# 第十五章 动态规划

## 计算题

### 15.2.1

最优括号化方案为: ((A1A2)((A3A4)(A5A6)))

#include <iostream>

#include <vector>

using namespace std;

void printOptimalParens(vector<vector<int>>& s, int i, int j) {

if (i == j) {

cout << "A" << i;

}

else {

cout << "(";

printOptimalParens(s, i, s[i][j]);

printOptimalParens(s, s[i][j] + 1, j);

cout << ")";

}

}

void matrixChainOrder(vector<int>& p) {

int n = p.size() - 1;

vector<vector<int>> m(n + 1, vector<int>(n + 1, 0));

vector<vector<int>> s(n + 1, vector<int>(n + 1, 0));

for (int l = 2; l <= n; ++l) {

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

int j = i + l - 1;

m[i][j] = INT\_MAX;

for (int k = i; k <= j - 1; ++k) {

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

if (q < m[i][j]) {

m[i][j] = q;

s[i][j] = k;

}

}

}

}

cout << "最优括号化方案为: ";

printOptimalParens(s, 1, n);

cout << endl;

cout << "最小计算代价为: " << m[1][n] << endl;

}

int main() {

vector<int> p = { 5, 10, 3, 12, 5, 50, 6 };

matrixChainOrder(p);

return 0;

}

### **15.4-1**

最长公共子序列: 100110或101010

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// 打印LCS

void printLCS(const string& X, const string& Y, vector<vector<int>>& dp) {

int i = X.length();

int j = Y.length();

string lcs = "";

// 从dp数组的右下角回溯

while (i > 0 && j > 0) {

if (X[i - 1] == Y[j - 1]) {

lcs = X[i - 1] + lcs; // 如果字符相等，加入lcs

i--;

j--;

}

else if (dp[i - 1][j] >= dp[i][j - 1]) {

i--; // 如果上方更大，移动到上方

}

else {

j--; // 如果左边更大，移动到左边

}

}

cout << "最长公共子序列: " << lcs << endl;

}

int main() {

string X = "10010101";

string Y = "010110110";

int m = X.length();

int n = Y.length();

// 创建dp数组

vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

// 填充dp数组

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

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

if (X[i - 1] == Y[j - 1]) {

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

}

else {

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

}

}

}

// 输出LCS

printLCS(X, Y, dp);

return 0;

}

### 15.5-2

代价：3.12

k5

/ \

k2 k7

/ \ /

k1 k3 k6

/ \ / / \

d0 d1 d2 d5 d6

/ \

d3 d7

\

d4

#include <iostream>

#include <vector>

#include <limits>

#include <string> // 添加此行以包含 <string> 头文件

using namespace std;

// 全局变量

vector<double> p = { 0.04, 0.06, 0.08, 0.02, 0.10, 0.12, 0.14 };

vector<double> q = { 0.06, 0.06, 0.06, 0.06, 0.05, 0.05, 0.05, 0.05 };

// 打印树的结构

void printTree(const vector<vector<int>>& root, int i, int j, const string& parent, bool isLeft);

// 计算最优BST的成本及树结构

void optimalBST(int n) {

// dp数组：e[i][j]表示i到j的最小搜索代价

vector<vector<double>> e(n + 2, vector<double>(n + 1, 0));

// w数组：w[i][j]表示i到j的权重

vector<vector<double>> w(n + 2, vector<double>(n + 1, 0));

// root数组：root[i][j]表示i到j的最优子树根节点

vector<vector<int>> root(n + 1, vector<int>(n + 1, 0));

// 初始化e和w数组

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

e[i][i - 1] = q[i - 1];

w[i][i - 1] = q[i - 1];

}

// 计算w数组

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

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

w[i][j] = w[i][j - 1] + p[j - 1] + q[j];

}

}

// 填充dp数组，计算最小搜索代价

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

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

int j = i + len - 1;

e[i][j] = numeric\_limits<double>::max();

