最长公共子序列

### 2.1.1 介绍

最长公共子序列（Longest Common Subsequence，简称 LCS）问题是计算机科学中的经典问题，在 DNA 序列对比、文本比较和版本控制系统等领域都有应用。给定两个序列，目标是找到在两个序列中都出现的最长子序列，该子序列不一定是连续的。

### 2.1.2 算法描述

使用动态规划方法解决最长公共子序列（LCS）问题。首先，定义两个输入序列 $X$ 和 $Y$，其中 $X$ 的长度为 $m$，$Y$ 的长度为 $n$。使用一个二维数组 $dp$ 来存储中间结果，其中 $dp[i][j]$ 表示序列 $X$ 的前 $i$ 个元素和序列 $Y$ 的前 $j$ 个元素的最长公共子序列的长度。

算法的核心思想是通过填充 $dp$ 数组来逐步构建最长公共子序列。使用另一个二维数组 $path$ 来记录填充 $dp$ 数组时的选择路径，以便在最后构建最长公共子序列时进行回溯。

具体而言，按照以下步骤执行算法：

1. 初始化 $dp$ 和 $path$ 数组为大小为 $(m+1) \times (n+1)$ 的二维数组，并将所有元素初始化为 0。
2. 使用双重循环遍历序列 $X$ 和序列 $Y$ 的所有元素：
   - 如果 $X[i]$ 等于 $Y[j]$，则说明当前元素是公共子序列的一部分，因此将 $dp[i][j]$ 更新为 $dp[i-1][j-1]+1$，并将 $path[i][j]$ 设置为 1，表示选择该元素。
   - 否则，比较 $dp[i-1][j]$ 和 $dp[i][j-1]$ 的值：

- 如果 $dp[i-1][j]$ 大于等于 $dp[i][j-1]$，则说明选择 $X$ 的第 $i$ 个元素作为公共子序列的一部分，将 $dp[i][j]$ 更新为 $dp[i-1][j]$，并将 $path[i][j]$ 设置为 2。
- 否则，选择 $Y$ 的第 $j$ 个元素作为公共子序列的一部分，将 $dp[i][j]$ 更新为 $dp[i][j-1]$，并将 $path[i][j]$ 设置为 3。

3. 最后，通过对 $path$ 数组进行回溯，可以构建得到最长公共子序列。

**Algorithm 1** 递归输出 LCS

```
1:  procedure LCS(i, j, X, path)
2:      if i = 0 or j = 0 then
3:          return
4:      if path[i][j] = 1 then
5:          LCS(i − 1, j − 1, X, path)
6:          PRINT(X[i])
7:      else if path[i][j] = 2 then
8:          LCS(i − 1, j, X, path)
9:      else
10:         LCS(i, j − 1, X, path)
```

**Algorithm 2** LCSLength 算法

```
1:  procedure LCSLENGTH(X, Y)
2:      m ← length of X
3:      n ← length of Y
4:      dp ← new int array of size (m + 1) × (n + 1)          ▷ 初始化为 0
5:      path ← new int array of size (m + 1) × (n + 1)        ▷ 初始化为 0
6:      for i ← 1 to m do
7:          for j ← 1 to n do
8:              if X[i] = Y[j] then
9:                  dp[i][j] ← dp[i − 1][j − 1] + 1
10:                 path[i][j] ← 1
11:             else if dp[i − 1][j] ≥ dp[i][j − 1] then
12:                 dp[i][j] ← dp[i − 1][j]
13:                 path[i][j] ← 2
14:             else
15:                 dp[i][j] ← dp[i][j − 1]
16:                 path[i][j] ← 3
17:     LCS(m, n, X, path)
```

### 2.1.3 分析和改进

LCS 算法的时间复杂度为 $O(m \times n)$，其中 $m$ 和 $n$ 分别是序列 $X$ 和序列 $Y$ 的长度。算法需要填充一个大小为 $(m+1) \times (n+1)$ 的二维数组，并进行双重循环遍历。空间复杂度也为 $O(m \times n)$，用于存储 $dp$ 和 $path$ 数组。

在分析代码时，注意到以下改进的可能性：

空间复杂度优化：给定代码中使用了一个大小为 $(n+1) * (m+1)$ 的一维数组 dp 来存储动态规划的结果。然而，实际上在每次迭代中，只需要使用 dp[i][j]、dp[i][j - 1] 和 dp[i - 1][j - 1] 这三个变量。因此，如果只计算长度的话，可以使用一维数组。这样可以将空间复杂度从 $O(m \times n)$ 优化为 $O(n)$。

空间复杂度优化后的伪代码如下：

**Algorithm 3** 计算最长公共子序列长度（空间复杂度 O(n)）

```
procedure LCS_SpaceSaving(X, Y)
    m ← length of (X)
    n ← length of (Y)
    dp ← 新数组(n + 1)
    for i ← 0 to n do
        dp[i] ← 0
    for i ← 1 to m do
        upLeft ← dp[0]                                  ▷ 保存左上角的值
        for j ← 1 to n do
            temp ← dp[j]      ▷ 保存当前位置的值，即 dp[i − 1][j]，因为 dp[j] 已经被更新了
            if X[i] = Y[j] then
                dp[j] ← upLeft + 1
            else
                dp[j] ← Max(dp[j], dp[j − 1])
            upLeft ← temp                               ▷ 更新左上角的值
```

### 2.1.4 运行结果

```
A and B LCS:
an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+
input+and+produces+some+values+as+output20212113xx2023
LCSLength is: 137
LCSLength is: 137

A and C LCS:
an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+
input+and+produces+some+values+as+output20212113xx2023
LCSLength is: 137
```

```
LCSLength is: 137

A and D LCS:
an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+
input+and+produces+some+values+as+output20212113xx2023
LCSLength is: 137
LCSLength is: 137

C and B LCS:
an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+
input+and+produces+some+values+as+output20212113xx2023
LCSLength is: 137
LCSLength is: 137
```

#### 2.1.5 结论

LCS 算法提供了一种有效的方法来寻找两个序列中的最长公共子序列。通过动态规划的思想，算法能够在较快的时间内计算出最长公共子序列的长度，并通过回溯路径构建得到最长公共子序列。如果不需要得到具体的序列，空间复杂度可以优化为 O(n)。

### 2.2 最长不上升子序列

#### 2.2.1 介绍

最长不上升子序列（Longest Decreasing Subsequence，简称 LDS）是指在给定序列中找到最长的子序列，使得子序列中的元素按照非递增的顺序排列。LDS 问题在计算机科学和算法设计中具有重要意义，广泛应用于数据分析、信号处理、优化问题等领域。

#### 2.2.2 算法描述

首先介绍一种基于最长公共子序列的方法，来求解最长不上升子序列问题的算法。

借助最长公共子序列（LCS）的思想来解决最长不上升子序列问题首先，将原始序列保存在 $num\_temp$ 中，并对 $num$ 进行非递增排序，得到新的序列。然后，使用动态规划的方法计算最长公共子序列的长度和路径。最后，根据路径信息输出最长不上升子序列。

下面是借助最长公共子序列（LCS）求解最长不上升子序列问题的算法的伪代码：

**Algorithm 4** 借助最长公共子序列求最长不上升子序列

```
1: procedure LDS_BY_LCS(num : vector<int>)
2:     n ← num.size()
3:     num_temp ← num
4:     sort(num.begin(), num.end(), greater<int>())                    ▷ 以降序排序
5:     dp ← vector<vector<int»(n + 1, vector<int>(n + 1, 0))
6:     path ← vector<vector<int»(n + 1, vector<int>(n + 1, 0))
7:     for i ← 1 to n do
8:         for j ← 1 to n do
9:             if num[i − 1] = num_temp[j − 1] then
10:                dp[i][j] ← dp[i − 1][j − 1] + 1
11:                path[i][j] ← 1
12:            else if dp[i − 1][j] ≥ dp[i][j − 1] then
13:                dp[i][j] ← dp[i − 1][j]
14:                path[i][j] ← 2
15:            else
16:                dp[i][j] ← dp[i][j − 1]
17:                path[i][j] ← 3
18:    printf " 最长不上升子序列长度为: " + dp[n][n]
19:    printf " 最长不上升子序列为: "
20:    procedure OUTPUT(i : int, j : int)
21:        if i = 0 or j = 0 then
22:            return
23:        if path[i][j] = 1 then
24:            output (i − 1, j − 1)
25:            printf num[i − 1]
26:        else if path[i][j] = 2 then
27:            output (i − 1, j)
28:        else
29:            output (i, j − 1)
30:    output (n, n)
```

