《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：2021211306

姓名：杜抒泽

学号：2021211110

第三章 动态规划

2021211110 杜抒泽

2023 年 12 月 10 日

目录

1 实验内容

1.1 实验要求

1. 利用“附件 1.最长公共子序列输入数据-2023”中给出的字符串 A, B, C, D 分别找出下列两两字符串间的最长公共子串，并输出结果：
   - A - B
   - C - D
   - A - D
   - C - B
2. 利用“附件 2.最大子段和输入数据-序列 1-2023”、“附件 2.最大子段和输入数据-序列 2-2023”，求这两个序列中的最长递减子序列。
3. 针对“附件 2.最大子段和输入数据-序列 1-2023”“附件 2.最大子段和输入数据-序列 2-2023” 中给出的序列 1、序列 2, 分别计算其最大子段和。
   - 要求
     1. 指出最大子段和在原序列中的位置
     2. 给出最大子段和具体值

1.2 题目需求分析

本次试验有三个题目。前两个题目分别要求求字符串的最长公共子序列和整数数列的最长非严格下降子序列，第三个题目要求整数数列的最大子段和。以下把三个题目分别编号为题目 1.1，题目 1.2 和题目 2。

我们将涉及到的所有问题进行形式化表述。

子序列（subsequence）：一个序列 $\{b_k\}$ 是另一个序列 $\{a_n\}$ 的子序列，当且仅当存在一个严格单调递增的序列 $\{i_k\}$，使得 $\forall j \in \{1, 2, \ldots, k\}\left(a_{\{i_j\}} = b_j\right)$。

注意区分子序列与子串（substring）：子序列不要求连续，而子串要求连续。对于 n 个串 s_i（或序列），求它们的最长公共子串是 $\mathcal{O}\left(\sum|s_i|\right)$ 的，而求最长公共子序列是 $\mathcal{O}\left(\prod|s_i|\right)$ 的。

最长公共子序列（Longest Common Subsequence, LCS）问题即，给定两个串或序列 $\{a_n\}$ 和 $\{b_m\}$，求存在的包含元素最多的的序列 $\{c_k\}$，使得 $\{c_k\}$ 既为 $\{a_n\}$ 的子序列，又为 $\{b_m\}$ 的子序列。

最长非严格下降子序列：给定一个序列 $\{a_n\}$，求最长的序列 $\{b_k\}$，使得 $\{b_k\}$ 既为 $\{a_n\}$ 的子序列，又满足 $\forall i \in \{1,2,\ldots,k-1\}\left(b_i \geq b_{\{i+1\}}\right)$。最长上升子序列一般简称为 LIS，但最长下降子序列没有常用的缩写，不过在代码中我还是将其叫做 LDS，毕竟要有个名字。

最大子段和（Maximum Subsegment Sum）：给定一个序列 $\{a_n\}$，求使得 $\sum_{\{i=l\}}^{r} a_i$ 最大的区间端点 l 和 r。

1.3 具体实验内容

具体地，我在本次实验中实现了以下内容：

- 利用 C++（C++20 语言标准）编写能够解决前述三个问题的算法，并对其进行封装，提供易于使用的接口。
- 实现泛型的 $\mathcal{O}\left(n^2\right)$ 最长公共子序列算法，用于求解包括但不限于字符串和整数数列的最长公共子序列。
- 实现基于最长公共子序列算法的 $\mathcal{O}\left(n^2\right)$ 最长非严格下降子序列算法。
- 实现 $\mathcal{O}(n)$ 最大子段和算法。
- 在提供的数据上运行所有算法并得到答案。
- 随机生成大规模数据并测试算法的性能。
- 对所有算法提供计时接口，对每次算法的运行时间进行统计，分析算法的时间复杂度是否符合预期。

2 算法设计

由于我是算法竞赛出身，在讲解算法时可能并不会使用和课上相同的术语，但我保证描述一定干练、正确且规范，表达我已经完全掌握了算法的思想，并且希望读者能够更好地理解算法的本质。

2.1 $\mathcal{O}\left(|s|^2\right)$ 最长公共子序列算法

2.1.1 题目背景与数学抽象

$\mathcal{O}\left(|s|^2\right)$ 最长公共子序列算法是非常经典的动态规划（Dynamic Programming, DP）算法，几乎每个 DP 的学习者都会很快接触到这个问题并学习这个算法。

抽象得到以下的问题：

给定长为 n 的序列 $\{a_n\}$ 和长为 m 的序列 $\{b_m\}$，求它们的最长公共子序列，要求不仅要得到最长公共子序列的长度，还要得到其内容。

2.1.2 算法思考过程

思考一个动态规划算法无非思考两个部分：状态和转移。

简单思考，尝试采用 $n \times m$ 个状态，每个状态表示 $\{a_n\}$ 的前 i 个元素和 $\{b_m\}$ 的前 j 个元素的 LCS 。记 $f_{i,j}$ 表示 $\{a_n\}$ 的前 i 个元素和 $\{b_m\}$ 的前 j 个元素的 LCS 的长度，显然得到边界条件 $f_{0,j} = f_{i,0} = 0$。