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

double cost = e[i][r - 1] + e[r + 1][j] + w[i][j];

if (cost < e[i][j]) {

e[i][j] = cost;

root[i][j] = r;

}

}

}

}

// 输出最优搜索代价

cout << "最小搜索代价: " << e[1][n] << endl;

// 输出树的结构

cout << "树的结构：" << endl;

printTree(root, 1, n, "root", false);

}

// 打印树的结构，父节点与子节点之间的关系

void printTree(const vector<vector<int>>& root, int i, int j, const string& parent, bool isLeft) {

if (i > j) {

cout << "d" << i - 1 << " 是 " << parent << " 的" << (isLeft ? "左" : "右") << "子节点" << endl;

return;

}

int r = root[i][j];

if (r != 0) {

// 打印当前节点的父子关系

if (parent == "root") {

cout << "k" << r - 1 << " 是根节点" << endl;

}

else {

if (isLeft) {

cout << "k" << r - 1 << " 是 " << parent << " 的左子节点" << endl;

}

else {

cout << "k" << r - 1 << " 是 " << parent << " 的右子节点" << endl;

}

}

// 打印左子树和右子树

printTree(root, i, r - 1, "k" + to\_string(r - 1), true); // 左子树

printTree(root, r + 1, j, "k" + to\_string(r - 1), false); // 右子树

}

}

int main() {

int n = p.size(); // 确保 n 的值与 p 的大小一致

optimalBST(n);

return 0;

}

## 算法设计题

### 15.1-3 切割钢条

通过 r 数组来记录每个长度的棒子在考虑切割成本后的最大收益。

对于每个长度 jjj，首先假设不进行切割，收益为 p[j]p[j]p[j]。

然后检查所有可能的切割点 i（从 1 到 j−1），对于每个切割点，计算收益 p[i]+r[j−i]−c，选择最大收益并更新 r[j]。

#include <iostream>

#include <vector>

#include <algorithm> // for max()

using namespace std;

int MODIFIED\_CUT\_ROD(const vector<int>& p, int n, int c) {

// 初始化 r 数组，r[i] 表示长度为 i 的棒子的最大收益

vector<int> r(n + 1, 0); // r[0] 到 r[n]

// 动态规划填充 r 数组

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

int q = p[j]; // 不切割时的收益，直接取价格 p[j]

for (int i = 1; i < j; ++i) { // 从 i = 1 到 j-1，寻找最佳切割位置

q = max(q, p[i] + r[j - i] - c); // 计算最大收益，减去切割成本

}

r[j] = q; // 更新长度为 j 的棒子的最大收益

}

return r[n]; // 返回长度为 n 的棒子的最大收益

}

int main() {

// 输入：棒子的价格列表 p 和棒子长度 n

vector<int> p = { 0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30 }; // 注意 p[0] 为空，实际价格从 p[1] 开始

int n = 10; // 棒子长度

int c = 2; // 每次切割的固定成本

// 计算并输出最大收益

cout << "最大收益: " << MODIFIED\_CUT\_ROD(p, n, c) << endl;

return 0;

}

### **15-9 字符串拆分**

子问题：将字符串分为左右两部分，将结果合并与当前结果比较

外层循环遍历区间的长度，第二层循环遍历区间的起始点，第三层循环遍历每个区间内的可能切割点,时间复杂度O(m^3)

#include <iostream>

#include <vector>

#include <limits>

#include <algorithm>

#include <functional>

using namespace std;

void MIN\_CUT\_COST(const vector<int>& L, int n) {

int m = L.size() - 1; // L的长度减1得到实际的切割点数量

// dp[i][j]表示区间[i, j]的最小切割成本

vector<vector<int>> dp(m + 1, vector<int>(m + 1, 0));

// cut[i][j]记录最优的切割点

vector<vector<int>> cut(m + 1, vector<int>(m + 1, -1));