此外，还可以使用动态规划的思想来解决最长不上升子序列问题。首先，定义一个长度为 $n$ 的数组 $dp$，其中 $dp[i]$ 表示以序列中第 $i$ 个元素结尾的最长不上升子序列的长度。初始化 $dp$ 数组的所有元素为 1，表示每个元素本身就构成一个长度为 1 的子序列。

然后，通过两层循环遍历序列中的每个元素，比较当前元素和前面已经遍历过的元素的大小关系。如果当前元素小于等于前面的元素，并且以前面元素结尾的子序列的长度加 1 大于以当前元素结尾的子序列的长度，则更新 $dp$ 数组和路径数组 $path$。

最后，遍历 $dp$ 数组找到最长的子序列长度和对应的索引，根据路径数组 $path$ 逆序输出最长不上升子序列的元素。

下面是基于动态规划的最长不上升子序列算法的伪代码：

**Algorithm 5** 最长不上升子序列

```
1:  procedure LDS(num: vector<int>)
2:      n ← num.size()
3:      dp ← vector<int>(n, 1) // dp[i] 表示以 num[i] 结尾的最长不上升子序列的长度
4:      path ← vector<int>(n, -1)
5:      for i ← 1 to n − 1 do
6:          for j ← 0 to i − 1 do
7:              if num[i] ≤ num[j] and dp[i] < dp[j] + 1 then
8:                  dp[i] ← dp[j] + 1
9:                  path[i] ← j
10:     ans ← dp[0]
11:     index ← 0
12:     for i ← 0 to n − 1 do
13:         if ans < dp[i] then
14:             ans ← dp[i]
15:             index ← i
16:     printf " 最长不上升子序列长度为： " + ans
17:     printf " 最长不上升子序列为： "
18:     procedure OUTPUT(x : int)                    ▷ 匿名函数，定义了一个输出函数
19:         if x = −1 then
20:             return
21:         output path[x]
22:         printf num[x]
23:     output index
```

### 2.2.3 分析和改进

第一个算法的时间复杂度为 $O(n^2)$，空间复杂度为 $O(n^2)$，其中 $n$ 是序列的长度。算法使用了一个二维数组 $dp$ 来保存最长公共子序列的长度，以及一个二维数组 $path$ 来保存路径信息。在输出最长不上升子序列时，使用了递归函数 $output$ 来回溯路径并输出结果。

第二个算法的时间复杂度为 $O(n^2)$，但是空间复杂度降为了 $O(n)$，其中 $n$ 是序列的长度。算法使用了一个一维数组 $dp$ 来保存最长公共子序列的长度，以及一个一维数组 $path$ 来保存路径信息。在输出最长不上升子序列时，使用了递归函数 $output$ 来回溯路径并输出结果。

当然，如果只找最长不上升序列的长度，时间复杂度还可以进一步降低。将原来的 dp 数组的存储由数值换成该序列中，不上升子序列长度为 i 的序列中的，最大的末尾数值

这其实就是一种几近贪心的思想：当前的不上升子序列长度如果已经确定，那么如果这种长度的子序列的结尾元素越大，后面的元素就可以更方便地加入到这条我们臆测的、可作为结果、的不上升子序列中。

伪代码如下：

**Algorithm 6** 计算最长不上升子序列的长度（时间复杂度 O(nlogn)）

**procedure** LDS_NLOGN(num : vector<int>)
    $n \leftarrow$ num.size()
    dp $\leftarrow$ vector<int>(n + 1, INT_MAX) ▷ dp[i] 表示长度为 i 的最长不上升子序列的最大末尾元素
    $len \leftarrow 0$
    **for** $i \leftarrow 0$ to $n - 1$ **do**
        **if** num$[i] \leq$ dp$[len]$ **then** ▷ 如果 num[i]≤ 最长不上升子序列的最大末尾元素，那么 num[i] 可以接在最长不上升子序列后面，形成一个更长的不上升子序列
            $len \leftarrow len + 1$
            dp$[len] \leftarrow$ num$[i]$
        **else** ▷ 否则，找到 dp 中第一个小于 num[i] 的元素，用 num[i] 替换它
            $pos \leftarrow$ upper_bound(dp.begin() + 1, dp.begin() + $len$ + 1, num$[i]$, greater<int>()) − dp.begin()
            dp$[pos] \leftarrow$ num$[i]$
    **printf** ” 最长不上升子序列长度为：” + $len$

第三个算法的时间复杂度为 $O(n \log n)$，其中 $n$ 是序列的长度。算法通过遍历输入序列并维护一个长度为 $L$ 的数组 dp，其中 $L$ 是当前最长不上升子序列的长度。在遍历过程中，算法通过二分查找找到合适的位置，将当前元素插入 dp 数组中。最后，dp 数组的长度即为最长不上升子序列的长度。需要注意的是，dp 数组中的元素不是真正的不上升子序列。

### 2.2.4 运行结果

```
num1:
最长不上升子序列长度为: 40
最长不上升子序列为: 99 99 95 93 89 87 87 79 76 72 68 68 68 60 56 51 50 47 36 31 27
27 27 19 3 0 0 0 -4 -10 -14 -15 -17 -22 -27 -57 -100 -211 -301 -305
最长不上升子序列长度为: 40
最长不上升子序列为: 99 99 95 91 89 87 87 79 76 72 65 61 60 56 56 51 50 47 36 31 27
27 27 19 3 0 0 0 -4 -10 -14 -15 -17 -22 -31 -100 -200 -230 -301 -305
最长不上升子序列长度为: 40

num2:
最长不上升子序列长度为: 30
最长不上升子序列为: 100 49 47 47 39 38 37 34 34 34 33 28 27 24 24 22 10 8 -10 -11
-16 -25 -32 -38 -39 -42 -44 -70 -304 -307
```

```
最长不上升子序列长度为: 30
最长不上升子序列为: 100 49 47 47 39 38 37 34 34 34 33 28 27 24 24 5 -2 -6 -10 -11
-25 -25 -32 -38 -41 -42 -44 -70 -304 -307
最长不上升子序列长度为: 30
```

#### 2.2.5 结论

通过使用转化为最长公共子序列或者动态规划，可以在给定序列中找到最长的不上升子序列。算法的复杂度为 $O(n^2)$，其中 $n$ 是序列的长度。

但是，如果只寻找最长不上升子序列的长度，时间复杂度可以进一步降低为 $O(n \log n)$。同时，也论证了一个观点: 时间复杂度越高的算法越全能。

通过分析和实验，发现该算法可以有效地解决最长不上升子序列问题，并且在实际应用中具有较好的性能。然而，当序列长度非常大时，该算法的时间复杂度可能变得较高，因此在处理大规模数据时需要考虑其他优化方法。

### 2.3 最大子段和

#### 2.3.1 介绍

最大子段和问题是一个经典的算法问题，其目标是在给定的整数数组中找到一个连续子数组，使得该子数组的和最大。这个问题在计算机科学和算法设计中具有重要意义，可以应用于许多实际问题，如股票交易、财务分析等领域。

#### 2.3.2 算法描述

首先，介绍基于分治法的最大子段和算法。基于分治法，将问题分解为更小的子问题，并通过递归求解子问题来获得最终的解。下面是伪代码:

**Algorithm 7** 最大子段和算法

```
1: function MaxSubSum(num, left, right)
2:     if left = right then
3:         return num[left] > 0?num[left] : 0
4:     mid ← (left + right)/2
5:     leftMax ← MaxSubSum(num, left, mid)
6:     rightMax ← MaxSubSum(num, mid + 1, right)
7:     leftBorderMax ← num[mid]
8:     leftBorderSum ← 0
9:     for i ← mid downto left do
10:        leftBorderSum ← leftBorderSum + num[i]
11:        if leftBorderSum > leftBorderMax then
12:            leftBorderMax ← leftBorderSum
13:    rightBorderMax ← num[mid + 1]
14:    rightBorderSum ← 0
15:    for i ← mid + 1 to right do
16:        rightBorderSum ← rightBorderSum + num[i]
17:        if rightBorderSum > rightBorderMax then
18:            rightBorderMax ← rightBorderSum
19:    borderMax ← leftBorderMax + rightBorderMax
20:    return max(max(leftMax, rightMax), borderMax)
21:
22: function MaxSubSum(num)                          ▷ 重载隐藏参数
23:    return MaxSubSum(num, 0, size(num) − 1)
```