有了状态，我们思考如何转移。假设我们已经计算得到了从 $f_{0,0}$ 到 $f_{i-1,j-1}$ 的整个矩阵，那么显然有以下的状态转移方程：

$$f_{i,j} = \begin{cases} f_{i-1,j-1} + 1 & \text{if } a_i = b_j \\ \max\left(f_{i-1,j}, f_{i,j-1}\right) & \text{otherwise} \end{cases}$$

这个状态转移方程的意义是，对于状态 (i, j)，如果 $a_i = b_j$，那么它们对 LCS 可能产生贡献，我们在当前状态将 a_i（或 b_j）加入状态 $(i-1, j-1)$ 求得的 LCS 中，此时 LCS 的长度是 $f_{i-1,j-1} + 1$；否则，a_i 和 b_j 至少有一个不在 LCS 中，此时的 LCS 是状态 $(i-1, j)$ 和 $(i, j-1)$ 求得的 LCS 中较长的一个，其长度是 $f_{i-1,j}$ 和 $f_{i,j-1}$ 中的较大值。

这个状态转移方程的正确性是显然的。我们有 $\mathcal{O}(nm)$ 个状态，每个状态需要花费 $\mathcal{O}(1)$ 的时间转移，因此总时间复杂度是 $\mathcal{O}(nm)$ 的，同时需要 $\mathcal{O}(nm)$ 的额外空间。

考虑时间和空间的优化。发现时间已经没办法再压缩了，因为无论去掉哪一个状态都会使得转移时信息不全。不过空间可以优化。我们发现，每次转移只需要用到上一行和当前行（或上一列和当前列）的信息，因此我们可以只维护两行（或两列）的信息，而不是整个矩阵。利用滚动数组的方法，额外空间可以被优化为 $\mathcal{O}(\min(m, n))$ 的。

别急！以上的空间优化的策略是基于我们只需要得到 LCS 长度而言的。我们现在还没有实现求得 LCS 内容的算法。

在 DP 时求内容通常采用在转移时记录转移来源的手法。我们增加一个数组 $dir_{i,j}$，每个元素属于 Left, Up, LeftUp 之一。$dir_{i,j}$ 表示状态 (i, j) 的转移来源。我们在转移时记录转移来源，最后从 $f_{n,m}$ 开始回溯，即可得到 LCS 的内容。dir 的转移方程如下：

$$dir_{i,j} = \begin{cases} \text{LeftUp} & \text{if } a_i = b_j \\ \text{Left} & \text{if } a_i \neq b_j \wedge f_{i-1,j} > f_{i,j-1} \\ \text{Up} & \text{if } a_i \neq b_j \wedge f_{i-1,j} < f_{i,j-1} \end{cases}$$

回溯的过程如下：

- 初始时，令 $i \leftarrow n$，$j \leftarrow m$。

- 每当到达了一个位置，若 $i = 0$ 或 $j = 0$，代表已经拿到了 LCS 的最前一个位置，结束回溯。否则：
  - 如果 $dir_{i,j} = \text{LeftUp}$，那么 $a_i = b_j$，同时将 a_i 加入答案序列的**最前面**，同时令 $i \leftarrow i - 1$，$j \leftarrow j - 1$。
  - 如果 $dir_{i,j} = \text{Left}$，那么 $a_i \neq b_j$，同时令 $i \leftarrow i - 1$。
  - 如果 $dir_{i,j} = \text{Up}$，那么 $a_i \neq b_j$，同时令 $j \leftarrow j - 1$。

回溯的过程是 $\mathcal{O}(n + m)$ 的，因此总时间复杂度是 $\mathcal{O}(nm)$ 的，同时需要 $\mathcal{O}(nm)$ 的额外空间。由于回溯需要的 $\mathcal{O}(n + m)$ 个状态在全部的 $\mathcal{O}(nm)$ 个空间中的位置不可预测，它的空间不可优化。

注意：实际上也可以不记录 dir 数组，直接利用串 a, b 和数组 f 来得到 LCS 的内容。但是这样做的话，f 就不能用滚动数组压维了，所以 $\mathcal{O}(nm)$ 的额外空间仍然是必要的。

由此，我们得到了一个可以求得 LCS 的长度和内容的算法，它需要 $\mathcal{O}(nm)$ 的时间和额外空间。

2.1.3 代码实现

代码实现如下，内存空间的分配等细节略去不表。

`f[i][j]` 和 `dir[i][j]` 的含义与前文中 $f_{i,j}$ 和 $dir_{i,j}$ 相同。其中 `dir` 中元素的类型是 `enum class Direction: std::uint_8 { Left, Up, LeftUp };`。