// 初始化：没有切割的区间成本为 0

for (int len = 2; len <= m; len++) { // 处理长度为2到m的区间

for (int i = 0; i <= m - len; i++) {

int j = i + len;

dp[i][j] = numeric\_limits<int>::max(); // 设置初始值为最大值

// 遍历区间[i, j]中的每个可能的切割点k

for (int k = i + 1; k < j; k++) {

int cost = L[j] - L[i] + dp[i][k] + dp[k][j]; // 计算当前切割点k的成本

if (cost < dp[i][j]) {

dp[i][j] = cost;

cut[i][j] = k; // 记录最优切割点

}

}

}

}

// 输出最小切割成本

cout << "最小切割成本: " << dp[0][m] << endl;

// 输出最优切割顺序

cout << "最优切割顺序: ";

vector<int> cuts;

function<void(int, int)> reconstruct = [&](int i, int j) {

if (i + 1 < j) {

int k = cut[i][j];

cuts.push\_back(L[k]);

reconstruct(i, k);

reconstruct(k, j);

}

};

reconstruct(0, m);

for (int cutPoint : cuts) {

cout << cutPoint << " ";

}

cout << endl;

}

int main() {

// 假设L为切割点位置数组，n为总长度

vector<int> L = { 0, 2, 8, 10, 20 }; // L[i]表示切割点

int n = 20; // 字符串长度

MIN\_CUT\_COST(L, n);

return 0;

}

### **15-11 库存规划**

子问题：dp[i,j]表示过了i个月，库存为j时的成本

外层循环遍历每个月，第二层循环遍历库存量0到D，第三层循环遍历当前月可能生产的机器数,时间复杂度O(nD^2)

#include <iostream>

#include <vector>

#include <limits>

#include <algorithm>

using namespace std;

int D = 0;

// 计算生产和库存的成本

int productionCost(int produced, int demand, int maxProduction, int costPerMachine) {

int extra = max(0, produced - maxProduction);

return extra \* costPerMachine; // 超出生产能力的机器需要额外的成本

}

// 计算库存持有成本

int holdingCost(int stock, const vector<int>& h) {

return h[stock]; // 持有stock台机器的库存成本

}

// 动态规划解决最小成本问题

void RinkyDinkMinCost(const vector<int>& demand, int m, int c, const vector<int>& h, int n) {

// dp[i][j]表示前i个月，库存为j台机器时的最小成本

vector<vector<int>> dp(n + 1, vector<int>(D + 1, numeric\_limits<int>::max()));

dp[0][0] = 0; // 初始条件：第0个月，库存为0的成本为0

// 遍历每个月

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

for (int j = 0; j <= D; j++) { // j为前i-1个月的库存

if (dp[i - 1][j] == numeric\_limits<int>::max()) continue;

// 尝试为当月生产的机器数

for (int x = 0; x <= min(D, m + j); x++) { // x为生产的机器数

int newStock = j + x - demand[i - 1]; // 新库存

if (newStock < 0 || newStock > D) continue;

// 计算新的成本

int cost = dp[i - 1][j] + productionCost(x, demand[i - 1], m, c) + holdingCost(newStock, h);

//cost为前i-1个月最小成本 + 当月生产成本 + 当月库存持有成本

dp[i][newStock] = min(dp[i][newStock], cost);// 更新最小成本

}

}

}

// 输出最小成本

cout << "最小成本: " << dp[n][0] << endl;

}

int main() {

int n = 5; // 假设5个月

vector<int> demand = { 5, 10, 7, 8, 6 }; // 每个月的需求量

int m = 7; // 每月最大生产量

int c = 2; // 兼职工人每台机器的成本

vector<int> h; // 库存持有成本

for (int i = 0; i < n; i++) D += demand[i]; // 计算总需求量

for (int i = 0; i <= D; i++) h.push\_back(i); //假设 库存持有成本为i/2

RinkyDinkMinCost(demand, m, c, h, n);

return 0;

}

## 证明题

### 15.2-5

在 MATRIX-CHAIN-ORDER 中，假设我们正在计算 m[i,j]，我们会遍历所有 k，计算出递归的结果 m[i,k] 和 m[k+1,j]，因此 m[i,j] 在计算时需要引用 m[i,k] 和 m[k+1,j]，这相当于每次计算 m[i,j] 时会引用其他 m[i,k] 和 m[k+1,j] 的值。