### 2.3.3 分析与改进

在之前的报告中，介绍了基于分治法的最大子段和算法。下面将介绍一种改进后的算法，基于动态规划思想，来解决最大子序列和问题。

改进后的最大子序列和算法基于动态规划思想，使用一个变量 $dp$ 来记录当前位置的最大子序列和。算法从数组的第二个元素开始遍历，若 $dp$ 大于零，则将当前元素加到 $dp$ 上；否则，将 $dp$ 更新为当前元素，并更新最大子序列的起始位置，因为负数对后边的序列是无用的。在遍历的过程中，不断更新最大子序列和 $ans$ 和最大子序列的终止位置 $right$。

下面是改进后的最大子序列和算法的伪代码：

**Algorithm 8** 最大子序列和改进算法

```
procedure MaxSum(num)
    n ← size(num)
    dp ← num[0]
    ans ← num[0]
    left ← 0
    right ← 0
    for i ← 1 to n − 1 do
        if dp > 0 then
            dp ← dp + num[i]
        else
            dp ← num[i]
            left ← i
        if dp > ans then
            ans ← dp
            right ← i
    printf " 最大子序列和为: " ans
    printf " 最大子序列为: "
    for i ← left to right do
        printf num[i]
```

### 2.3.4 运行结果

```
num1最大子段和为: 6914
num1最大子序列和为: 6914
最大子序列为: 64 87 99 39 31 9 99 -2 -7 83 -46 8 16 55 -88 31 -96 51 -60 90 -13 80
50 -88 -9 -84 95 68 -23 24 53 -94 91 60 -34 -19 -53 -40 13 -31 -35 70 25 38 65 49
-99 68 -18 17 79 70 11 -93 93 -24 13 74 70 20 -2 66 97 -20 -56 89 5 -86 87 -56 53
60 73 15 -83 -73 -11 59 -85 87 -24 -81 79 70 -12 29 -4 63 -58 -48 94 20 -68 -10 76
97 72 -56 -45 -96 3 53 60 13 97 65 22 78 99 -12 68 -13 24 -73 -89 22 61 -31 73 5 27
 81 -85 55 68 -56 43 60 -19 -23 77 -91 -61 -57 22 -39 -64 29 41 -15 -43 -43 -4 -47
49 -21 66 0 56 45 71 -16 -35 68 60 -26 98 -22 -62 56 51 -63 -83 -62 -48 -33 9 11 5
57 93 35 32 -80 -54 -87 -82 -96 39 93 -89 50 29 47 7 -13 80 23 -85 -38 3 25 36 31
92 46 82 -23 -46 91 89 -40 76 -12 53 -88 -74 27 49 14 42 -60 -32 -43 -18 65 -57 27
27 46 68 -29 63 84 -9 40 -42 -4 -32 -35 82 19 35 -15 84 76 -28 -42 -99 39 79 -54 -9
 98 -77 95 -82 -60 -86 3 0 -85 70 -80 33 0 57 73 94 -50 -91 -46 0 42 -98 43 68 -18
-4 25 32 65 -29 -62 -76 78 12 -30 -10 61 94 92 -67 20 -51 33 95 -97 -94 -14 68 31
-15 -55 19 23 -44 25 -17 -48 45 72 37 -22 48 -31 -27 82 16 -20 30 -100 -200 50 60
300 -2 8 -230 45 78 -69 54 29 300 -57 63 70 -120 80 -100 40 20 30 2021 -211 -301
304 -305 307 2023

num2最大子段和为: 2583
```

```
num2最大子序列和为: 2583
最大子序列为: 34 1 -5 40 8 2 6 23 30 42 -4 45 -25 -23 -22 34 -13 -11 -12 16 44 -3
-11 -7 -30 34 49 -47 1 -21 -37 14 33 -37 28 -33 15 -36 36 27 -8 -31 24 -16 -7 38 24
 34 48 -27 -22 5 33 9 -26 -2 48 -20 22 38 -42 4 5 -49 10 47 -6 27 8 -10 34 -11 -25
-35 -17 38 43 -9 -8 -16 -25 -32 10 -38 -41 -18 -1 37 1 25 -39 -35 10 -23 -23 -20 43
 -4 -42 -9 44 26 -16 41 -1 20 -33 50 -100 130 32 -99 2 -44 36 -15 80 8 15 -15 31 10
 20 -40 60 -70 20 30 40 -21 211 301 -304 305 -307 2023
```

#### 2.3.5 结论

基于分治的算法的时间复杂度为 $O(n \log n)$，其中 $n$ 是输入数组的大小。这是因为在每次递归调用中，问题的规模减少一半，而对于每个子问题，需要线性时间来计算边界最大和。因此，总体的时间复杂度为 $O(n \log n)$。

基于动态规划的算法的时间复杂度为 $O(n)$，其中 $n$ 是输入数组的大小。

### 2.4 凸多边形最优三角剖分

#### 2.4.1 介绍

凸多边形最优三角剖分问题是计算机图形学中的一个经典问题，其目标是在给定的凸多边形中找到一种三角剖分方式，使得剖分后的三角形的权值之和最小。

#### 2.4.2 算法描述

凸多边形最优三角剖分算法基于动态规划思想，通过填表的方式求解最优解。算法维护一个二维数组 $dp$，其中 $dp[i][j]$ 表示从第 $i$ 个顶点到第 $j$ 个顶点的最优三角剖分的最小权值。同时，算法还维护一个二维数组 $s$，用于记录最优三角剖分对应的顶点。算法从小到大枚举区间长度，然后从左端点开始枚举，计算每个区间的最优三角剖分权值，并更新 $dp$ 和 $s$ 数组。最后，调用 $draw\_stations$ 函数将最优三角剖分可视化。

下面是凸多边形最优三角剖分问题的动态规划算法的伪代码：

**Algorithm 9** 凸多边形最优三角剖分算法

**procedure** OPTIMALTRIANGULATION(stations)
    $n \leftarrow \text{len}(stations)$
    $dp \leftarrow$ [[0 for _ in range($n$)] for _ in range($n$)]  ▷ n × n 二维列表，初始化为 0
    $s \leftarrow$ [[0 for _ in range($n$)] for _ in range($n$)]  ▷ n × n 二维列表，初始化为 0
    **for** $i$ in range($n-1$) **do**
        $dp[i][i+1] \leftarrow 0$
    **for** len_ in range($3, n+1$) **do**
        **for** $l$ in range($n - \text{len\_} + 1$) **do**
            $r \leftarrow l + \text{len\_} - 1$
            $dp[l][r] \leftarrow$ 极大值
            **for** $k$ in range($l+1, r$) **do**
                **if** $dp[l][r] > dp[l][k] + dp[k][r] + weight$ **then**
                    $dp[l][r] \leftarrow dp[l][k] + dp[k][r] + weight$
                    $s[l][r] \leftarrow k$
    **调用** $draw\_stations(stations, s)$  ▷ 画图展示最优三角剖分
    **print** $dp[0][n-1]$

### 2.4.3 分析和改进

凸多边形最优三角剖分算法的时间复杂度主要取决于两个嵌套的循环：外层循环枚举区间长度，内层循环枚举左端点。对于给定的区间长度，内层循环的迭代次数是固定的，取决于区间长度。因此，算法的时间复杂度可以表示为 $O(n^3)$，其中，$n$ 是凸多边形的顶点数。在内层循环中，还进行了常数次的计算和更新操作，它们的时间复杂度可以忽略不计。

因此，凸多边形最优三角剖分算法的时间复杂度为 $O(n^3)$，在顶点数较小的情况下，算法能够在合理的时间范围内求解最优三角剖分问题。

在点的数量比较大时，采用 $O(n^3)$ 的算法会花费较多的时间。下面将提出一种凸多边形最优三角剖分问题的近似算法，该算法采用了贪心策略和随机选择的方法。首先，算法初始化了一些必要的变量和数据结构。然后，从小到大枚举区间长度和左端点，利用随机选择的 $k$ 个分割点对权值进行近似计算。在计算过程中，算法根据当前的最优解更新策略来更新最小权值和对应的分割点。最后，算法输出凸多边形的最优三角剖分的最小权值。

该算法的时间复杂度为 $O(n^3)$，其中 $n$ 是基站列表的长度。算法中的两个嵌套循环分别对应了区间长度和左端点的枚举，因此需要 $O(n^2)$ 的时间复杂度。在每个区间和左端点的循环中，算法通过随机选择的 $k$ 个分割点对权值进行近似计算，这需要 $O(k)$ 的时间复杂度。因此，总的时间复杂度为 $O(n^2 \cdot k)$，由于 $k \leq 5$，所以时间复杂度为 $O(n^2)$

