查找序列中最长递增子序列算法（longest strictly-increasing subsequence——LSS）的设计，优化以及其伪代码实现

Part1:

1.核心思路：采取 DP（动态规划）实现对最长严格递增子序列查找

2.首先我们给出最长严格递增子序列的定义：在数组 T a[n]中，if i<j，则一定有 a[i]<a[j]

3.对于算法的分析：

Step1.引入数组 dp[i]表示以第 i 个元素为结尾的最长严格递增子序列的长度,由于每个元素最少可以单独形成一个子序列，因此初始值为 dp[i] =1,for 0<=i<=n-1（其中 n 为数组的长度）

引入数组 prev，记录每个元素在最长递增子序列中的 num[i]的前一个元素的索引，我们将 prev 的所有元素初始化为-1，故如果 pre[i]=-1,则 num[i]是该递增序列的起点

Step2.对于序列 vector<T> num 中的每个元素 num[i],遍历之前的所有元素 num[j]（1<=j<i）

(1)如果有 num[j]<num[i],则 dp[i]=max(dp[i],dp[j]+1),,同时更新 prev[i] = j;

(2)如果 num[j]>=num[i],则不更新 dp[i],

Step3.遍历整个 dp 数组,取最大值作为最大递增子序列的长度

LSS 的长度=max（dp[k]）(k 从 0 到 n-1)

Step4.回溯输出 LSS，从 dp 数组中找到最大值对应的索引，然后根据 prev 数组从后向前进行回溯，依次记录 LSS 的元素

4.对于算法复杂度的分析：

这种算法的算法复杂度来源于两个方面（循环）

a. 外层遍历序列的所有元素，代价是 O(n)

b. 内层遍历 num[i]之前的所有元素,代价也是 O(n)

所以总体的时间复杂度就是 $O(n^2)$

5.C++伪代码的实现:

```
#include <vector>

#include <iostream>

#include <algorithm>

vector<int> findLSS(vector<int> nums) {


    int n = nums.size();

    if (n == 0) return {};


    vector<int> dp(n, 1), prev(n, -1);

    int maxIndex = 0;


    // 动态规划计算 dp 和 prev

    for (int i = 1; i < n; ++i) {

        for (int j = 0; j < i; ++j) {

            if (nums[j] < nums[i] && dp[i] < dp[j] + 1) {

                dp[i] = dp[j] + 1;

                prev[i] = j;
```

```
            }
        }
        if (dp[i] > dp[maxIndex]) maxIndex = i;   // 更新最大长度索引
    }

    // 回溯构造 LSS
    vector<int> lss
    for (int i = maxIndex; i != -1; i = prev[i]) {
        lss.push_back(nums[i]);
    }
    reverse(lss.begin(), lss.end());
    return lss;
}
```

6.对实例的算法分析:

输入; nums ={10,9,2,5,3,7,101,18}

LSS 的寻找过程:

1. 初始化:

dp = [1, 1, 1, 1, 1, 1, 1, 1]

prev = [-1, -1, -1, -1, -1, -1, -1, -1]

2. 更新 dp 和 prev**:**

   i=1:num[1] = 9,dp 和 prev 不更新

i=2:num[2] = 2,dp 和 prev 不更新

i=3:num[3] = 5:

j = 2:num[2] <num[3],更新 dp[3] = dp[2]+1 = 2,prev[3] = j = 2

i=4:num[4] = 3:

j = 2:num[2] < num[4] ,更新 dp[4] = dp[2] +1=2,prev[4] =2

i=5:nums[5] = 7:

j = 3:num[3]<num[5],更新 dp[5] = dp[3] +1 = 3,prev[5] = 3

i=6:num[6] = 101:

j = 5:num[5]<num[6],更新 dp[6] = dp[5] +1 = 4,prev[6] = 5

i=7:num[7] = 18:

j = 5:num[5]<num[7],更新 dp[7] = dp[5] +1 = 4,prev[7] = 5

3.回溯

从 maxIndex = 6 开始，nums[6] = 101，前一元素为 prev[6] = 5

nums[5] = 7，前一元素为 prev[5] = 3。

nums[3] = 5，前一元素为 prev[3] = 2。

得到 LSS 为[2,5,7,101]

