# 第二次算法设计作业

姓名： 孙谦昊 学号： 2023141460321

【说明】共两道题，至少完成一题，全完成可加分。解答应包括解题思路（方法、时间复杂度分析）、代码（请在关键代码旁加上注释）以及按示例输入得到的输出截图，设计算法时需要考虑边界约束。作业提交时请只保留选做的题，删除空白题。

【题一】 **跳跃游戏**

给你一个非负整数数组nums，你最初位于数组的第一个下标。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标，如果可以，返回true；否则，返回false。

【示例1】

输入：nums = [2,3,1,1,4]

输出：true

解释：可以先跳1步，从下标0到达下标1, 然后再从下标1跳3步到达最后一个下标。

【示例2】

输入：nums = [3,2,1,0,4]

输出：false

解释：无论怎样，总会到达下标为3的位置。但该下标的最大跳跃长度是0， 所以永远不可能到达最后一个下标。

【约束】

1 <= nums.length <= 104

0 <= nums[i] <= 105

【解题思路】

首先这一道题最好采用贪心算法进行解决。可以从数组的起始位置开始，每次选择一个可以跳跃到的最远位置，然后从这个位置继续跳跃，直到到达数组的结束位置或者无法继续跳跃为止。在跳跃的过程中，如果i大于max\_reach，则无法到达位置i，返回false，表示无法到达最后一个下标。反之则更新max\_reach为max(max\_reach, i + nums[i])；如果max\_reach大于或等于数组的最后一个下标（n - 1），则可以直接返回true，表明可以达到最后一个下标。由于整个过程中只遍历了一遍数组，所以说时间复杂度是O（n）。

【代码】

#include <iostream>

#include <vector>

using namespace std;

bool canJump(vector<int>& nums) {

int max\_reach = 0; //存储可以到达的最远位置

int n = nums.size();

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

if (i > max\_reach) {

return false; // 超出数组则不对

}

//更新max\_reach

max\_reach = max(max\_reach, i + nums[i]);

if (max\_reach >= n - 1) {

return true; //如果到达了最后一个则返回true，表明正确的找到了最后一个

}

}

return false; //默认返回作物

}

int main() {

vector<int> nums1 = { 3, 4, 1, 0, 2, 1 };

vector<int> nums2 = { 3, 0, 4, 1, 2, 0, 0, 2 };

if (canJump(nums1))

{

cout << "true" << endl;

}

else

{

cout << "false" << endl;

}

if (canJump(nums2))

{

cout << "true" << endl;

}

else

{

cout << "false" << endl;

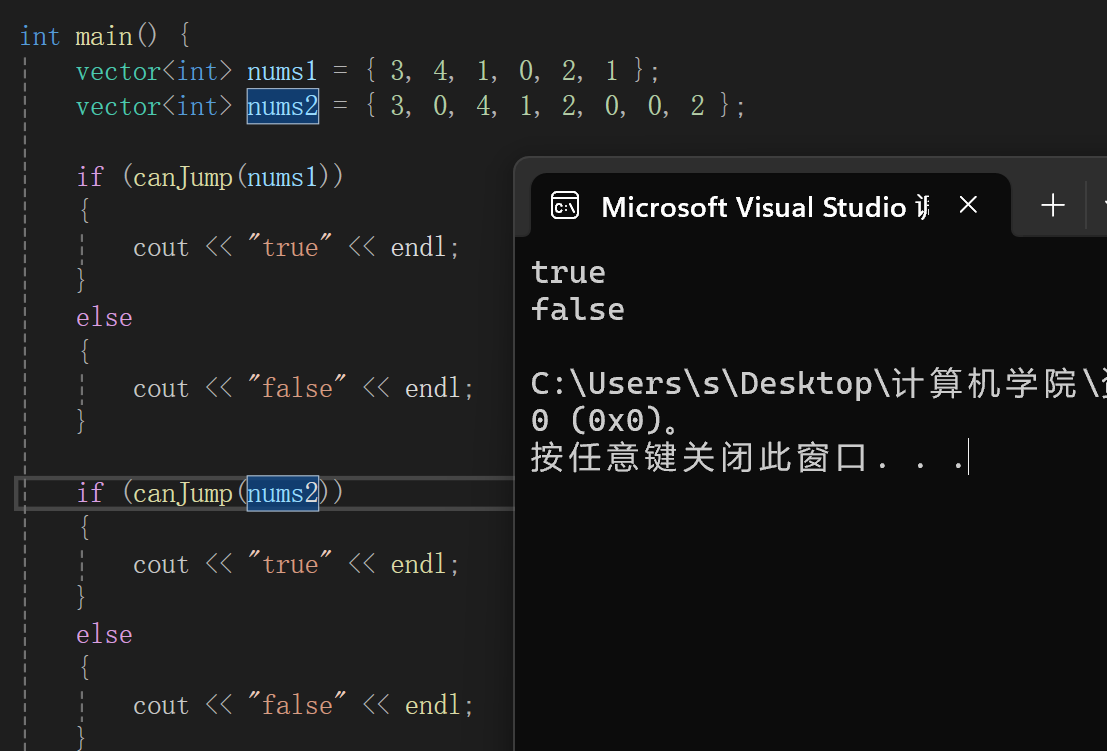
}

return 0;

}

【输出截图】1、输入：nums = [3, 4, 1, 0, 2, 1]

2、输入: nums = [3, 0, 4, 1, 2, 0, 0, 2]



【题二】 **买卖股票的最佳时机**  
给定一个数组prices，它的第i个元素prices[i]表示一支给定股票第i天的价格。

你只能选择某一天买入这只股票，并选择在未来的某一个不同的日子卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回0。

【示例1】

输入：prices = [7,1,5,3,6,4]

输出：5

解释：在第 2 天（股票价格=1）的时候买入，在第 5 天（股票价格=6）的时候卖出，最大利润= 6-1 = 5 。

Ps: 注意利润不能是7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

【示例2】

输入：prices = [7,6,4,3,1]

输出：0

解释：在这种情况下, 没有交易完成, 所以最大利润为 0。

【约束】

1 <= prices.length <= 105

0 <= prices[i] <= 104

【解题思路】

这个问题同样是基于贪心算法展开计算的。在计算最大利润时首先创建一个minPrice来保存最低价格，并默认初始的最低价格为nums的第一个元素。随后再创建一个maxProfit来保存最大利润。在进行遍历数组的时候动态更新minPrice并且动态的更新maxProfit的值。

由于是完整的遍历了一遍数组，所以时间复杂度是O（n）。

【代码】

#include <iostream>

#include <vector>

using namespace std;

int MaxProfit(vector<int>& nums)

{

int n = nums.size();

int minPrice = nums[0];

int maxProfit = 0;

if (n <= 1)

{

return 0; //如果输入的值无法满足计算的条件直接返回

}

for (int i = 0; i < n; i++)

{

if (nums[i] < minPrice)

{

minPrice = nums[i]; //更新最小值

}

else

{

int currentProfit = nums[i] - minPrice; //计算当前的利润

if (currentProfit > maxProfit)

{

//如果当前的利润大于之前计算出的最大的利润则重新更新最大的利润

maxProfit = currentProfit;

}

}

}

return maxProfit; //返回最大值

}

int main()

{

vector<int> num1 = { 2, 1, 4, 8, 3 };

vector<int> num2 = { 3, 4, 5, 1, 2, 6, 6, 7 };

cout << MaxProfit(num1) << endl;

cout << MaxProfit(num2) << endl;

}

【输出截图】1、prices = [2, 1, 4, 8, 3]

1. prices = [3, 4, 5, 1, 2, 6, 6, 7]