没有采用滚动数组的手法优化数组 `f` 的空间。

```
namespace LCS {

template<typename T>
std::vector<T> _solve(const std::vector<T> &a, const std::vector<T> &b) {
  int n = a.size() - 1, m = b.size() - 1;
  for (int i = 1; i ⩽ n; ++i)
    for (int j = 1; j ⩽ m; ++j) {
      if (a[i] == b[j]) {
        f[i][j] = f[i - 1][j - 1] + 1;
        dir[i][j] = Direction::LeftUp;
      } else if (f[i - 1][j] > f[i][j - 1]) {
        f[i][j] = f[i - 1][j];
        dir[i][j] = Direction::Up;
      } else {
        f[i][j] = f[i][j - 1];
        dir[i][j] = Direction::Left;
      }
    }
```

```
  std::vector<T> ans;
  for (int i = n, j = m; i && j;) {
    switch (dir[i][j]) {
      case Direction::LeftUp:
        ans.emplace_back(a[i--]);
        /* fallthrough */
      case Direction::Left:
        --j;
        break;
      case Direction::Up:
        --i;
        break;
      default:
        throw std::runtime_error("Should not reach here!");
    }
  }
  std::reverse(ans.begin(), ans.end());
  return ans;
}

} // namespace LCS
```

2.2 将最长下降子序列问题规约到最长公共子序列问题

2.2.1 题目背景与数学抽象

这个问题实际上非常诡异。最长下降子序列（严格和非严格都）是有非常简单的 $\mathcal{O}(n \log n)$ 做法的，不是很理解为什么强行要规约到一个更难的问题上求解。

不过既然都这么说了，该做还是得做。抽象得到以下的问题：

给定长为 n 的序列 $\{a_n\}$，求它的最长非严格下降子序列的长度和内容。要求将其规约到 LCS 问题，用动态规划求解。

2.2.2 算法思考过程

我们已经实现了 LCS 算法，现在只需要考虑如何规约即可。

这个问题看起来很简单，实际上也很简单。LCS 是两个序列的 LCS，但我们现在只有一个序列啊。另一个序列怎么变出来呢？

一个很朴素的想法是，我们找到原序列里的最大值和最小值，然后创建一个新序列，按顺序包含从最大值递减到最小值的所有数字。这样，原序列的最长下降子序列就是新序列和

原序列的最长公共子序列了。但是细心想想会发现，这样求出来的是最长严格下降子序列[1]，并且复杂度和值域相关，难以接受。更朴素的想法是统计原序列中每个数字出现的次数，然后在新序列中每个数字重复它的出现次数次——等等，这是不是说直接把原序列按降序排序一遍就好了？

对，就这么简单。一个序列和自己的降序排序的 LCS 就是它的最长非严格下降子序列。**以下全部都是和本题题目要求无关的内容，用以拓展。**

能不能再给力一点啊？

好，我们考虑如何求最长严格下降子序列。虽然这个不是题目所求的问题，但是也可以思考一下。大部分朋友应该几秒钟就想出来了，把排序后的序列去重就可以了，就这么简单。

上述的规约可以在 $\mathcal{O}(n \log n)$ 的时间内完成，瓶颈是排序。规约后求 LCS 的时间复杂度是 $\mathcal{O}(n^2)$，因此用求 LCS 的算法求最长严格下降子序列的时间复杂度是 $\mathcal{O}(n^2)$ 的，时间和空间瓶颈都是求 LCS。

和本题要求无关的是，求最长（严格或非严格）下降子序列的 $\mathcal{O}(n \log n)$ 算法是很经典的，只需要贪心地从前向后遍历每个元素，若当前元素比当前最长下降子序列的最后一个元素小，那么将其加入最长下降子序列的末尾，否则在最长下降子序列中二分查找第一个比它小的元素，将其替换为当前元素。这个算法的正确性是显然的，时间复杂度是 $\mathcal{O}(n \log n)$ 的，只需要 $\mathcal{O}(1)$ 的额外空间（不含存储结果）。然而，需要注意的是，这个算法不能容易地输出最长下降子序列的内容，需要其他手段维护。

2.2.3 代码实现

这个算法的实现没啥好说的其实。求严格和非严格解的算法我都写了，取消注释后拿到的就是严格解。

```cpp
namespace LDS {

template<typename T>
std::vector<T> solve(const std::vector<T> &seq, bool time = true) {
  auto sorted = seq;
  // 使用 std::vector<T>::reverse_iterator 实现降序排序更简洁
  std::sort(sorted.rbegin(), sorted.rend());

  // 取消注释以下两行以求得最长严格下降子序列
  // auto it = std::unique(sorted.rbegin(), sorted.rend());
  // sorted.erase(sorted.begin(), it.base());
```

[1]严格和非严格的区别是，求出的子序列中相邻的两项能不能存在相等的情况。

```
  return LCS::solve(seq, sorted, false);
}

} // namespace LDS
```

