第二次作业

3.2.

问题整理

寻找最长单调递增子序列

背景:

给定一个由 (n) 个元素组成的序列。

任务:

设计一个时间复杂性为 (O(n\log n)) 的算法，找出该序列的最长单调递增子序列。

提示:

- 一个长度为 (i) 的候选子序列的最后一个元素至少与一个长度为 (i-1) 的候选子序列的最后一个元素一样大。
- 通过指向输入序列中元素的指针来维持候选子序列。

要求：编写该算法，并在算法中加入详细的注释解释。

C++版本(无聊多写了几个版本)

```
#include <iostream>
#include <vector>
#include <algorithm>
std::vector<int> findLIS(const std::vector<int>& nums) {
    // `tail`数组用于存储每个长度的LIS的最小末尾值。
    std::vector<int> tail;
    // `prev`数组用于回溯构建LIS，存储每个位置的前一个位置的索引。
    std::vector<int> prev(nums.size(), -1);
    // `indices`数组用于存储tail中每个末尾值在nums中的位置。
    std::vector<int> indices(nums.size());
    for (int i = 0; i < nums.size(); i++) {
        // 使用`lower_bound`在`tail`数组中进行二分搜索，找到当前值应该替换或插入的
位置。
        auto it = std::lower_bound(tail.begin(), tail.end(), nums[i]);
        // 计算插入的位置索引。
        int index = it - tail.begin();
        // 如果找到的位置在tail的末尾，说明当前值比所有已知LIS的末尾值都大，所以增加
到tail。
        if (it == tail.end()) {
            tail.push_back(nums[i]);
        } else {
            // 否则替换找到的位置为当前值，因为更小的值可能会有更长的LIS。
            *it = nums[i];
        }
        // 更新索引数组。
        indices[index] = i;
        // 更新当前位置的前一个位置的索引。
        prev[i] = index > 0 ? indices[index - 1] : -1;
    }
```

```
    // 回溯`prev`数组以重建LIS。
    int len = tail.size();
    int last_index = indices[len - 1];
    std::vector<int> lis(len);
    for (int i = len - 1; i >= 0; i--) {
        lis[i] = nums[last_index];
        last_index = prev[last_index];
    }
    return lis;
}
int main() {
    std::vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    std::vector<int> lis = findLIS(nums);
    std::cout << "Longest Increasing Subsequence: ";
    for (int num : lis) {
        std::cout << num << " ";
    }
    return 0;
}
```

Python版本

```
from bisect import bisect_left
def findLIS(nums):
    # tail列表用于存储每个长度的LIS的最小末尾值。
    tail = []
    # prev列表用于回溯构建LIS，存储每个位置的前一个位置的索引。
    prev = [-1] * len(nums)
    # indices列表用于存储tail中每个末尾值在nums中的位置。
    indices = [-1] * len(nums)
    for i, num in enumerate(nums):
        # 使用bisect_left在tail列表中进行二分搜索，找到当前值应该替换或插入的位置。
        index = bisect_left(tail, num)
        # 如果找到的位置在tail的末尾，说明当前值比所有已知LIS的末尾值都大，所以增加到
tail。
        if index == len(tail):
            tail.append(num)
        else:
            # 否则替换找到的位置为当前值，因为更小的值可能会有更长的LIS。
            tail[index] = num
        # 更新索引列表。
        indices[index] = i
        # 更新当前位置的前一个位置的索引。
        prev[i] = indices[index - 1] if index > 0 else -1
    # 回溯prev列表以重建LIS。
    lis = []
    last_index = indices[len(tail) - 1]
    while last_index != -1:
        lis.append(nums[last_index])
        last_index = prev[last_index]
    # 因为我们是从最后一个元素开始回溯的，所以需要反转结果以得到正确的顺序。
    return lis[::-1]
# 测试代码
nums = [10, 9, 2, 5, 3, 7, 101, 18]
lis = findLIS(nums)
```

```
print("Longest Increasing Subsequence:", lis)
```