Part2 算法的优化

我们希望通过改进算法，降低算法的复杂度为 O(nlogn),由此我们加入了动态规划+二分查找的方式来改进算法，改进后的算法如下：

1. 核心思路：

 （1）.使用一个辅助数组 sub 维护当前找到的最长递增子序列（note：不一定是最终的递增子序列，但它的长度与最终结果相同）。

 (2).通过二分查找高效更新 sub 的内容。(core)

2.对于算法的分析：

Step1.对于每个元素 nums[i]：

Case1.如果 nums[i] 大于 sub 的最后一个元素，则将 nums[i] 添加到 sub

Case2.否则，通过二分查找找到 sub 中第一个大于等于 nums[i] 的位置，将其替换为 nums[i]，以保证 sub 递增序列尽可能小。

同时维护一个 parent 数组，记录每个元素的前一个元素的索引，以便回溯出完整的 LSS。

Step2. 使用 parent 数组，从 sub 最后一个元素开始回溯，得到最长严格递增子序列。

3.对于算法复杂度的分析：

 优化后的算法的复杂度主要来自两个方面：

1. 每次插入或者替换使用二分查找，时间复杂度减小为 O(logn)
2. 遍历所有元素的复杂度不变，仍为O(n)

 故此算法的时间复杂度为 O(nlogn),达到了我们预期的效果

3. 伪代码的实现

```
    vector<int> findLSS(vector<int>& nums) {
        if (nums.empty()) return {};

        vector<int> sub;            // 存储递增子序列
        vector<int> subIndex;     // 存储递增子序列对应的原索引
        vector<int> parent(nums.size(), -1);   // 记录每个元素的前一
元素索引

        for (int i = 0; i < nums.size(); ++i) {
            // 找到 sub 中第一个 >= nums[i] 的位置
            int pos = lower_bound(sub.begin(), sub.end(), nums[i]) -
sub.begin();

            if (pos == sub.size()) {
                sub.push_back(nums[i]);          // 添加新元素
                subIndex.push_back(i);
            } else {
                sub[pos] = nums[i];               // 替换已有元素
                subIndex[pos] = i;
            }

            // 更新当前元素的上一个元素的索引
```

```
        if (pos > 0) parent[i] = subIndex[pos - 1];
    }

    // 回溯构造最长递增子序列
    vector<int> lss;
    for (int i = subIndex.back(); i != -1; i = parent[i]) {
        lss.push_back(nums[i]);
    }
    reverse(lss.begin(), lss.end());
    return lss;
}
```

4. 对算法的实例分析：

输入数据：nums = {4, 2, 3, 1, 5};

LSS 的寻找过程：

1.初始化：

sub = []：存储当前递增子序列。

subIndex = []：存储 sub 中元素的索引。

parent = [-1, -1, -1, -1, -1]：初始化所有元素为 -1

2. 遍历每个元素并更新状态：

step1. i = 0, nums[0] = 4

sub 为空，直接添加 4：sub = [4], subIndex = [0]

step2. i = 1, nums[1] = 2

二分查找 sub，找到位置 pos = 0，替换 sub[0] :sub= 2，更新 subIndex = [1]

step3. i = 2, nums[2] = 3

二分查找 sub，找到位置 pos = 1，添加 3：sub= [2,3]，更新 parent[2] = 1

step4. i = 3, nums[3] = 1

二分查找 sub，找到位置 pos = 0，替换 sub[0] = 1：sub = [1, 3], subIndex = [3, 2]

step5. i = 4, nums[4] = 5

二分查找 sub，找到位置 pos = 2，添加 5：sub = [1, 3, 5], subIndex = [3, 2, 4]

更新 parent[4] = 2

3. 回溯输出 LSS：

从 subIndex.back() = 4 开始回溯：

nums[4] = 5，前一元素为 parent[4] = 2。

nums[2] = 3，前一元素为 parent[2] = 1。

nums[1] = 2，前一元素为 parent[1] = -1（回溯结束）。

最终回溯结果为：[2, 3, 5]