2.3 $\mathcal{O}(n)$ 最大子段和算法

2.3.1 题目背景与数学抽象

这个就更简单了！

给定长为 n 的整数数列 $\{a_n\}$，求它的最大子段和和取得这个最大值的区间端点。

2.3.2 算法思考过程

我不觉得这个问题是 DP，如果按照 DP 思考的话首先相当的应该是 $\mathcal{O}(n^3)$ 的区间 DP 套路，而不是凭空冒出来一个 $\mathcal{O}(n)$ 的算法。当然如果就强行把一个贪心递推说成 DP 的话也没啥错误，只能说是认知不同。

这个题是十分经典的贪心模板题，基本上是当作算法课贪心部分的第一道例题出现的。

最容易想到的是 $\mathcal{O}(n^3)$ 的纯朴素算法（枚举所有 $\mathcal{O}(n^2)$ 个区间，然后 $\mathcal{O}(n)$ 地求和）以及到 $\mathcal{O}(n^2)$ 的优化（枚举所有 $\mathcal{O}(n)$ 个区间的左端点，然后 $\mathcal{O}(n)$ 地求出 $\mathcal{O}(n)$ 个区间的和）。

只要想到我们可以在保证正确性的同时，向前同时挪动两个端点（而不是必须固定一个挪动另一个），这个问题的 $\mathcal{O}(n)$ 算法就呼之欲出了。算法流程如下（以下“区间”与“子段”两词混用）：

- 从前向后遍历序列。记录当前左端点为序列开始处。
- 遍历到每个位置时将当前值累加到当前区间和。同时，将当前区间和与记录的最大区间和比较：
  - 如果更大则更新答案（最大区间和、记录的左端点、当前位置作为右端点，三者是一个整体）。
  - 如果小于 0 则置为 0，同时将当前左端点设为下一个位置。这相当于清空当前区间，因为此时不选的收益为 0，要更优于选一个负数。
- 遍历完成后，记录的答案即为最大子段和和取得这个最大值的区间端点。

非常简单的一个算法，正确性也很显然。可以说，这个问题的 $\mathcal{O}(n)$ 算法比 $\mathcal{O}(n^2)$、$\mathcal{O}(n^2 \log n)$ 和 $\mathcal{O}(n^3)$ 的算法在实现上都容易得多。需要的额外空间是 $\mathcal{O}(1)$ 的。

3 程序设计说明

本次作业只提供了两个源文件。DP/LCS.cpp 包含了前两个题目的算法，DP/MSS.cpp 包含了第三个题目的算法。

3.1 如何运行

代码在 C++20 标准下编写，请使用支持 C++20 标准的编译器。以下是推荐的编译命令：

```
g++ -std=c++20 -O2 LCS.cpp -o LCS
./LCS

g++ -std=c++20 -O2 MSS.cpp -o MSS
./MSS
```

3.2 算法接口实现

三种算法均被封装在 namespace 中的名为 solve 的函数中。以下是它们的原型。下述的所有接口中都含有一个 bool 类型的参数 time，默认为 true。如果 time 为 true，那么算法会在运行时计时，并输出运行时间。时间单位采用微秒（10^{-6} s）。

```
namespace LCS {

template<typename T>
std::vector<T> solve(std::vector<T> a, std::vector<T> b, bool time = true);

std::string
solve(const std::string &a_, const std::string &b_, bool time = true);

} // namespace LCS
```

上述前者接收 std::vector<T> 类型的参数，求解两个序列的最长公共子序列，返回最长公共子序列的内容。后者接收 std::string 类型的参数，求解两个字符串的最长公共子序列，返回最长公共子序列的内容。

```
namespace LDS {

template<typename T>
std::vector<T> solve(const std::vector<T> &seq, bool time = true);

} // namespace LDS
```

接收 std::vector<T> 类型的参数，求解一个序列的最长非严格下降子序列，返回最长非严格下降子序列的内容。

```cpp
namespace MSS {

std::tuple<int, int, int> solve(const std::vector<int> &seq, bool time = true);

} // namespace MSS
```

接收 std::vector<int> 类型的参数，求解一个序列的最大子段和，返回最大子段和、最大子段和的左端点和最大子段和的右端点。

3.3 随机大规模数据生成

对于 LCS 和 MSS 问题，分别生成一份大数据用于测试速度：

- LCS：生成两个长度为 10^4 的字符串，包含 ASCII 33~126 的所有字符，求解它们的 LCS。使用以下的生成器生成字符串：

```cpp
auto rand_string = [](int len) -> std::string {
  std::string s;
  s.reserve(len);
  std::generate_n(std::back_inserter(s), len, [&]() -> char {
    static std::mt19937 gen(std::random_device{}());
    static std::uniform_int_distribution<> dis(33, 126);
    return static_cast<char>(dis(gen));
  });
  return s;
};
```