由此可见，我们每次在计算 m[i,j] 时，会**引用 2 次**其他条目。

**第一层求和** 代表了不同的子问题，从 i = 1 到 n，即考虑不同的区间。

**第二层求和** 代表了不同切割点的选择，即考虑在不同位置进行切割。i到n-l+1

**第三层（内层）求和** 每个 m[i,j] 被引用两次（一次计算当前条目，一次参考其他条目）。

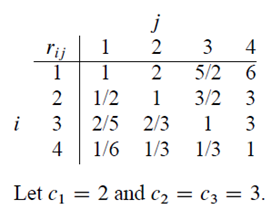
求和后的结果即为原等式

### 15.3-6：

#### Ck=0时，问题具有最优子结构

考虑分为两部分1-k,k+1-n（这里我们已经求得了k），两条子路径为最优，问题的最优解可以通过组合子问题的最优解得到

#### (2)基于以下数据讨论第二问



最优解1-4：最终得到4

考虑1-3和3-4：子问题1-3的最优解为2.5，3-4为3，但合并后需要扣除佣金3，结果为2.5

因此在组合子路径时，佣金（与交易次数有关）会影响到总成本，从而导致无法通过简单的子问题解法得到全局最优解

# 贪心

### 16.1-4

“**区间图着色问题**”。我们可以创建一个区间图，顶点表示给定的活动，边连接不兼容的活动。在这种情况下，求解最少的教室数量实际上就是图的最小着色数，也就是说，找到最少的颜色来使图中的所有相邻顶点颜色不同

#### 算法流程

1. 按照每个活动的开始时间对活动进行排序。
2. 建立classroom结构体和相应的数组，结构体中用end表示结束时间，isBusy表示是否被占用
3. 对于每一个活动：

如果其开始时间不冲突（即当前有空闲教室），从 classrooms中查询一个空闲教室，安排该活动。

如果没有空闲教室，从数组中中添加一个教室。

#include <iostream>

#include <vector>

#include <algorithm>

struct Activity {

int start; // 活动的开始时间

int end; // 活动的结束时间

};

struct Clssroom

{

int end;

int isBusy = 0; // 0表示空闲，1表示忙碌

};

// 比较函数：按活动的开始时间排序

bool compare(Activity a, Activity b) {

return a.start < b.start;

}

int scheduleActivities(std::vector<Activity>& activities) {

// 第一步：按活动的开始时间排序

std::sort(activities.begin(), activities.end(), compare);

int numClassrooms = 0; // 使用的教室数量

std::vector<Clssroom> classrooms;

// 遍历每个活动

for (auto& activity : activities) {

bool assigned = false;

// 释放已结束活动的教室

for (int i = 0; i < classrooms.size(); ++i) {

if (activity.start >= activities[i].end) {

classrooms[i].isBusy = 0; // 设置为空闲

}

}

// 查找空闲教室

for (int i = 0; i < classrooms.size(); ++i) {

// 如果找到一个空闲教室，分配给当前活动

if (classrooms[i].isBusy == 0) {

classrooms[i].isBusy = 1; // 设置为忙碌

classrooms[i].end = activity.end; // 设置活动结束时间

assigned = true;

break;

}

}

// 如果没有空闲教室，新增一个教室

if (!assigned) {

classrooms.push\_back({ activity.end,1 }); // 新增一个教室并设置为忙碌

numClassrooms++; // 使用的教室数量加一

}

}

// 返回最少教室数量

return numClassrooms;

}

int main() {

// 示例：活动列表，包含开始和结束时间

std::vector<Activity> activities = { {1, 4}, {2, 6}, {5, 8}, {7, 9}, {3, 5} };

// 调用函数并输出结果

int result = scheduleActivities(activities);

std::cout << "所需的最少教室数量: " << result << std::endl;

return 0;

