算法第三次作业

李雨轩
计算机2205
2204112913

一. 3-2 求最长上升子列

1. 题目描述

要求在$O(n \log n)$的时间复杂度内实现求解给定整数序列中的最长上升子序列（Longest Increasing Subsequence，简称LIS）的算法

2. 问题分析与算法设计

1. **算法思路**：动态规划是解决最长上升子序列问题的常用方法之一。但在这个算法中，我们采用一种贪心+二分的策略来解决。具体来说，我们维护一个 sub_seq 数组，其中存储当前最长上升子序列。我们遍历输入的整数序列，对于每个数字，如果它大于 sub_seq 的最后一个元素，我们将其添加到 sub_seq 中；否则，我们使用二分查找找到 sub_seq 中第一个大于或等于当前数字的位置，并将其替换为当前数字。通过这种方式，我们能够保持 sub_seq 中的元素是递增的，从而确保其为最长上升子序列。
2. **时间复杂度**：该算法的时间复杂度为 $O(n \log n)$，其中n是输入整数序列的长度。这是由于在每次迭代中，我们执行 $O(n \log n)$的二分查找操作，且总共有 $O(n)$ 次迭代。
3. **空间复杂度**：该算法的空间复杂度为$O(n)$，其中 n 是输入整数序列的长度。这是由于我们维护了一个额外的数组 sub_seq 来存储当前的最长上升子序列。

3. 代码实现

```
static int lengthOfLIS(vector<int>& nums) {
    if(nums.size() == 0) return 0;

    vector<int> sub_seq = {nums[0]};

    for (vector<int>::iterator pos = nums.begin() + 1; pos != nums.end();
      pos++) {
        if (*pos > sub_seq.back())
            sub_seq.push_back(*pos);
        else
            *(lower_bound(sub_seq.begin(), sub_seq.end(), *pos)) = *pos;
    }
```

```
        return sub_seq.size();
    }
};
```

二. 3-4 线性规划问题

1. 题目描述

考虑下面的线性规划问题：

$$max\sum_{i=1}^{n}c_ix_i$$
$$\sum_{i=1}^{n}a_ix_i \le b$$
$$x_i \in \mathbb{Z}^*, 1 \le i \le n$$

试设计一个解此问题的动态规划算法，并分析算法的计算复杂性

2. 问题分析与算法设计

2.1 问题描述

给定线性规划问题，其中目标是最大化变量 x_i 的加权和，即 $\sum_{i=1}^{n}c_ix_i$，同时满足约束条件 $\sum_{i=1}^{n}a_ix_i \le b$ ，其中 x_i 是整数（正整数）。我们的目标是找到满足约束条件下能够使目标函数最大化的整数解。

2.2 算法设计

给定问题描述，我们可以使用动态规划来解决（这是一个完全背包问题）。下面是基于动态规划的算法设计：

1. **初始化：** 创建一个二维数组 `dp`，其中 `dp[i][j]` 表示考虑前 `i` 个变量，并且总和为 `j` 时的最大加权和。
2. **状态转移方程：** 对于每个变量 `x_i`，考虑两种情况：
    - 如果当前的约束 `a_i` 大于当前总和 `j`，则无法将 `x_i` 添加到总和中，因此 `dp[i][j]` 与 `dp[i-1][j]` 相同。
    - 如果 `a_i` 小于等于当前总和 `j`，则我们可以选择添加 `x_i` 或者不添加 `x_i`。我们选择其中能够使目标函数最大化的方案，即

$$\mathrm{dp}[i][j] = \max(\mathrm{dp}[i-1][j], \mathrm{dp}[i][j-a_i] + c_i)$$

同时，如果选择了添加 x_i，则更新 x[i-1]，表示 x_i 的计数加一。

3. **返回结果：** 返回 dp[dimension][b]，即考虑了所有变量并且总和为 b 时的最大加权和。

2.3 算法具体描述

- 函数接受了变量 x、c、a 和 b，分别表示变量值、变量的权重、约束条件的系数和约束条件的限制值。
- 使用二维数组 dp 来存储动态规划的结果。
- 在两个嵌套的循环中，更新 dp 数组的值，最终返回 dp[dimension][b] 作为最优解。

3. 代码实现

```
static int optimizeSolve(vector<int>&x, vector<int> &c, vector<int>& a, int b ){
    int dimension = x.size();
    vector<vector<int>> dp(dimension + 1, vector<int>(b + 1, 0));

    for(int i = 1; i <= dimension; i++){
     for(int j = 1; j <= b; j++){
      if(a[i-1] > j)
        dp[i][j] = dp[i - 1][j];
      else{
        dp[i][j] = max(dp[i-1][j], dp[i][j-a[i-1]]+c[i-1]);
        if(dp[i-1][j] < dp[i][j-a[i-1]]+c[i-1]) x[i-1]++;
      }
     }
    }
    return dp[dimension][b];
  }
```