- MSS：生成一个长度为 10^7 的整数数列，每个数的范围是 $[-100, 100]$，求解它的最大子段和。使用以下的生成器生成整数数列：

```cpp
auto rand_seq = [](int len) -> std::vector<int> {
  std::vector<int> seq;
  std::generate_n(std::back_inserter(seq), len, [&]() {
    static std::mt19937 gen(std::random_device{}());
    static std::uniform_int_distribution<> dis(-100, 100);
    return dis(gen);
  });
  return seq;
};
```

4 测试、运行结果与分析说明

运行前文所述的推荐的编译运行命令，得到以下的输出：

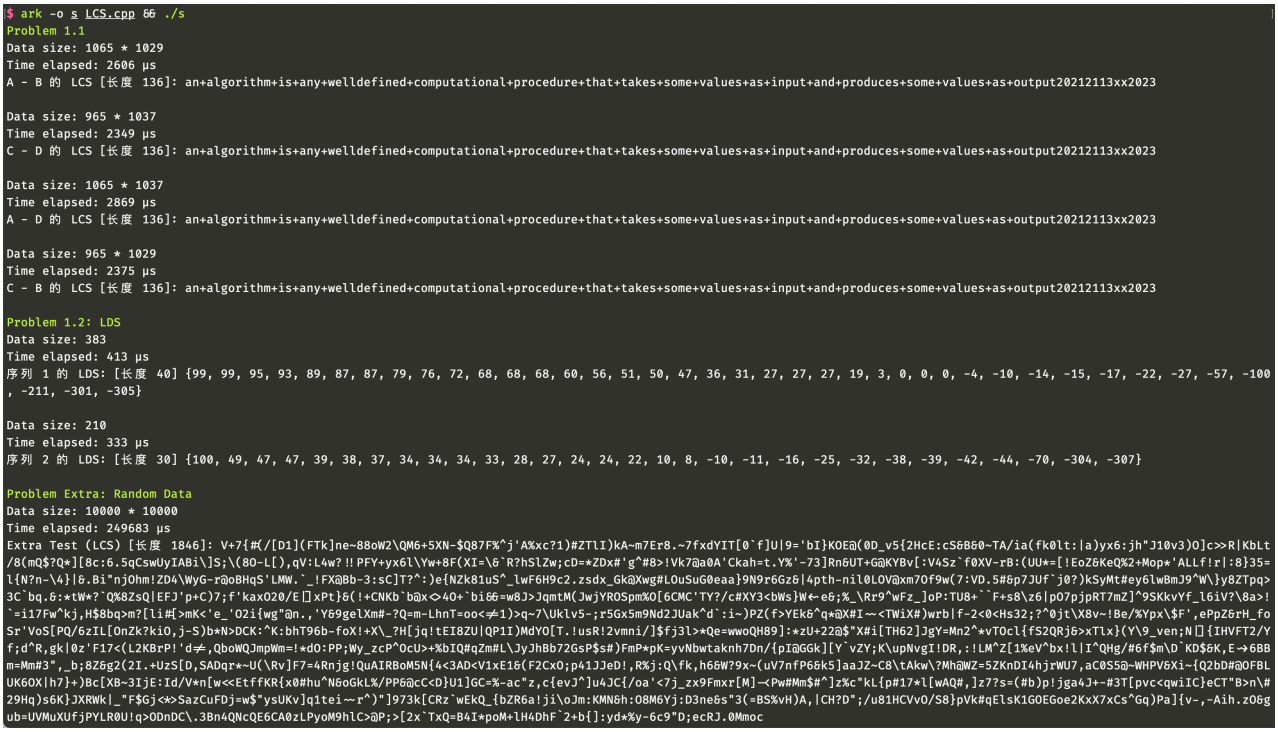

图 1 LCS.cpp 运行结果

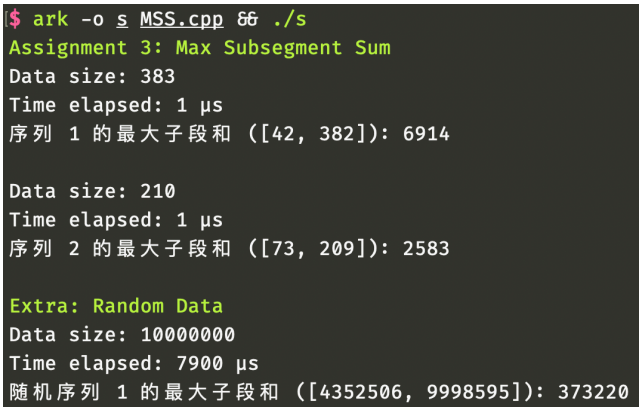

图 2 MSS.cpp 运行结果

图片中包含运行结果和所用时间信息。

4.1 对于给定数据的答案

- 题目 1.1 的 (A,B)，(C,D)，(A,D)，(C,B) 四对字符串两两的最长公共子序列均是

```
an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+
values+as+input+and+produces+some+values+as+output20212113xx2023
```