下面是凸多边形最优三角剖分问题的近似算法的伪代码：

**Algorithm 10** 凸多边形最优三角剖分近似算法

**procedure** OPTIMALTRIANGULATIONAPPROX(stations, k)
    $n \leftarrow$ len$(stations)$
    $dp \leftarrow$ [[0 for _ in range$(n)$] for _ in range$(n)$]  ▷ n × n 二维列表，初始化为 0
    $s \leftarrow$ [[0 for _ in range$(n)$] for _ in range$(n)$]  ▷ n × n 二维列表，初始化为 0
    **for** $i$ in range$(n-1)$ **do**
        $dp[i][i+1] \leftarrow 0$
    **for** len_ in range$(3, n+1)$ **do**
        **for** $l$ in range$(n -$ len_ $+ 1)$ **do**
            $r \leftarrow l +$ len_ $- 1$
            ▷ 贪心策略，直接从随机选择的 k 个分割点中选取最好的分割点，范围 [l+1,r-1]
            **if** $r - l - 1 > k$ **then**
                $split\_points \leftarrow$ random_choice$(\{l+1, l+2, ..., r-1\}, k,$ replace $=$ False)
            **else**  ▷ 如果区间长度小于等于 k，则选取所有的分割点
                $split\_points \leftarrow \{l+1, l+2, ..., r-1\}$
            **for** $k$ in split_points **do**
                **if** $dp[l][r] == 0$ or $dp[l][r] > dp[l][k] + dp[k][r] + weight$ **then**
                    $dp[l][r] \leftarrow dp[l][k] + dp[k][r] + weight$
                    $s[l][r] \leftarrow k$
    **调用** $draw\_stations(stations, s,$ title $=$ "Optimal Triangulation Approx") ▷ 画图展示最优三角剖分
    **print** $dp[0][n-1]$

### 2.4.4 运行结果

```
21个基站凸多边形数据
0 19 20
0 7 19
0 6 7
0 1 6
1 5 6
1 2 5
2 4 5
2 3 4
7 8 19
8 18 19
8 9 18
9 17 18
9 13 17
9 12 13
9 11 12
```

```
9 10 11
13 16 17
13 15 16
13 14 15
最优三角剖分的最小权值150529.07457599998
0 19 20
0 7 19
0 6 7
0 1 6
1 5 6
1 2 5
2 4 5
2 3 4
7 8 19
8 18 19
8 9 18
9 17 18
9 13 17
9 12 13
9 11 12
9 10 11
13 16 17
13 15 16
13 14 15
最优三角剖分的近似最小权值150529.07457599998
```

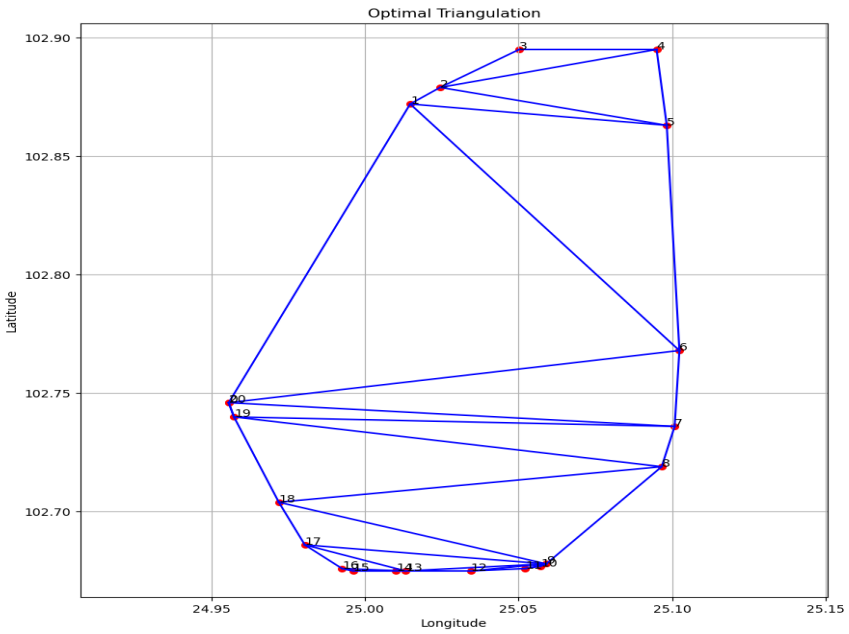


**图 1:** 21 个基站最优解

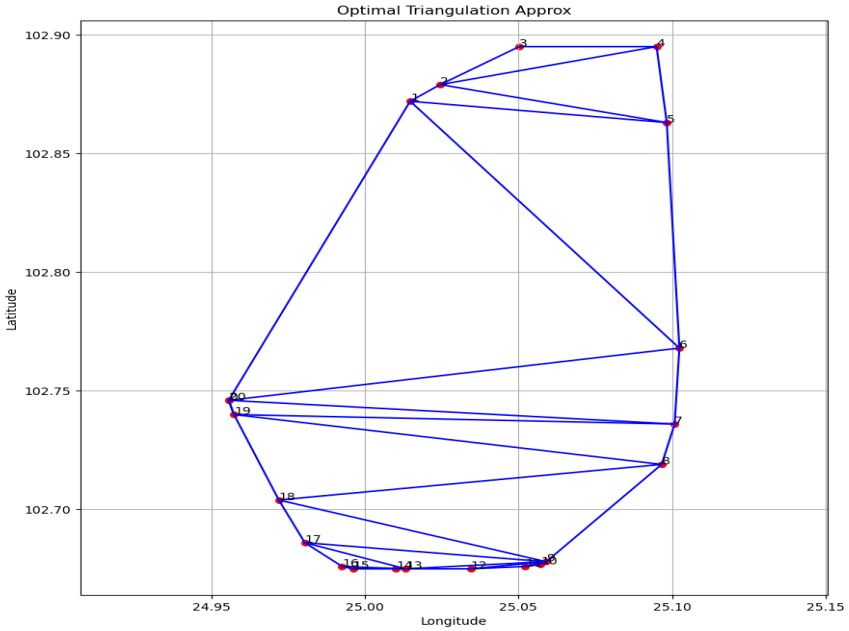


**图 2:** 21 个基站近似解

```
29个基站凸多边形数据
0 1 28
1 11 28
1 9 11
1 2 9
2 3 9
3 8 9
3 4 8
4 7 8
4 5 7
5 6 7
9 10 11
11 12 28
12 13 28
13 27 28
13 15 27
13 14 15
15 24 27
15 16 24
16 23 24
16 17 23
17 22 23
17 19 22
17 18 19
19 21 22
19 20 21
```

```
24 26 27
24 25 26
最优三角剖分的最小权值93108.00227200001
0 1 28
1 11 28
1 9 11
1 2 9
2 3 9
3 8 9
3 4 8
4 7 8
4 5 7
5 6 7
9 10 11
11 12 28
12 13 28
13 27 28
13 15 27
13 14 15
15 24 27
15 16 24
16 23 24
16 17 23
17 22 23
17 19 22
17 18 19
19 21 22
19 20 21
24 26 27
24 25 26
最优三角剖分的近似最小权值93108.00227200001
```

