



**本科课程报告**

**课程名称： 算法设计**

**任课教师： 刘权辉**

**学生姓名：**

**学生学号：**

**学院： 计算机学院**

**专业： 计算机科学与技术**

**日期： 2024.6.18**

**期末总结**

随着本课程的结束，我对这趟算法设计的学习之旅进行了深刻的反思和总结。在刘老师的精心指导下，我们不仅深入探讨了算法的理论知识，还通过实践掌握了解决复杂问题的关键技能。这门课程不仅丰富了我的技术视野，也锻炼了我的逻辑思维和问题解决能力。以下是我对本学期所学知识的回顾和总结，希望能够为未来的学习和研究工作打下坚实的基础。

**知识框架**

1. 递归与分治

概念：将复杂问题分解为更小的子问题，递归解决。

适用条件：问题可分解为相似子问题，子问题相互独立。

基本步骤：分解、解决子问题、合并结果。

2. 动态规划

概念：将问题分解为多个阶段，通过阶段决策求解。

基本要素：最优子结构、重叠子问题。

设计步骤：定义最优解结构、递归定义最优值、自底向上计算。

3. 贪心算法

概念：在每一步选择中都采取当前状态下最好的选择。

基本要素：贪心选择性质、最优子结构。

应用问题：活动安排、最优装载、哈夫曼编码等。

4. 回溯法

概念：通过试错的方式尝试分步解决问题。

搜索策略：深度优先搜索。

算法框架：递归回溯、迭代回溯。

5. 分支限界法

概念：在搜索解空间树时，通过剪枝避免无效搜索。

搜索方式：广度优先搜索或最小耗费优先搜索。

**部分算法伪代码实现**

经过一学期的算法学习，对于部分我印象比较深刻的算法，我希望能够用伪代码将其记录下来

第二章：递归与分治

二分搜索

function BinarySearch(array, target, left, right)

while left <= right

mid = (left + right) / 2

if array[mid] == target

return mid

else if array[mid] < target

left = mid + 1

else

right = mid - 1

return -1

大整数乘法（分治法）

function MultiplyLargeIntegers(X, Y)

if length(X) == 1 and X[0] == 0 or Y[0] == 0

return [0]

half = length(X) / 2

a = X[0...half-1]

b = X[half...length(X)-1]

c = Y[0...half-1]

d = Y[half...length(Y)-1]

return Combine(

Multiply(a, d), // 乘法1

Plus(Multiply(b, d), ShiftLeft(Multiply(a, c), half)), // 乘法2+乘法3

Plus(Multiply(a + b, c), ShiftLeft(Multiply(d, c), half)) // 乘法4

)

第三章：动态规划

矩阵链乘问题

function MatrixChainOrder(p, n)

m = array of n-1 \* n-1

for i from 1 to n-1

m[i][i] = 0

for l from 2 to n-1

for i from 1 to n-l

j = i + l

m[i][j] = infinity

for k from i to j-1

q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j]

if q < m[i][j]

m[i][j] = q

// 构造最优解

s = array of n

function ConstructSolution(i, j)

if i == j

s[i] = i

return

k = m[i][j-1]

ConstructSolution(i, k)

ConstructSolution(k+1, j)

第四章：贪心算法

活动安排问题

function GreedySelector(S, f)

n = length(S)

A = array of n boolean values, all initialized to false

A[1] = true

for j from 2 to n

if S[j] >= f[A[n]]

A[j] = true

else

while S[j] < f[A[n]]

n = A.index of false values

// 返回最大相容活动子集

return A

哈夫曼编码

function HuffmanCoding(frequency)

while length(frequency) > 1

select two nodes x, y with minimum frequency

create a new node z with frequency = frequency[x] + frequency[y]

insert z into the priority queue

return the remaining node as the root of Huffman tree

第五章：回溯法

0-1背包问题

function Knapsack(weights, values, capacity)

n = length(weights)

for i from 1 to n

for w from 0 to capacity

if weights[i] <= w

背包容量 = w - weights[i]

if values[i] + Knapsack(weights[i+1:], values[i+1:], 背包容量) > 最大价值

更新 最大价值

return 最大价值

第六章：分支限界法

单源点最短路径问题

function Dijkstra(graph, source)