其长度为 136 个字符。

- 题目 1.2 中，第一组数据的最长非严格下降子序列长 40 个元素。以下是最长非严格下降子序列之一：

```
```

```
{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}
```

第二组数据的最长非严格下降子序列长 30 个元素。以下是最长非严格下降子序列之一：

```
{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}
```

此外，若求最长严格下降子序列，第一组数据的最长严格下降子序列长 35 个元素，第二组数据的最长严格下降子序列长 26 个元素，具体内容略去不表，一来题目没有要求，二来这个答案本来就不唯一。不过我跟同学聊天的时候发现有不少同学读错了题求成了这个。

- 题目 2 的第一组数据的最大子段和是 6914，取到该值的区间是 $[43, 382]$。第二组数据的最大子段和是 2583，取到该值的区间是 $[73, 209]$。

以上所有答案均经过验证。

4.2 算法效率

如前文中两图所示，我们对所有算法进行了计时。以下是统计表格：

题目	时间复杂度	数据规模	实际运行时间
1.1 LCS	$\mathcal{O}(nm)$	$n = 1065, m = 1029$	2.584 ms
		$n = 965, m = 1037$	2.35 ms
		$n = 1065, m = 1037$	2.607 ms
		$n = 965, m = 1029$	2.353 ms
		$n = 10000, m = 10000$	244.095 ms
1.2 最长下降子序列	$\mathcal{O}(n^2)$	$n = 383$	0.413 ms
		$n = 218$	0.333 ms
2 最大子段和	$\mathcal{O}(n)$	$n = 383$	0（小于机器计时精度）
		$n = 210$	0（小于机器计时精度）
		$n = 10000000$	7.9 ms

图 3　算法运行时间统计表

算法的高性能可见一斑！

从表中可以看出，所有算法的运行时间符合时间复杂度理论估计的预期。例如，LCS 的最后一组数据相对于第一组数据的数据规模中 n 和 m 都增加了约 10 倍，而运行时间增加了约 100 倍，符合 $\mathcal{O}(nm)$ 的时间复杂度。

根据这些时间数据推算，我的机器每秒约能进行 10^9 以上次基本运算，这也符合现代计算机的运行速度。当然和我算法竞赛出身，写算法自带小常数可能也有关系（叉腰）。

当然题目算法本身也很不复杂，在算法竞赛里，复杂度 $\mathcal{O}\left(n^2\right)$ 的题的 n 最多只能出 5000，再多就会引起选手对正解复杂度的误判（比如 $n \leq 10000$ 的题目很容易被选手误判成正解 $\mathcal{O}\left(n\sqrt{n}\right)$，进而尝试思考不存在的解法）。不过现在是进行实验，试错成本比结果未知的比赛低得多，因此我进行了各种可能的测试，来确保我的算法的正确性和鲁棒性。

5 总结

本次实验到此完全结束。我在本次实验中实现了 $\mathcal{O}\left(n^2\right)$ 求 LCS 的动态规划算法，并利用它实现了求最长下降子序列的算法。同时，我还实现了 $\mathcal{O}(n)$ 求最大子段和的算法。我在提供的数据上运行了所有算法，并得到了正确的答案。我还随机生成了大规模数据并测试了算法的性能，验证了算法时间复杂度的正确性，加深了对动态规划的理解。实验仍有以下改进方向：

- 优化算法。求 LCS 的算法中，f 数组的空间能被压到 $\mathcal{O}(\min(n, m))$，或者不记录 dir 数组（但 f 不再能压维）。虽然这对总复杂度没有影响，但是可以减小空间常数。如果实际给出的 n 和 m 中有至少一方巨大，那么这个优化的效果会十分明显。
- 在更大的数据量下进行测试，验证算法面对大数据集时的表现。
- 尝试使用更多算法解决，如采用二分求最长下降子序列的 $\mathcal{O}(n \log n)$ 算法，也十分容易实现。
- 分模块编写程序，使之成为一个真正的项目。