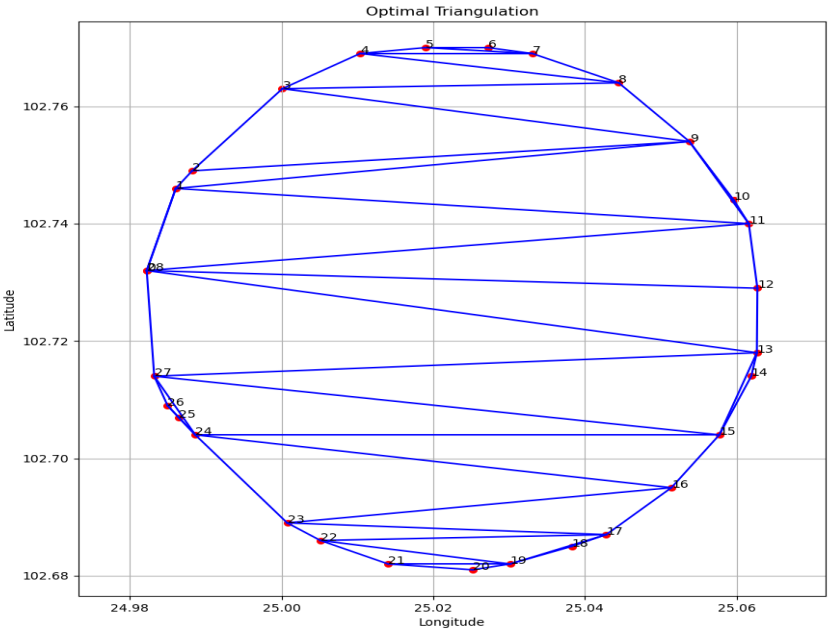


**图 3:** 29 个基站最优解

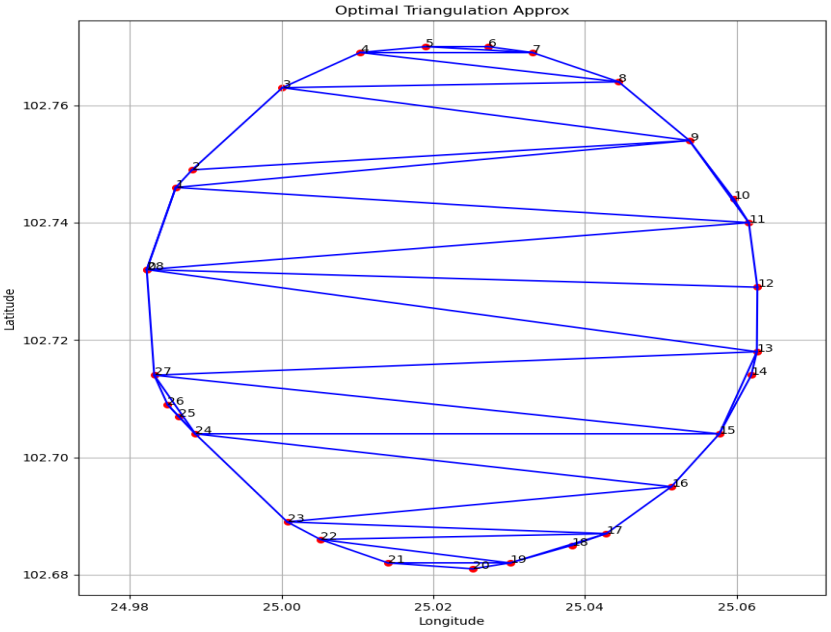


**图 4:** 29 个基站近似解

### 2.4.5 结论

凸多边形最优三角剖分问题是一个经典的计算机图形学问题，动态规划算法提供了一种有效的解决方案。通过填表和状态转移的方式，算法可以在 $O(n^3)$ 的时间复杂度内求解最优三角剖分的最小权值。通过对算法的分析和实验结果的验证，可以得出结论：凸多边形最优三角剖分算法能够正确地找到凸多边形的最优三角剖分，使得剖分后的三角形的权值之和最小。

此外，基于近似算法的凸多边形最优三角剖分解法，通过贪心策略和随机选择的方法，近似计算凸多边形的最优三角剖分的最小权值。算法的时间复杂度为 $O(n^2)$，其中 $n$ 是基站列表的长度。通过实验和分析，可以得出该算法在给定的时间复杂度范围内，能够得到较好的近似解。

## 2.5 0-1 背包问题

### 2.5.1 介绍

0-1 背包问题是一个经典的组合优化问题，它的目标是在给定的一组物品中，选取一些物品放入容量有限的背包中，使得选取的物品总重量不超过背包的容量，并且选取的物品总价值最大化。该问题的名称来源于每个物品只能选择放入背包一次（0 表示不放入，1 表示放入）的限制条件。

在实际应用中，0-1 背包问题有着广泛的应用，例如资源分配、货物装载、投资组合等领域。

### 2.5.2 算法描述

0-1 背包问题的算法采用动态规划的方法进行求解。算法的时间复杂度主要取决于物品数量 $n$ 和背包容量 $m$。

在算法中，使用一个二维数组 $f$ 来保存前 $i$ 个物品放入容量为 $j$ 的背包中的最大价值。算法的核心部分是两个嵌套的循环，这两个循环分别遍历物品和背包容量，每个内部循环的时间复杂度为 $O(1)$。因此，算法的总时间复杂度为 $O(nm)$。

以下是解决 0-1 背包问题的伪代码：

**Algorithm 11** 0-1 背包问题

```
1: procedure ZeroOneKnapsack(capacity, weights, values)
2:     n ← Length(weights)                                  ▷ 物品数量
3:     m ← capacity                                         ▷ 背包容量
4:     f ← Create2DArray(n + 1, m + 1)                      ▷ 初始化二维数组
5:     for i ← 1 to n do
6:         for j ← 1 to m do
7:             if weights[i − 1] ≤ j then
8:                 f[i][j] ← max(f[i − 1][j], f[i − 1][j − weights[i − 1]] + values[i − 1])
9:             else
10:                f[i][j] ← f[i − 1][j]
11:    x ← CreateArray(n)                                   ▷ 初始化最优解列表
12:    j ← m
13:    for i ← n downto 1 do
14:        if f[i][j] > f[i − 1][j] then
15:            x[i − 1] ← 1
16:            j ← j − weights[i − 1]
17:    return f[n][m], x                                    ▷ 返回最大价值和最优解
```

### 2.5.3 分析和改进

在原始的动态规划算法中，使用一个二维数组 $f$ 来保存前 $i$ 个物品放入容量为 $j$ 的背包中的最大价值。算法的核心部分是两个嵌套的循环，这两个循环分别遍历物品和背包容量，每个内部循环的时间复杂度为 $O(1)$。因此，算法的总时间复杂度为 $O(nm)$。

在空间复杂度方面，需要额外的二维数组 $f$ 来保存中间结果，其大小为 $(n+1) \times (m+1)$。因此，算法的空间复杂度为 $O(nm)$。

然而，如果不需要输出具体的物品放入情况，只关心最大的物品总价值，可以进一步优化空间复杂度。优化后的算法只使用一个一维数组 $f$ 来保存背包容量从 0 到 $m$ 的最大价值。这样，算法的空间复杂度可以降低到 $O(m)$。

以下是优化后的伪代码：

**Algorithm 12** 0-1 背包问题（优化后）

```
procedure ZeroOneKnapsackSavingSpace(capacity, weights, values)
    n ← Length(weights)                                  ▷ 物品数量
    m ← capacity                                         ▷ 背包容量
    f ← CreateArray(m + 1)                               ▷ 初始化一维数组
    for i ← 1 to n do
        for j ← m downto 1 do
            if weights[i − 1] ≤ j then
                f[j] ← max(f[j], f[j − weights[i − 1]] + values[i − 1])
    return f[m]                                          ▷ 返回最大价值
```

### 2.5.4 运行结果

```
第1组，最大价值为： 1181
第1个物品放入背包中，重量为14，价值为50
第2个物品放入背包中，重量为11，价值为72
第4个物品放入背包中，重量为17，价值为69
第8个物品放入背包中，重量为26，价值为59
第9个物品放入背包中，重量为10，价值为49
第17个物品放入背包中，重量为37，价值为74
第18个物品放入背包中，重量为19，价值为71
第23个物品放入背包中，重量为13，价值为63
第24个物品放入背包中，重量为15，价值为59
第25个物品放入背包中，重量为9，价值为48
第26个物品放入背包中，重量为10，价值为41
第32个物品放入背包中，重量为8，价值为50
第33个物品放入背包中，重量为11，价值为48
第38个物品放入背包中，重量为8，价值为51
第43个物品放入背包中，重量为28，价值为72
第44个物品放入背包中，重量为16，价值为46
第45个物品放入背包中，重量为9，价值为41
第49个物品放入背包中，重量为23，价值为52
第50个物品放入背包中，重量为18，价值为77
第51个物品放入背包中，重量为38，价值为89
背包总重量为： 340
最优解正确

第2组，最大价值为： 1661
第1个物品放入背包中，重量为10，价值为61
第5个物品放入背包中，重量为33，价值为61
第6个物品放入背包中，重量为44，价值为79
第9个物品放入背包中，重量为15，价值为59
第11个物品放入背包中，重量为12，价值为30
第22个物品放入背包中，重量为20，价值为74
```

```
第23个物品放入背包中，重量为18，价值为46
第27个物品放入背包中，重量为28，价值为51
第31个物品放入背包中，重量为24，价值为64
第34个物品放入背包中，重量为31，价值为54
第42个物品放入背包中，重量为26，价值为54
第49个物品放入背包中，重量为9，价值为19
第52个物品放入背包中，重量为18，价值为67
第54个物品放入背包中，重量为42，价值为73
第55个物品放入背包中，重量为15，价值为44
第58个物品放入背包中，重量为11，价值为36
第60个物品放入背包中，重量为18，价值为49
第61个物品放入背包中，重量为8，价值为79
第70个物品放入背包中，重量为45，价值为77
第77个物品放入背包中，重量为18，价值为74
第80个物品放入背包中，重量为20，价值为46
第81个物品放入背包中，重量为11，价值为35
第82个物品放入背包中，重量为23，价值为46
第83个物品放入背包中，重量为26，价值为73
第84个物品放入背包中，重量为9，价值为39
第88个物品放入背包中，重量为36，价值为74
第89个物品放入背包中，重量为24，价值为67
第95个物品放入背包中，重量为30，价值为80
第100个物品放入背包中，重量为25，价值为50
背包总重量为：  649
最优解正确

第1组，最大价值为：  1181
第2组，最大价值为：  1661
```