JAVA版本

```
import java.util.Arrays;
import java.util.ArrayList;
public class LIS {
    public static ArrayList<Integer> findLIS(int[] nums) {
        // `tail`列表用于存储每个长度的LIS的最小末尾值。
        ArrayList<Integer> tail = new ArrayList<>();
        // `prev`数组用于回溯构建LIS，存储每个位置的前一个位置的索引。
        int[] prev = new int[nums.length];
        Arrays.fill(prev, -1);
        // `indices`数组用于存储tail中每个末尾值在nums中的位置。
        int[] indices = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {
            // 使用`binarySearch`在`tail`列表中进行二分搜索，找到当前值应该替换或
插入的位置。
            int index = java.util.Collections.binarySearch(tail,
nums[i]);
            if (index < 0) {
                index = -(index + 1);
            }
            // 如果找到的位置在tail的末尾，说明当前值比所有已知LIS的末尾值都大，所以
增加到tail。
            if (index == tail.size()) {
                tail.add(nums[i]);
            } else {
                // 否则替换找到的位置为当前值，因为更小的值可能会有更长的LIS。
                tail.set(index, nums[i]);
            }
            // 更新索引数组。
            indices[index] = i;
            // 更新当前位置的前一个位置的索引。
            prev[i] = index > 0 ? indices[index - 1] : -1;
        }
        // 回溯`prev`数组以重建LIS。
        int len = tail.size();
        int last_index = indices[len - 1];
        ArrayList<Integer> lis = new ArrayList<>(len);
        for (int i = len - 1; i >= 0; i--) {
            lis.add(nums[last_index]);
            last_index = prev[last_index];
        }
        java.util.Collections.reverse(lis);
        return lis;
    }
    public static void main(String[] args) {
        int[] nums = {10, 9, 2, 5, 3, 7, 101, 18};
        ArrayList<Integer> lis = findLIS(nums);
        System.out.print("Longest Increasing Subsequence: ");
        for (int num : lis) {
            System.out.print(num + " ");
        }
    }
```

3.4.

问题描述

考虑这个线性规划问题

$max \sum_{i=1}^{n} c_i x_i$

$\sum_{i=1}^{n} a_i x_i \leq b$

x_i为非负数，$1 \leq i \leq n$

设计一个解决此问题的动态规划算法

分析此算法的计算复杂性

C++版本

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
int linearProgramming(const std::vector<int>& a, const std::vector<int>&
c, int b) {
    int n = a.size();
    // dp[j] 表示对于约束值为 j 时目标函数的最大值
    std::vector<int> dp(b + 1, INT_MIN);
    // 基础情况：没有项目且约束值为0时的值为0
    dp[0] = 0;
    for (int i = 0; i < n; ++i) {
        // 反向更新dp以确保先前的状态决策不会过早地被覆盖
        for (int j = b; j >= a[i]; --j) {
            // 检查选择当前项目 i 是否有益
            if (dp[j - a[i]] != INT_MIN) {
                dp[j] = std::max(dp[j], dp[j - a[i]] + c[i]);
            }
        }
    }
    // 处理不存在有效解的情况
    return dp[b] == INT_MIN ? -1 : dp[b];
}
int main() {
    std::vector<int> a = {2, 3, 4, 5};  // 约束的系数
    std::vector<int> c = {3, 4, 5, 6};  // 目标函数的系数
    int b = 8;  // 约束的值
    int result = linearProgramming(a, c, b);
    if (result != -1) {
        std::cout << "目标函数的最大值: " << result << std::endl;
    } else {
        std::cout << "不存在有效解。" << std::endl;
    }
    return 0;
}
```

复杂性分析

1. **时间复杂性**:
   - `for (int i = 0; i < n; ++i)` 这个外部循环运行了 `n` 次，其中 `n` 是 `a` 和 `c` 的元素数量。
   - 对于每一个 `i`，`for (int j = b; j >= a[i]; --j)` 这个内部循环最坏的情况下会运行 `b` 次。

   因此，总的时间复杂性是 $O(n \times b)$
2. **空间复杂性**:
   - `std::vector<int> dp(b + 1, INT_MIN)` 这个向量的大小是 `b + 1`。

   因此，空间复杂性是 $O(b)$