附录 1 - 附件清单

```
DP
├── LCS.cpp
└── MSS.cpp
```

附录 2 - 源代码

LCS.cpp

```cpp
// 注：这些头文件全部都用到了。
#include <algorithm>
#include <chrono>
#include <cstdint>
#include <format>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <string_view>
#include <vector>

namespace LCS {

enum class Direction : std::uint8_t { Left = 1, Up, LeftUp };

std::vector<std::vector<int>> f;
std::vector<std::vector<Direction>> dir;

template<typename T>
std::vector<T> _solve(const std::vector<T> &a, const std::vector<T> &b) {
  int n = a.size() - 1, m = b.size() - 1;
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j) {
      if (a[i] == b[j]) {
        f[i][j] = f[i - 1][j - 1] + 1;
        dir[i][j] = Direction::LeftUp;
      } else if (f[i - 1][j] > f[i][j - 1]) {
        f[i][j] = f[i - 1][j];
        dir[i][j] = Direction::Up;
      } else {
        f[i][j] = f[i][j - 1];
        dir[i][j] = Direction::Left;
```

```
      }
    }

  std::vector<T> ans;
  for (int i = n, j = m; i && j;) {
    switch (dir[i][j]) {
      case Direction::LeftUp:
        ans.emplace_back(a[i--]);
        /* fallthrough */
      case Direction::Left:
        --j;
        break;
      case Direction::Up:
        --i;
        break;
      default:
        throw std::runtime_error("Should not reach here!");
    }
  }
  std::reverse(ans.begin(), ans.end());
  return ans;
}

template<typename T>
std::vector<T> solve(std::vector<T> a, std::vector<T> b, bool time = true) {
  a.reserve(a.size() + 1);
  b.reserve(b.size() + 1);
  a.insert(a.begin(), T{});
  b.insert(b.begin(), T{});

  f.resize(a.size() + 1);
  dir.resize(a.size() + 1);
  for (size_t i = 0; i ⩽ a.size(); ++i) {
    f[i].resize(b.size() + 1);
    dir[i].resize(b.size() + 1);
  }

  if (!time)
    return _solve(a, b);

  auto start = std::chrono::high_resolution_clock::now();
  auto ans = _solve(a, b);
  auto end = std::chrono::high_resolution_clock::now();
```

```
  std::cout << std::format(
      "Data size: {} * {}\nTime elapsed: {} μs\n", a.size() - 1, b.size() - 1,
      std::chrono::duration_cast<std::chrono::microseconds>(end - start).count()
  );
  return ans;
}

std::string
solve(const std::string &a_, const std::string &b_, bool time = true) {
  std::vector<char> a(a_.begin(), a_.end());
  std::vector<char> b(b_.begin(), b_.end());
  a.reserve(a.size() + 1);
  b.reserve(b.size() + 1);
  a.insert(a.begin(), '\0');
  b.insert(b.begin(), '\0');

  f.resize(a.size() + 1);
  dir.resize(a.size() + 1);
  for (size_t i = 0; i ⩽ a.size(); ++i) {
    f[i].resize(b.size() + 1);
    dir[i].resize(b.size() + 1);
  }

  if (!time) {
    auto ans = _solve(a, b);
    return std::string(ans.begin(), ans.end());
  }

  auto start = std::chrono::high_resolution_clock::now();
  auto ans = _solve(a, b);
  auto anss = std::string(ans.begin(), ans.end());
  auto end = std::chrono::high_resolution_clock::now();

  std::cout << std::format(
      "Data size: {} * {}\nTime elapsed: {} μs\n", a.size() - 1, b.size() - 1,
      std::chrono::duration_cast<std::chrono::microseconds>(end - start).count()
  );
  return anss;
}

} // namespace LCS

namespace LDS {
```

```cpp
template<typename T>
std::vector<T> solve(const std::vector<T> &seq, bool time = true) {
  auto start = std::chrono::high_resolution_clock::now();
  auto sorted = seq;
  std::sort(sorted.rbegin(), sorted.rend());

  // 取消注释以下两行以求得最长严格下降子序列
  // auto it = std::unique(sorted.rbegin(), sorted.rend());
  // sorted.erase(sorted.begin(), it.base());

  auto ans = LCS::solve(seq, sorted, false);

  if (time) {
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << std::format(
        "Data size: {}\nTime elapsed: {} μs\n", seq.size(),
                  std::chrono::duration_cast<std::chrono::microseconds>(end -
start).count()
    );
  }

  return ans;
}

} // namespace LDS

void output(const std::string_view &prompt, const std::string &s) {
  std::cout << std::format("{} [长度 {}]: {}\n", prompt, s.size(), s)
            << std::endl;
}

void problem1_1() {
  constexpr const static std::string_view FILE_PATH =
      "附件 1.最长公共子序列输入文件-2023.txt";

  std::cout << "\033[32mProblem 1.1\n\033[0m";
  std::ifstream fin(FILE_PATH.data());
  std::string a, b, c, d;
  fin >> a >> a;
  fin >> b >> b;
  fin >> c >> c;
  fin >> d >> d;

  output("A - B 的 LCS", LCS::solve(a, b));
```

```
  output("C - D 的 LCS", LCS::solve(c, d));
  output("A - D 的 LCS", LCS::solve(a, d));
  output("C - B 的 LCS", LCS::solve(c, b));
}

void problem1_2() {
  constexpr const static std::string_view SEQ1_FILE_PATH =
      "附件 2.最大子段和输入数据-序列 1-2023.txt";
  constexpr const static std::string_view SEQ2_FILE_PATH =
      "附件 2.最大子段和输入数据-序列 2-2023.txt";

  auto read_from_file = [&](const std::string_view &path) -> std::vector<int> {
    std::vector<int> seq;
    std::ifstream fin(path.data());
    for (int n; fin >> n;)
      seq.emplace_back(n);
    return seq;
  };

  auto to_string = [](const std::vector<int> &seq) -> std::string {
    std::string s = std::format("[长度 {}] {{", seq.size());
    for (int n : seq)
      s += std::format("{}, ", n);
    s.pop_back();
    s.pop_back();
    s.append("}");
    return s;
  };

  std::cout << "\033[32mProblem 1.2: LDS\033[0m\n";

  auto seq1 = read_from_file(SEQ1_FILE_PATH);
  auto seq2 = read_from_file(SEQ2_FILE_PATH);

  std::cout << std::format("序列 1 的 LDS: {}\n", to_string(LDS::solve(seq1)))
            << std::endl;
  std::cout << std::format("序列 2 的 LDS: {}\n", to_string(LDS::solve(seq2)))
            << std::endl;
}

void extra() {
  auto rand_string = [](int len) -> std::string {
    std::string s;
```

```
    s.reserve(len);
    std::generate_n(std::back_inserter(s), len, [&]() -> char {
      static std::mt19937 gen(std::random_device{}());
      static std::uniform_int_distribution<> dis(33, 126);
      return static_cast<char>(dis(gen));
    });
    return s;
  };

  std::cout << "\033[32mProblem Extra: Random Data\033[0m\n";
  std::string a = rand_string(10000);
  std::string b = rand_string(10000);
  output("Extra Test (LCS)", LCS::solve(a, b));
}

int main() {
  problem1_1();
  problem1_2();
  extra();
}
```