#### 2.5.5 结论

0-1 背包问题是一个经典的组合优化问题，在实际应用中有着广泛的应用。通过使用动态规划算法，可以高效地求解 0-1 背包问题，找到最大的物品总价值并确定最优解。

上述内容介绍了 0-1 背包问题的定义和应用，并给出了原始算法和优化后算法的伪代码和分析。通过实现优化后的算法，可以根据给定的物品重量和价值，以及背包的容量，求解出最大的物品总价值。优化后的算法的时间复杂度为 $O(nm)$，空间复杂度为 $O(m)$，在实际问题中具有较高的效率和可行性。

## 3 附录：完整代码

### 3.1 最长公共子序列

```
#include <iostream>
#include <cstdio>
```

```cpp
#include <cstring>
using namespace std;

char *A, *B, *C, *D;
// 求最大值
int Max(int a, int b)
{
    return a > b ? a : b;
}

// 读取文件
void input()
{
    FILE *file = freopen("input1.txt", "r", stdin);
    if (file)
    {
        const int bufferSize = 1500;
        char *line = new char[bufferSize];
        char currentChar = '\0';
        while (fgets(line, bufferSize, file))
        {
            // printf("line is %s\n", line);
            if (line[0] != '\n' && line[0] != '␣')
            {
                int len = strlen(line);
                if (line[1] == ':')
                {
                    currentChar = line[0];
                }
                else
                {
                    line[len - 1] = '\0'; // 去掉换行符
                    switch (currentChar)
                    {
                    case 'A':
                        A = new char[len];
                        strcpy(A, line);
                        // printf("%s\n", A);
                        break;
                    case 'B':
                        B = new char[len];

                        strcpy(B, line);
                        // printf("%s\n", B);
                        break;
                    case 'C':
                        C = new char[len];
```

```
                        strcpy(C, line);
                        // printf("%s\n", C);
                        break;
                    case 'D':
                        D = new char[len];
                        strcpy(D, line);
                        // printf("%s\n", D);
                        break;
                    default:
                        break;
                    }
                }
            }
        }
        delete[] line;
    }
    fclose(stdin);
}

// 递归输出LCS
void LCS(int i, int j, char *X, int **path)
{
    if (i == 0 || j == 0)
        return;
    if (path[i][j] == 1)
    {
        LCS(i - 1, j - 1, X, path);
        printf("%c", X[i]);
    }
    else if (path[i][j] == 2)
        LCS(i - 1, j, X, path);
    else
        LCS(i, j - 1, X, path);
}
// PPT上的算法,空间复杂度O(mn)
void LCSLength(char *X, char *Y)
{
    int m = strlen(X);
    int n = strlen(Y);
    int **dp = new int *[m + 1];
    int **path = new int *[m + 1];
    for (int i = 0; i <= m; i++)
    {
        dp[i] = new int[n + 1];
        path[i] = new int[n + 1];
        memset(dp[i], 0, sizeof(int) * (n + 1));
        memset(path[i], 0, sizeof(int) * (n + 1));
```

```
    }
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
                if (X[i] == Y[j])
                {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                    path[i][j] = 1;
                }
                else if (dp[i - 1][j] >= dp[i][j - 1])
                {
                    dp[i][j] = dp[i - 1][j];
                    path[i][j] = 2;
                }
                else
                {
                    dp[i][j] = dp[i][j - 1];
                    path[i][j] = 3;
                }
        }
    }
    LCS(m, n, X, path);
    printf("\nLCSLength␣is:␣%d\n", dp[m][n]);
    for (int i = 0; i <= m; i++)
    {
        delete[] path[i];
        delete[] dp[i];
    }
    delete[] dp;
    delete[] path;
}

// 只计算长度,空间复杂度O(n)
void LCS_Spacesaving(char *X, char *Y)
{
    int m = strlen(X);
    int n = strlen(Y);
    int *dp = new int[n + 1];
    for (int i = 0; i <= n; i++)
    {
        dp[i] = 0;
    }
    for (int i = 1; i <= m; i++)
    {
        int upLeft = dp[0]; // 保存左上角的值
        for (int j = 1; j <= n; j++)
```

```
        {
            int temp = dp[j]; // 保存当前位置的值,即dp[i-1][j],因为dp[j]已经被更新
                了
            if (X[i] == Y[j])
            {
                dp[j] = upLeft + 1;
            }
            else
            {
                dp[j] = Max(dp[j], dp[j - 1]);
            }
            upLeft = temp; // 更新左上角的值
        }
    }
    printf("LCSLength is: %d\n", dp[n]);
    delete[] dp;
}

int main()
{
    input();
    printf("A and B LCS:\n");
    LCSLength(A, B);
    LCS_Spacesaving(A, B);

    printf("\nA and C LCS:\n");
    LCSLength(C, D);
    LCS_Spacesaving(C, D);

    printf("\nA and D LCS:\n");
    LCSLength(A, D);
    LCS_Spacesaving(A, D);

    printf("\nC and B LCS:\n");
    LCSLength(C, B);
    LCS_Spacesaving(C, B);
    return 0;
}
```

## 3.2 最长不上升子序列

```
#include <cstdio>
#include <cstring>
#include <vector>
#include <functional>
#include <algorithm>
```

```
using namespace std;
// 从文件中读取数据
vector<int> input(const char fileName[])
{
    FILE *file = freopen(fileName, "r", stdin);
    vector<int> result;
    if (file)
    {
        int line;
        while (scanf("%d", &line) != EOF)
        {
            result.push_back(line);
        }
    }
    fclose(stdin);
    return result;
}

// 借助最长公共子序列求最长不上升子序列
void LDS_by_LCS(vector<int> num)
{
    int n = num.size();
    vector<int> num_temp = num;
    sort(num.begin(), num.end(), greater<int>());
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
    vector<vector<int>> path(n + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; ++j)
        {
            if (num[i - 1] == num_temp[j - 1])
            {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                path[i][j] = 1;
            }
            else if (dp[i - 1][j] >= dp[i][j - 1])
            {
                dp[i][j] = dp[i - 1][j];
                path[i][j] = 2;
            }
            else
            {
                dp[i][j] = dp[i][j - 1];
                path[i][j] = 3;
            }
        }
    }
```

```
    printf("最长不上升子序列长度为：%d\n", dp[n][n]);
    printf("最长不上升子序列为：");
    typedef function<void(int, int)> PrintFunction;
    PrintFunction output = [&](int i, int j)
    {
        if (i == 0 || j == 0)
            return;
        if (path[i][j] == 1)
        {
            output(i - 1, j - 1);
            printf("%d␣", num[i - 1]);
        }
        else if (path[i][j] == 2)
            output(i - 1, j);
        else
            output(i, j - 1);
    };
    output(n, n);
    printf("\n");
}

// 最长不上升子序列
void LDS(vector<int> num)
{
    int n = num.size();
    vector<int> dp(n, 1);  // dp[i]表示以num[i]结尾的最长不上升子序列的长度
    vector<int> path(n, -1);
    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j <= i - 1; j++)
        {
            if (num[i] <= num[j] && dp[i] < dp[j] + 1)
            {
                dp[i] = dp[j] + 1;
                path[i] = j;
            }
        }
    }
    int ans = dp[0];
    int index = 0;
    for (int i = 0; i < n; i++)
    {
        if (ans < dp[i])
        {
            ans = dp[i];
            index = i;
        }
```

```
    }
    printf("最长不上升子序列长度为: %d\n", ans);
    printf("最长不上升子序列为: ");
    typedef function<void(int)> PrintFunction;
    PrintFunction output = [&](int x)
    {
        if (x == -1)
            return;
        output(path[x]);
        printf("%d␣", num[x]);
    };
    output(index);
    printf("\n");
}

// 只计算长度,时间复杂度O(nlogn)
void LDS_nlogn(vector<int> num)
{
    int n = num.size();
    // dp[i]表示长度为i的最长不上升子序列的最大末尾元素
    vector<int> dp(n + 1, INT_MAX);
    int len = 0;
    for (int i = 0; i < n; i++)
    {
        // 如果num[i] 最长不上升子序列的最大末尾元素, 那么num[i]可以接在最长不上升
            子序列后面, 形成一个更长的不上升子序列
        if (num[i] <= dp[len])
        {
            dp[++len] = num[i];
        }
        else // 否则, 找到dp中第一个小于num[i]的元素, 用num[i]替换它
        {
            int pos = upper_bound(dp.begin() + 1, dp.begin() + len + 1, num[i],
                greater<int>()) - dp.begin();
            // printf("dp[pos-1] = %d, dp[pos] = %d, dp[pos+1] = %d\n", dp[pos -
                1], dp[pos], dp[pos + 1]);
            // printf("num[i] = %d\n", num[i]);
            dp[pos] = num[i];
        }
    }
    printf("最长不上升子序列长度为: %d\n", len);
}

int main()
{
    vector<int> num1 = input("input2_1.txt");
    vector<int> num2 = input("input2_2.txt");
```

```
    printf("num1:\n");
    LDS_by_LCS(num1);
    LDS(num1);
    LDS_nlogn(num1);
    printf("\nnum2:\n");
    LDS_by_LCS(num2);
    LDS(num2);
    LDS_nlogn(num2);
    return 0;
}
```