distance = array of size graph.V with all values set to infinity

distance[source] = 0

priorityQueue = new MinHeap()

priorityQueue.insert(source, distance[source])

while priorityQueue is not empty

u = priorityQueue.extractMin()

for each vertex v adjacent to u

edgeWeight = graph.getEdgeWeight(u, v)

if distance[u] + edgeWeight < distance[v]

distance[v] = distance[u] + edgeWeight

priorityQueue.decreaseKey(v, distance[v])

return distance

**解题报告**

**题目描述**

一只青蛙想要过河。 假定河流被等分为若干个单元格，并且在每一个单元格内都有可能放有一块石子（也有可能没有）。 青蛙可以跳上石子，但是不可以跳入水中。

给你石子的位置列表 stones（用单元格序号 升序 表示）， 请判定青蛙能否成功过河（即能否在最后一步跳至最后一块石子上）。开始时， 青蛙默认已站在第一块石子上，并可以假定它第一步只能跳跃 1 个单位（即只能从单元格 1 跳至单元格 2 ）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1 个单位。 另请注意，青蛙只能向前方（终点的方向）跳跃。

**输入、输出、数据范围**

示例1

输入：stones = [0,1,3,5,6,8,12,17]

输出：true

解释：青蛙可以成功过河，按照如下方案跳跃：跳 1 个单位到第 2 块石子, 然后跳 2 个单位到第 3 块石子, 接着 跳 2 个单位到第 4 块石子, 然后跳 3 个单位到第 6 块石子, 跳 4 个单位到第 7 块石子, 最后，跳 5 个单位到第 8 个石子（即最后一块石子）。

示例2

输入：stones = [0,1,2,3,4,8,9,11]

输出：false

解释：这是因为第 5 和第 6 个石子之间的间距太大，没有可选的方案供青蛙跳跃过去。

数据范围

* 2 <= stones.length <= 2000
* 0 <= stones[i] <= 231 - 1
* stones[0] == 0
* stones 按严格升序排列

**解题思路一——记忆化搜索 + 二分查找**

思路及算法

最直接的想法是使用深度优先搜索的方式尝试所有跳跃方案，直到我们找到一组可行解为止。但是不加优化的该算法的时间复杂度在最坏情况下是指数级的，因此考虑优化。

注意到当青蛙每次能够跳跃的距离仅取决于青蛙的「上一次跳跃距离」。而青蛙此后能否到达终点，只和它「现在所处的石子编号」以及「上一次跳跃距离」有关。因此我们可以将这两个维度综合记录为一个状态。使用记忆化搜索的方式优化时间复杂度。

具体地，当青蛙位于第i个石子，上次跳跃距离为 lastDis时，它当前能够跳跃的距离范围为 [lastDis−1,lastDis+1]。我们需要分别判断这三个距离对应的三个位置是否存在石子。注意到给定的石子列表为升序，所以我们可以利用二分查找来优化查找石子的时间复杂度。每次我们找到了符合要求的位置，我们就尝试进行一次递归搜索即可。

为了优化编码，我们可以认为青蛙的初始状态为：「现在所处的石子编号」为 0（石子从 0 开始编号），「上一次跳跃距离」为 0（这样可以保证青蛙的第一次跳跃距离为 1）。

代码实现

static class Solution1 {

private Boolean[][] rec;

public boolean canCross(int[] stones) {

int n = stones.length;

rec = new Boolean[n][n];

return dfs(stones, 0, 0);

}

private boolean dfs(int[] stones, int i, int lastDis) {

if (i == stones.length - 1) {

return true;

}

if (rec[i][lastDis] != null) {

return rec[i][lastDis];

}

for (int curDis = lastDis - 1; curDis <= lastDis + 1; curDis++) {

if (curDis > 0) {

int j = Arrays.binarySearch(stones, i + 1, stones.length, curDis + stones[i]);

if (j >= 0 && dfs(stones, j, curDis)) {

return rec[i][lastDis] = true;

}

}

}

return rec[i][lastDis] = false;