MSS.cpp

```
// 注：这些头文件全部都用到了。
#include <chrono>
#include <format>
#include <fstream>
#include <iostream>
#include <random>
#include <string_view>
#include <tuple>
#include <vector>

namespace MSS {

std::tuple<int, int, int> solve_(const std::vector<int> &seq) {
  int max_sum = 0, max_left = 0, max_right = 0;
  int cur_left = 0, sum = 0;
  for (size_t i = 0; i < seq.size(); ++i) {
    sum += seq[i];
    if (sum > max_sum)
      std::tie(max_sum, max_left, max_right) =
          std::make_tuple(sum, cur_left, i);
```

```
    if (sum < 0) {
      sum = 0;
      cur_left = i + 1;
    }
  }
  return {max_sum, max_left, max_right};
}

std::tuple<int, int, int> solve(const std::vector<int> &seq, bool time = true)
{
  if (!time)
    return solve(seq);

  auto start = std::chrono::high_resolution_clock::now();
  auto ret = solve_(seq);
  auto end = std::chrono::high_resolution_clock::now();
  std::cout << std::format(
      "Data size: {}\nTime elapsed: {} μs\n", seq.size(),
     std::chrono::duration_cast<std::chrono::microseconds>(end - start).count()
  );
  return ret;
}

} // namespace MSS

void output(
    const std::string_view &prompt, const std::tuple<int, int, int> &ret
) {
  std::cout << std::format(
      "{} 的最大子段和 ([{}, {}]): {}\n", prompt, std::get<1>(ret),
      std::get<2>(ret), std::get<0>(ret)
  );
  std::cout << std::endl;
};

void problem2() {
  constexpr const static std::string_view SEQ1_FILE_PATH =
      "附件 2.最大子段和输入数据-序列 1-2023.txt";
  constexpr const static std::string_view SEQ2_FILE_PATH =
      "附件 2.最大子段和输入数据-序列 2-2023.txt";

  auto read_from_file = [&](const std::string_view &path) -> std::vector<int> {
    std::vector<int> seq;
    std::ifstream fin(path.data());
```

```
    for (int n; fin >> n;)
      seq.emplace_back(n);
    return seq;
  };

  std::cout << "\033[32mAssignment 3: Max Subsegment Sum\033[0m\n";
  auto seq1 = read_from_file(SEQ1_FILE_PATH);
  auto seq2 = read_from_file(SEQ2_FILE_PATH);

  output("序列 1", MSS::solve(seq1));
  output("序列 2", MSS::solve(seq2));
}

void extra() {
  auto rand_seq = [](int len) -> std::vector<int> {
    std::vector<int> seq;
    std::generate_n(std::back_inserter(seq), len, [&]() {
      static std::mt19937 gen(std::random_device{}());
      static std::uniform_int_distribution<> dis(-100, 100);
      return dis(gen);
    });
    return seq;
  };

  std::cout << "\033[32mExtra: Random Data\033[0m\n";
  output("随机序列 1", MSS::solve(rand_seq(10000000)));
}

int main() {
  problem2();
  extra();
}
```

点此回到附录以前。