### 3.3 最大子段和

```
#include <cstdio>
#include <iostream>
#include <vector>

using namespace std;
// 从文件中读取数据
vector<int> input(const char fileName[])
{
    FILE *file = freopen(fileName, "r", stdin);
    vector<int> result;
    if (file)
    {
        int line;
        while (scanf("%d", &line) != EOF)
        {
            result.push_back(line);
        }
    }
    fclose(stdin);
    return result;
}

// 最大子段和，分治法，O(nlogn)
int MaxSubSum(vector<int> num, int left, int right)
{
    if (left == right)
    {
        return num[left] > 0 ? num[left] : 0;
    }
    int mid = (left + right) / 2;
    int leftMax = MaxSubSum(num, left, mid);
    int rightMax = MaxSubSum(num, mid + 1, right);
    int leftBorderMax = num[mid];
```

```
    int leftBorderSum = 0;
    for (int i = mid; i >= left; i--)
    {
        leftBorderSum += num[i];
        if (leftBorderSum > leftBorderMax)
        {
            leftBorderMax = leftBorderSum;
        }
    }
    int rightBorderMax = num[mid + 1];
    int rightBorderSum = 0;
    for (int i = mid + 1; i <= right; i++)
    {
        rightBorderSum += num[i];
        if (rightBorderSum > rightBorderMax)
        {
            rightBorderMax = rightBorderSum;
        }
    }
    int borderMax = leftBorderMax + rightBorderMax;
    return max(max(leftMax, rightMax), borderMax);
}
// 最大子段和，分治法，重载隐藏参数
int MaxSubSum(vector<int> num)
{
    return MaxSubSum(num, 0, num.size() - 1);
}

// 最大子序列和，动态规划，O(n)
void MaxSum(vector<int> num)
{
    int n = num.size();
    int dp = num[0], ans = num[0];
    int left = 0, right = 0;
    for (int i = 1; i < n; i++)
    {
        if (dp > 0)
        {
            dp += num[i];
        }
        else
        {
            dp = num[i];
            left = i;
        }
        if (dp > ans)
        {
```

```
            ans = dp;
            right = i;
        }
    }
    printf("最大子序列和为: %d\n", ans);
    printf("最大子序列为: ");
    for (int i = left; i <= right; i++)
    {
        printf("%d␣", num[i]);
    }
    printf("\n");
}

int main()
{
    vector<int> num1 = input("input2_1.txt");
    vector<int> num2 = input("input2_2.txt");

    printf("num1最大子段和为:␣%d\n", MaxSubSum(num1));
    printf("num1");
    MaxSum(num1);

    printf("\nnum2最大子段和为:␣%d\n", MaxSubSum(num2));
    printf("num2");
    MaxSum(num2);
    return 0;
}
```

### 3.4 凸多边形最优三角剖分

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


def degree_to_radian(degree):
    """
    将给定的经纬度转化为弧度
    :param degree: 1个输入(经纬度)
    :return: 转化为的弧度
    """
    return degree * np.pi / 180


def calculate_distance(lat1, lon1, lat2, lon2):
    """
```

```
    距离公式
    :param lat1: 纬度1
    :param lon1: 经度1
    :param lat2: 纬度2
    :param lon2: 经度2
    :return: 距离/m,保留6位小数
    """
    if abs(lat1 - lat2) < 1e-6 and abs(lon1 - lon2) < 1e-6:
        return 0
    rad_lat1 = degree_to_radian(lat1)
    rad_lon1 = degree_to_radian(lon1)
    rad_lat2 = degree_to_radian(lat2)
    rad_lon2 = degree_to_radian(lon2)
    # R为赤道半径/m
    R = 6378.137 * 1000
    dis = R * np.arccos(
        np.cos(rad_lat1) * np.cos(rad_lat2) * np.cos(rad_lon1 - rad_lon2)
        + np.sin(rad_lat1) * np.sin(rad_lat2)
    )
    return round(dis, 6)


class Station:
    """
    基站类
    """

    def __init__(self, eNodeB_id, latitude, longitude, number):
        """
        基站类的初始化
        :param eNodeB_id:  基站编号
        :param latitude:   x坐标
        :param longitude:  y坐标
        :param number:   逆时针顺序标号
        """
        self.eNodeB_id = eNodeB_id
        self.x = longitude
        self.y = latitude
        self.number = number


def initialize_stations(path):
    """
    初始化基站
    :param path: 基站数据文件路径
    :return: 基站列表
    """
```

```
    data = pd.read_excel(path, sheet_name=1)
    # 删除未命名的列和行
    data = data.loc[:, ~data.columns.str.contains("^Unnamed")]
    data = data.dropna(axis=0, how="any")

    eNodeB_id = data["ENODEBID"].tolist()
    longitude = data["LONGITUDE"].tolist()
    latitude = data["LATITUDE"].tolist()
    number = data["逆时针顺序标号"].tolist()
    stations = [
        Station(eNodeB_id, lon, lat, num)
        for eNodeB_id, lon, lat, num in zip(eNodeB_id, longitude, latitude, number)
    ]
    return stations


def draw_stations(stations, s, title="Optimal Triangulation"):
    """
    绘制基站图
    :param stations: 基站列表
    :param s: s[i][j]表示从i到j的最优三角剖分的最小权值对应的k
    """
    n = len(stations)
    # 画图展示最优三角剖分
    plt.figure(figsize=(10, 10))
    plt.xlim(
        min([station.x for station in stations]) - 0.01,
        max([station.x for station in stations]) + 0.01,
    )
    plt.ylim(
        min([station.y for station in stations]) - 0.01,
        max([station.y for station in stations]) + 0.01,
    )
    plt.title(title)
    plt.xlabel("Longitude")
    plt.ylabel("Latitude")
    plt.grid()
    plt.axis("equal")
    # 画出基站的位置
    for station in stations:
        plt.scatter(station.x, station.y, c="r", s=30)
        plt.text(station.x, station.y, int(station.number), fontsize=10)

    # 递归打印最优三角剖分
    def print_optimal_triangulation(l, r):
        if l + 1 >= r:  # 递归边界
            return
```

```
        print(l, s[l][r], r)
        plt.plot(  # 连接l和k=s[l][r]
            [stations[l].x, stations[s[l][r]].x],
            [stations[l].y, stations[s[l][r]].y],
            c="b",
        )
        plt.plot(  # 连接k=s[l][r]和r
            [stations[s[l][r]].x, stations[r].x],
            [stations[s[l][r]].y, stations[r].y],
            c="b",
        )
        print_optimal_triangulation(l, s[l][r])
        print_optimal_triangulation(s[l][r], r)

    print_optimal_triangulation(0, n - 1)
    plt.show()