}

### 16.2-6 分数背包问题

常规算法中使用快排将物品按价值密度排序后再进行选取，为了实现线性复杂度，可以使用快速选择算法找到中位数，将原问题划分为两部分，且优先对价值较大的一部分进行求解

#include <iostream>

#include <vector>

#include <algorithm>

struct Item {

int value; // 物品的价值

int weight; // 物品的重量

double density; // 物品的价值密度

// 计算物品的价值密度

void calculateDensity() {

density = (double)value / weight;

}

};

// 查找中位数的函数（线性时间选择）

int partition(std::vector<Item>& items, int low, int high) {

double pivot = items[high].density;

int i = low - 1;

for (int j = low; j < high; ++j) {

if (items[j].density > pivot) {

++i;

std::swap(items[i], items[j]);

}

}

std::swap(items[i + 1], items[high]);

return i + 1;

}

double findMedian(std::vector<Item>& items, int low, int high) {

if (low == high) return items[low].density;

int pivotIndex = partition(items, low, high);

int leftSize = pivotIndex - low + 1;

if (leftSize == (high - low + 1) / 2 + 1)

return items[pivotIndex].density;

else if (leftSize < (high - low + 1) / 2 + 1)

return findMedian(items, pivotIndex + 1, high);

else

return findMedian(items, low, pivotIndex - 1);

}

// 贪心算法求解分数背包问题

double fractionalKnapsack(std::vector<Item>& items, int W) {

// 计算每个物品的价值密度

for (auto& item : items) {

item.calculateDensity();

}

// 计算中位数

double median = findMedian(items, 0, items.size() - 1);

double totalWeight = 0;

double totalValue = 0;

// 处理高价值密度的物品

for (auto& item : items) {

if (item.density > median) {

if (totalWeight + item.weight <= W) {

totalWeight += item.weight;

totalValue += item.value;

}

else {

double remainingWeight = W - totalWeight;

totalWeight += remainingWeight;

totalValue += remainingWeight \* item.density;

break;

}

}

}

return totalValue;

}

int main() {

std::vector<Item> items = {

{60, 10, 0},

{100, 20, 0},

{120, 30, 0}

};

int W = 50; // 背包容量

double result = fractionalKnapsack(items, W);

std::cout << "Max Value: " << result << std::endl;

return 0;

}

### 16.2-7 最大化收益

“**贪心匹配**” ，根据大数与大数配对原则，将AB都升序排序即可。

证明：各取出两个数研究局部最优性（进行一定的数学运算即可）

#include <iostream>

#include <vector>

#include <algorithm>

// 计算收益的函数

double calculateProfit(const std::vector<int>& A, const std::vector<int>& B) {

double profit = 1.0;

int n = A.size();

// 计算收益

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

profit \*= pow(A[i], B[i]);

}

return profit;

}

int main() {

int n;

// 输入集合的大小

std::cout << "请输入集合大小 n: ";

std::cin >> n;

std::vector<int> A(n), B(n);

// 输入集合 A

std::cout << "请输入集合 A 的元素: ";

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

std::cin >> A[i];

}

// 输入集合 B

std::cout << "请输入集合 B 的元素: ";

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

std::cin >> B[i];

}

// 对集合 A 和 B 分别进行排序

std::sort(A.begin(), A.end());

std::sort(B.begin(), B.end());

// 计算并输出最大收益

double maxProfit = calculateProfit(A, B);

std::cout << "最大收益: " << maxProfit << std::endl;

return 0;

}

### 16-3.3 斐波那契数列和哈夫曼编码

哈夫曼编码:

序号 8: 0

序号 7: 10

序号 6: 110

序号 5: 1110

序号 4: 11110

序号 3: 111110

序号 2: 1111110

序号 1: 1111111

不难发现**规律**:第一个数由n个1组成，第二个数由n-1个1和末尾1个0组成，往后每个数(设为p),都由n-p个1和1个0组成

#### 普通哈夫曼树

#include <iostream>

#include <queue>

#include <vector>

#include <unordered\_map>

#include <string>

using namespace std;