}

}

复杂度分析：

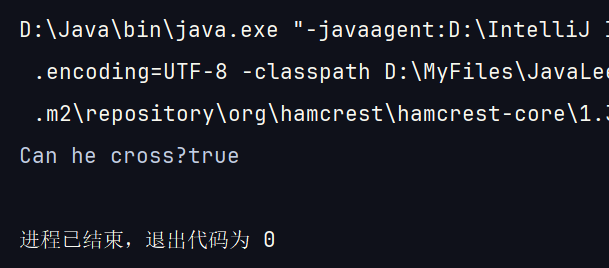
时间复杂度：O(n2log n)，其中 n 是石子的数量。因为青蛙仅能在石子间跳跃，且不能向后方（起点的方向）跳跃，而第 i 个石子后方只有 i−1 个石子，因此在任意位置，青蛙的「上一次跳跃距离」至多只有 O(n) 种，状态总数为 O(n2)。最坏情况下我们要遍历每一个状态，每次我们需要 O(log n)的时间查找指定位置的石子是否存在，相乘即可得到最终时间复杂度。

空间复杂度：O(n2)，其中 n 是石子的数量。最坏情况下我们需要记录全部 O(n2)个状态。

样例测试：

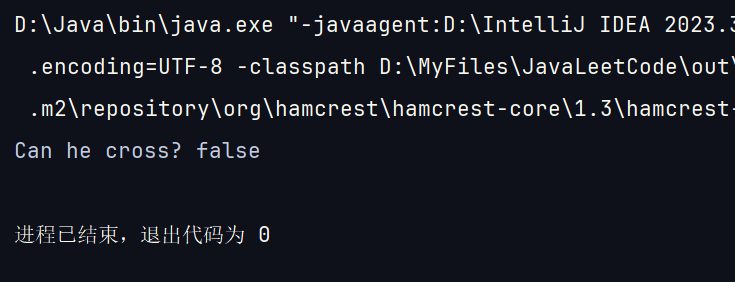
样例1

int[] stones = new int[]{0, 1, 3, 5, 6, 8, 12, 17};

测试结果如下

样例2

int[] stones = new int[]{0, 1, 2, 3, 4, 8, 9, 11};

测试结果如下：

**解题思路二——动态规划**

思路及算法

我们也可以使用动态规划的方法，令 dp[i][k] 表示青蛙能否达到「现在所处的石子编号」为 i 且「上一次跳跃距离」为 k 的状态。

这样我们可以写出状态转移方程：

dp[i][k]=dp[j][k−1]⋁dp[j][k]⋁dp[j][k+1]

式中 j 代表了青蛙的「上一次所在的石子编号」，满足 stones[i]−stones[j]=k。

和方法一相同，状态转移的初始条件为 dp[0][0]=true，表示：「现在所处的石子编号」为 0（石子从 0 开始编号），「上一次跳跃距离」为 0（这样可以保证青蛙的第一次跳跃距离为 1）。当我们找到一个 dp[n−1][k] 为真时，我们就知道青蛙可以到达终点（第 n−1 个石子）。

具体地，对于第 i 个石子，我们首先枚举所有的 j（即上一次所在的石子编号），那么「上一次跳跃距离」k 即为 stones[i]−stones[j]。如果在第 j 个石子上，青蛙的「上一次跳跃距离」可以为 k−1,k,k+1 三者之一，那么我们此时的方案即为合法方案。因此我们只需要检查 dp[j][k−1],dp[j][k],dp[j][k+1] 是否有至少一个为真即可。

优化

为了优化程序运行速度，我们还将推出两个结论，并做出优化：

1. 「现在所处的石子编号」为 i 时，「上一次跳跃距离」k 必定满足 k≤i。

* 当青蛙位于第 0 个石子上时，青蛙的上一次跳跃距离限定为 0，之后每次跳跃，青蛙所在的石子编号至少增加 1，而每次跳跃距离至多增加 1。
* 跳跃 m 次后，青蛙「现在所处的石子编号」i≥m，「上一次跳跃距离」k≤m，因此 k≤i。
* 这样我们可以将状态数约束在 O(n2)。
* 我们可以从后向前枚举「上一次所在的石子编号」j，当「上一次跳跃距离」k 超过了 j+1 时，我们即可以停止跳跃，因为在第 j 个石子上我们至多只能跳出 j+1 的距离。