# 凸多边形最优三角剖分问题
def optimal_triangulation(stations):
    """
    凸多边形最优三角剖分问题
    :param stations: 基站列表
    :return: 最优三角剖分的最小权值
    """
    n = len(stations)
    # dp[i][j]表示从i到j的最优三角剖分的最小权值
    dp = [[0 for _ in range(n)] for _ in range(n)]
    # s[i][j]表示从i到j的最优三角剖分的最小权值对应的k
    s = [[0 for _ in range(n)] for _ in range(n)]
    # 初始化dp[i][i+1]
    for i in range(n - 1):
        dp[i][i + 1] = 0
    # 从小到大枚举区间长度
    for len_ in range(3, n + 1):
        # 从小到大枚举左端点
        for l in range(n - len_ + 1):  # l + len_ - 1 < n
            r = l + len_ - 1
            # 从l+1到r-1枚举k
            for k in range(l + 1, r):
                # 计算三角形lkr的权值
                weight = (
                    calculate_distance(
                        stations[l].y, stations[l].x, stations[k].y, stations[k].x
                    )
                    + calculate_distance(
                        stations[k].y, stations[k].x, stations[r].y, stations[r].x
```

```
                    )
                    + calculate_distance(
                        stations[r].y, stations[r].x, stations[l].y, stations[l].x
                    )
                )
                # 更新dp[l][r]
                if dp[l][r] == 0 or dp[l][r] > dp[l][k] + dp[k][r] + weight:
                    dp[l][r] = dp[l][k] + dp[k][r] + weight
                    s[l][r] = k

    # 画图展示最优三角剖分
    draw_stations(stations, s)

    return dp[0][n - 1]


# 凸多边形最优三角剖分问题,O(n^2)的近似算法
def optimal_triangulation_approx(stations, k):
    """
    凸多边形最优三角剖分问题,O(n^2)的近似算法
    :param stations: 基站列表
    :param k: 随机选取k个点,最大为5
    :return: 最优三角剖分的最小权值
    """
    n = len(stations)
    # dp[i][j]表示从i到j的最优三角剖分的最小权值
    dp = [[0 for _ in range(n)] for _ in range(n)]
    # s[i][j]表示从i到j的最优三角剖分的最小权值对应的k
    s = [[0 for _ in range(n)] for _ in range(n)]
    # 初始化dp[i][i+1]
    for i in range(n - 1):
        dp[i][i + 1] = 0
    # 从小到大枚举区间长度
    for len_ in range(3, n + 1):
        # 从小到大枚举左端点
        for l in range(n - len_ + 1):  # l + len_ - 1 < n
            r = l + len_ - 1
            # 贪心策略，直接从随机选择的k个分割点中选取最好的分割点，范围[l+1,r-1]
            # 随机选取k个分割点
            if r - l - 1 > k:
                split_points = np.random.choice(range(l + 1, r), k, replace=False)
            else:  # 如果区间长度小于等于k，则选取所有的分割点
                split_points = range(l + 1, r)
            # 选取最好的分割点
            for k in split_points:
                # 计算三角形lkr的权值
                weight = (
```

```
                    calculate_distance(
                        stations[l].y, stations[l].x, stations[k].y, stations[k].x
                    )
                    + calculate_distance(
                        stations[k].y, stations[k].x, stations[r].y, stations[r].x
                    )
                    + calculate_distance(
                        stations[r].y, stations[r].x, stations[l].y, stations[l].x
                    )
                )
                # 更新dp[l][r]
                if dp[l][r] == 0 or dp[l][r] > dp[l][k] + dp[k][r] + weight:
                    dp[l][r] = dp[l][k] + dp[k][r] + weight
                    s[l][r] = k

    draw_stations(stations, s, title="Optimal Triangulation Approx")

    return dp[0][n - 1]


if __name__ == "__main__":
    print('21个基站凸多边形数据')
    stations = initialize_stations(
        r"C:\Users\Administrator\Desktop\算法设计与分析-编程作业-第3章-动态规划
            -2023-301-304\附件3-1.21个基站凸多边形数据-2023.xls"
    )
    print('最优三角剖分的最小权值{}'.format(optimal_triangulation(stations)))
    print('最优三角剖分的近似最小权值{}'.format(optimal_triangulation_approx(
        stations, 5)))

    print('\n29个基站凸多边形数据')
    stations = initialize_stations(
        r"C:\Users\Administrator\Desktop\算法设计与分析-编程作业-第3章-动态规划
            -2023-301-304\附件3-2.29个基站凸多边形数据-2023.xls"
    )
    print('最优三角剖分的最小权值{}'.format(optimal_triangulation(stations)))
    print('最优三角剖分的近似最小权值{}'.format(optimal_triangulation_approx(
        stations, 5)))
```

## 3.5 0-1 背包

```
import re

path = r"C:\Users\Administrator\Desktop\算法设计与分析-编程作业-第3章-动态规划
    -2023-301-304\附件4.0-1背包问题输入数据-2023.txt"
```

```
def read_file(group):
    """
    读取文本文件内容
    :param group: 选择要读取的组，1表示第一组，2表示第二组
    :return:  背包容量，物品重量列表，物品价值列表
    """
    # 读取文本文件内容
    with open(path, "r") as file:
        content = file.read()

    # 根据参数group选择要读取的组
    if group == 1:
        start_index = content.index("第一组") + len("第一组") + 1
        end_index = content.index("第二组")
    elif group == 2:
        start_index = content.index("第二组") + len("第二组") + 1
        end_index = len(content)
    else:
        raise ValueError("Invalid group number. Must be 1 or 2.")

    # 提取容量、重量和价值数据
    group_data = content[start_index:end_index].strip().split("\n\n")

    # 提取容量
    re_pattern = r"(\d+)"  # 正则表达式，匹配数字
    capacity = int(re.findall(re_pattern, group_data[0])[0])

    # 提取重量和价值列表
    weights = re.findall(re_pattern, group_data[1])
    weights = [int(weight) for weight in weights]
    values = re.findall(re_pattern, group_data[2])
    values = [int(value) for value in values]
    return capacity, weights, values


def zero_one_knapsack(capacity, weights, values):
    """
    0-1背包问题
    :param capacity: 背包容量
    :param weights: 物品重量列表
    :param values: 物品价值列表
    :return: 最大价值和最优解，最优解为一个列表，x[i] = 1表示第 i + 1 个物品放入背
        包中
    """
    # 初始化二维数组
    n = len(weights)  # 物品数量
```

```
    m = capacity  # 背包容量
    f = [
        [0 for _ in range(m + 1)] for _ in range(n + 1)
    ]  # f[i][j]表示前i个物品放入容量为j的背包中的最大价值

    # 动态规划
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if weights[i - 1] <= j:
                f[i][j] = max(f[i - 1][j], f[i - 1][j - weights[i - 1]] + values[i
                    - 1])
            else:
                f[i][j] = f[i - 1][j]

    # 逆推出最优解，x[i] = 1表示第 i + 1 个物品放入背包中
    x = [0 for _ in range(n)]
    j = m
    for i in range(n, 0, -1):
        if f[i][j] > f[i - 1][j]:
            x[i - 1] = 1
            j -= weights[i - 1]
    return f[n][m], x


def solve(num):
    """
    解决背包问题
    :param num: 选择要解决的问题，1表示第一组，2表示第二组
    :return: None
    """
    n, w, v = read_file(num)
    max_value, x = zero_one_knapsack(n, w, v)
    print(f"第{num}组，最大价值为：", max_value)
    sum_weight, check = 0, 0
    for i in range(len(x)):
        if x[i] == 1:
            sum_weight += w[i]
            check += v[i]
            print("第{}个物品放入背包中，重量为{}，价值为{}".format(i + 1, w[i], v[
                i]))
    print("背包总重量为：", sum_weight)

    # 检查最优解是否正确
    if max_value == check:
        print("最优解正确\n")
    else:
        print("最优解错误\n")
```

```python
def zero_one_knapsack_saving_space(capacity, weights, values):
    """
    0-1背包问题，空间复杂度为O(m)
    :param capacity: 背包容量
    :param weights: 物品重量列表
    :param values: 物品价值列表
    :return: 最大价值和最优解，最优解为一个列表，x[i] = 1表示第 i + 1 个物品放入背
        包中
    """
    # 初始化一维数组
    n = len(weights)  # 物品数量
    m = capacity  # 背包容量
    f = [0 for _ in range(m + 1)]  # f[j]表示容量为j的背包中的最大价值

    # 动态规划
    for i in range(1, n + 1):
        for j in range(m, 0, -1):
            if weights[i - 1] <= j:
                f[j] = max(f[j], f[j - weights[i - 1]] + values[i - 1])

    return f[m]


def solve_saving_space(num):
    """
    解决背包问题
    :param num: 选择要解决的问题，1表示第一组，2表示第二组
    :return: None
    """
    n, w, v = read_file(num)
    max_value = zero_one_knapsack_saving_space(n, w, v)
    print(f"第{num}组，最大价值为：", max_value)


if __name__ == "__main__":
    solve(1)
    solve(2)

    solve_saving_space(1)
    solve_saving_space(2)
```