// 哈夫曼树节点结构

struct Node {

char data; // 字符

int freq; // 字符频率

Node\* left;

Node\* right;

// 构造函数

Node(char data, int freq) : data(data), freq(freq), left(nullptr), right(nullptr) {}

};

// 比较节点频率的比较器

struct Compare {

bool operator()(Node\* l, Node\* r) {

return l->freq > r->freq; // 小的频率优先

}

};

// 递归生成哈夫曼编码

void generateHuffmanCode(Node\* root, string code, unordered\_map<char, string>& huffmanCode) {

// 如果是叶子节点，保存编码

if (!root) return;

if (!root->left && !root->right) {

huffmanCode[root->data] = code;

}

// 向左子树递归

generateHuffmanCode(root->left, code + "0", huffmanCode);

// 向右子树递归

generateHuffmanCode(root->right, code + "1", huffmanCode);

}

int main() {

int n;

cout << "请输入符号的个数 n: ";

cin >> n;

vector<char> symbols(n);

vector<int> frequencies(n);

// 输入符号和频率

cout << "请输入每个符号的频率: " << endl;

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

cout << "符号 " << i + 1 << ": ";

cin >> symbols[i];

cout << "频率: ";

cin >> frequencies[i];

}

// 创建哈夫曼树节点

priority\_queue<Node\*, vector<Node\*>, Compare> minHeap;

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

minHeap.push(new Node(symbols[i], frequencies[i]));

}

// 构建哈夫曼树

while (minHeap.size() > 1) {

// 取出最小的两个节点

Node\* left = minHeap.top();

minHeap.pop();

Node\* right = minHeap.top();

minHeap.pop();

// 创建一个新的父节点，频率是两个子节点频率之和

Node\* newNode = new Node('$', left->freq + right->freq);

newNode->left = left;

newNode->right = right;

// 将新节点插入到队列中

minHeap.push(newNode);

}

// 获取哈夫曼编码

unordered\_map<char, string> huffmanCode;

generateHuffmanCode(minHeap.top(), "", huffmanCode);

// 输出每个符号的哈夫曼编码

cout << "\n哈夫曼编码:" << endl;

for (auto& pair : huffmanCode) {

cout << pair.first << ": " << pair.second << endl;

}

return 0;

}

#### 斐波那契-哈夫曼编码

#include <iostream>

#include <queue>

#include <vector>

#include <unordered\_map>

#include <string>

using namespace std;

// 哈夫曼树节点结构

struct Node {

int data; // 数字表示的符号（斐波那契数列中的索引）

int freq; // 字符频率（斐波那契数列中的数值）

Node\* left;

Node\* right;

// 构造函数

Node(int data, int freq) : data(data), freq(freq), left(nullptr), right(nullptr) {}

};

// 比较节点频率的比较器

struct Compare {

bool operator()(Node\* l, Node\* r) {

return l->freq > r->freq; // 小的频率优先

}

};

// 递归生成哈夫曼编码

void generateHuffmanCode(Node\* root, string code, unordered\_map<int, string>& huffmanCode) {

// 如果是叶子节点，保存编码

if (!root) return;

if (!root->left && !root->right) {

huffmanCode[root->data] = code;

}

// 向左子树递归

generateHuffmanCode(root->left, code + "0", huffmanCode);

// 向右子树递归

generateHuffmanCode(root->right, code + "1", huffmanCode);

}

// 生成前 n 项斐波那契数列

vector<int> generateFibonacci(int n) {

vector<int> fibonacci(n);

if (n > 0) fibonacci[0] = 1;

if (n > 1) fibonacci[1] = 1;

for (int i = 2; i < n; ++i) {

fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];

}

return fibonacci;

}

int main() {

int n;

cout << "请输入斐波那契数列的项数 n: ";

cin >> n;

// 生成前 n 项斐波那契数列

vector<int> fibonacci = generateFibonacci(n);

// 创建哈夫曼树节点

priority\_queue<Node\*, vector<Node\*>, Compare> minHeap;

// 使用斐波那契数列的值作为频率

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

minHeap.push(new Node(i + 1, fibonacci[i])); // 符号默认为 1, 2, 3, ...