2． 当第 i 个石子与第 i−1 个石子距离超过 i 时，青蛙必定无法到达终点。

* 由结论 1 可知，当青蛙到达第 i−1 个石子时，它的「上一次跳跃距离」至多为 i−1，因此青蛙在第 i 个石子上最远只能跳出 i 的距离。
* 而距离第 i−1 个石子最近的石子即为第 i 个石子，它们的距离超过了青蛙当前能跳出的最远距离，因此青蛙无路可跳。
* 因此我们可以提前检查是否有相邻的石子不满足条件，如果有，我们可以提前返回 false。

复杂度分析

时间复杂度：O(n2)，其中 n 是石子的数量。因为青蛙仅能在石子间跳跃，且不能向后方（起点的方向）跳跃，而第 i 个石子后方只有 i−1 个石子，因此在任意位置，青蛙的「上一次跳跃距离」至多只有 n 种，状态总数为 n2。最坏情况下我们要遍历每一个状态，每次我们只需要 O(1) 的时间计算当前状态是否可达，相乘即可得到最终时间复杂度。

空间复杂度：O(n2)，其中 n 是石子的数量。我们需要记录全部 n2个状态。

代码实现：

static class Solution2 {

public boolean canCross(int[] stones) {

int n = stones.length;

boolean[][] dp = new boolean[n][n];

dp[0][0] = true;

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

if (stones[i] - stones[i - 1] > i) {

return false;

}

}

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

for (int j = i - 1; j >= 0; --j) {

int k = stones[i] - stones[j];

if (k > j + 1) {

break;

}

dp[i][k] = dp[j][k - 1] || dp[j][k] || dp[j][k + 1];

if (i == n - 1 && dp[i][k]) {

return true;

}

}

}

return false;

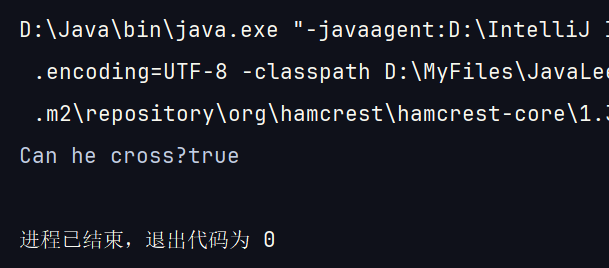
}

}

样例测试：

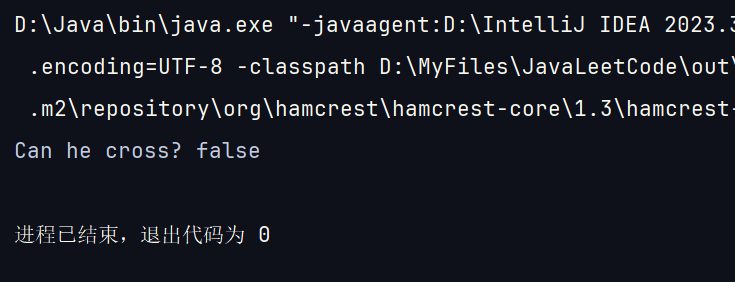
样例1

int[] stones = new int[]{0, 1, 3, 5, 6, 8, 12, 17};

测试结果如下

样例2

int[] stones = new int[]{0, 1, 2, 3, 4, 8, 9, 11};

测试结果如下：

**总结**

本课程不仅提升了我的编程能力，更重要的是锻炼了我的逻辑思维和问题解决能力。我学会了如何分析问题、选择合适的算法策略，并实现高效解决方案。

我期待将这些宝贵的知识应用到更广泛的领域中，继续在算法的世界里探索和成长。

通过这次深入的学习和实践，我更加坚信算法的力量，它不仅能够帮助我们解决具体问题，更能培养我们面对复杂性时的思考和应对能力。在未来的学习中，我将继续深化对算法的理解，并探索更多算法的实际应用。