}

// 构建哈夫曼树

while (minHeap.size() > 1) {

// 取出最小的两个节点

Node\* left = minHeap.top();

minHeap.pop();

Node\* right = minHeap.top();

minHeap.pop();

// 创建一个新的父节点，频率是两个子节点频率之和

Node\* newNode = new Node(0, left->freq + right->freq); // 无符号

newNode->left = left;

newNode->right = right;

// 将新节点插入到队列中

minHeap.push(newNode);

}

// 获取哈夫曼编码

unordered\_map<int, string> huffmanCode;

generateHuffmanCode(minHeap.top(), "", huffmanCode);

// 输出每个斐波那契数列项的哈夫曼编码

cout << "\n哈夫曼编码:" << endl;

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

cout << "符号 " << i + 1 << " (" << fibonacci[i] << "): " << huffmanCode[i + 1] << endl;

}

return 0;

}

### 16-1 找零问题

#### (a)描述一个贪心算法，用于由25美分硬币、10美分硬币、5美分硬币和1美分硬币组成的找零。证明你的算法能够产生最优解。

每次选择能用的面额最大的硬币，确保在每一步都能使用最少的硬币数来减小剩余金额。

#### (b)

考虑将目标数字用c进制表示，**按需进位**，不难发现这样一定能使消耗的硬币数最少

#### (c)

考虑面额有1，3，4，5，需要找零7

贪心策略：5 1 1

最优策略：3 4

#### (d) 动态规划

遍历dp数组1-n，每次循环中查找可用的硬币,总时间复杂度为O(nk)

#include <iostream>

#include <climits>

#include <algorithm>

#include <vector>

using namespace std;

int minCoins(int coins[], int k, int n) {

// dp[i]表示金额i所需的最小硬币数

vector<int>dp(n+1,0);

// 初始化dp数组，默认情况下将所有金额初始化为INT\_MAX（代表无法构成该金额）

for (int i = 0; i <= n; i++) dp[i] = INT\_MAX;

// 金额为0时，不需要硬币

dp[0] = 0;

// 遍历所有金额

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

// 遍历所有硬币

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

if (coins[j] <= i) {

// 如果硬币可以用来支付该金额，更新dp[i]为最小的硬币数

dp[i] = std::min(dp[i], dp[i - coins[j]] + 1);

}

}

}

// 如果dp[n]没有改变，说明无法凑成金额n，返回-1

if (dp[n] == INT\_MAX) {

return -1;

}

return dp[n];

}

int main() {

// 测试用例1：硬币面额为1, 5, 10, 25，目标金额为30

int coins1[] = { 1, 5, 10, 25 };

int k1 = 4; // 硬币种类数

int n1 = 30; // 目标金额

std::cout << "最少硬币数： " << minCoins(coins1, k1, n1) << std::endl; // 输出： 2 (25 + 5)

// 测试用例2：硬币面额为1, 3, 4，目标金额为6

int coins2[] = { 1, 3, 4 };

int k2 = 3; // 硬币种类数

int n2 = 6; // 目标金额

std::cout << "最少硬币数： " << minCoins(coins2, k2, n2) << std::endl; // 输出： 2 (3 + 3)

// 测试用例3：硬币面额为2, 5，目标金额为3

int coins3[] = { 2, 5 };

int k3 = 2; // 硬币种类数

int n3 = 3; // 目标金额

std::cout << "最少硬币数： " << minCoins(coins3, k3, n3) << std::endl; // 输出： -1 (无法找零)

// 测试用例4：硬币面额为1，目标金额为100

int coins4[] = { 1 };

int k4 = 1; // 硬币种类数

int n4 = 100; // 目标金额

std::cout << "最少硬币数： " << minCoins(coins4, k4, n4) << std::endl; // 输出： 100 (需要100个1分硬币)

return 0;

}