**广州软件学院2022 —— 2023学年第一学期考查卷**

**课程代码： GF2001 课程名称： 算法课程设计**

装订线 装订线 装订线

**学分： 1**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 题次 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总 分 | 评卷人 |
| 分数 | 40 | 60 |  |  |  |  |  |  | 100 |  |
| 评分 |  |  |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| 得分 | 评卷人 |
|  |  |

一、用自己的话简要描述所学的快速排序、暴力求解、贪心法、动态规划等四种算法的特点、应用场景及其实现思路。（40分）

#### 算法特点：

快速排序：是一种分治算法，它的基本思想是将一个数组分成两个子数组，将两个子数组分别排序，然后将结果合并起来。它的主要特点是在大多数情况下都能够提供较快的排序速度，并且不需要使用额外的内存空间。

暴力求解：是一种简单直接的算法，它尝试使用最简单的方法解决问题。它的特点是实现简单，但是在处理大规模问题时速度会很慢。

贪心法：是一种在每一步都选择最优解的算法。它的基本思想是，对于每一步，都选择当前看起来最优的决策，期望这样的决策能够导致最终的最优解。它的主要特点是实现简单，但是并不保证能够得到最优解。

动态规划：是一种通过拆分问题为若干个子问题来解决复杂问题的方法。它的基本思想是将大问题拆分成若干个子问题，然后递归地解决这些子问题，并将子问题的解保存在一个表格中，以便在需要的时候可以快速访问。它的主要特点是能够解决较大规模的问题，但是需要使用较多的内存空间，并且可能会出现重复计算的问题。

#### 应用场景：

快速排序：快速排序通常用于排序数据，特别是大型数据集。它的主要优势在于其平均时间复杂度为 O(nlogn)，速度快，并且不需要使用额外的内存空间。

暴力求解：暴力求解通常用于解决简单的问题，特别是当问题规模较小时。它的主要优势在于其实现简单，易于理解。

贪心法：贪心法通常用于解决最优化问题，即在满足某些限制条件的情况下，寻找最优解。它的主要优势在于实现简单，但是并不保证能够得到最优解。

动态规划：动态规划通常用于解决复杂的最优化问题，即在满足某些限制条件的情况下，寻找最优解。它的主要优势在于能够解决较大规模的问题，但是需要使用较多的内存空间，并且可能会出现重复计算的问题。

#### 实现思路：

快速排序：快速排序的基本思想是在数组中选择一个元素作为“基准”，然后通过比较数组中的其他元素与基准的大小关系，将数组分成两个子数组。在这两个子数组中分别进行快速排序，然后将排好序的子数组合并起来即可。

暴力求解：暴力求解的实现思路很简单，就是对于给定的问题，枚举所有的可能解，然后在所有可能解中选择最优解即可。

贪心法：贪心法的实现思路也很简单，就是在每一步都选择当前看起来最优的决策，期望这样的决策能够导致最终的最优解。

动态规划：动态规划的实现思路比较复杂，需要满足以下几个条件：

1、最优子结构性质：问题的最优解包含其子问题的最优解。

2、无后效性：某个状态以前的过程不会影响以后的状态，只与当前状态有关。

3、有重叠子问题性质：子问题会重复出现。

根据这些条件，我们可以使用表格或数组来记录已经求解过的子问题的结果，以便在需要的时候快速访问。在求解每一个子问题时，可以使用递推或者自底向上的方法。

递推方法：

在递推方法中，我们从最小的子问题开始求解，逐步递推到较大的子问题。每一步都使用之前求解的子问题的结果来计算当前子问题的结果。这种方法的优点在于实现简单，但是可能会出现重复计算的问题。

自底向上的方法：

在自底向上的方法中，我们从最小的子问题开始求解，逐步递推到较大的子问题。这种方法的优点在于能够避免重复计算，但是实现较为复杂。

|  |  |
| --- | --- |
| 得分 | 评卷人 |
|  |  |

二、假设某游戏角色需要不断学习技能，该角色的天赋为200，每学习一种技能，将获得一定的攻击力和防御力，同时消耗一定的天赋。设计3种算法使得该角色通过学习后得到的防御力最高（最优解或次优解均可），注意代码规范。(每种算法20分，共60分)

各技能信息如下格式：

|  |  |  |  |
| --- | --- | --- | --- |
| 技能名称 | 消耗天赋值 | 获得攻击力 | 获得防御力 |
| 技能1 | 10 | 10 | 10 |
| 技能2 | 20 | 50 | 20 |
| 技能3 | 30 | 40 | 80 |
| 技能4 | 40 | 60 | 20 |
| 技能5 | 50 | 40 | 80 |
| 技能6 | 15 | 50 | 20 |
| 技能7 | 55 | 70 | 20 |
| 技能8 | 70 | 100 | 30 |
| 技能9 | 80 | 110 | 100 |

算法一：暴力求解

关键代码：

#include <iostream>

#include <algorithm>

using namespace std;

struct Skill {

string name; // 技能名称

int cost; // 消耗天赋值

int attack; // 获得的攻击力

int defense; // 获得的防御力

};

bool cmp(Skill a, Skill b) {

return a.defense > b.defense; // 按防御力从大到小排序

}

int getMaxDefense(Skill \*skills, int n, int talent) {

int maxDefense = 0; // 角色的最大防御力

for (int i = 0; i < (1 << n); i++) { // 枚举所有的学习情况

int defense = 0; // 角色的防御力

int cost = 0; // 消耗的天赋值

for (int j = 0; j < n; j++) { // 枚举所有的技能

if (i & (1 << j)) { // 如果当前技能被学习

if (cost + skills[j].cost > talent) { // 如果当前的天赋值已经不够学习当前技能了

break; // 则跳出循环

}

cost += skills[j].cost; // 增加天赋值

defense += skills[j].defense; // 增加防御力

}

}

maxDefense = max(maxDefense, defense); // 取最大防御力

}

return maxDefense; // 返回角色的最大防御力

}

int main() {

int talent = 200; // 角色天赋值

Skill skills[9] = {

{"技能1", 10, 10, 10},

{"技能2", 20, 50, 20},

{"技能3", 30, 40, 80},

{"技能4", 40, 60, 20},

{"技能5", 50, 40, 80},

{"技能6", 15, 50, 20},

{"技能7", 55, 70, 20},

{"技能8", 70, 100, 30},

{"技能9", 80, 110, 100}

}; // 技能信息

sort(skills, skills + 9, cmp); // 将技能按防御力从大到小排序

int totalDefense = getMaxDefense(skills, 9, talent); // 角色最大防御力

cout << "本次使用暴力求解算法：" << endl;

cout << "角色的最大防御力为：" << totalDefense << endl;

return 0;

}

运行结果截图：



算法二：贪心算法

关键代码：

int getMaxDefense(Skill \*skills, int n, int talent) {

int totalDefense = 0;

for (int i = 0; i < 9; i++) {

if (talent >= skills[i].cost) { // 如果角色的天赋值足够学习当前技能

talent -= skills[i].cost; // 扣除天赋值

totalDefense += skills[i].defense; // 增加防御力

}

}

return totalDefense; // 返回角色的最大防御力

}

运行结果截图：



算法三：动态规划

关键代码：

int getMaxDefense(Skill \*skills, int n, int talent) {

int\* dp = new int[talent + 1]; // dp数组

for (int i = 0; i <= talent; i++) { // 初始化dp数组

dp[i] = 0;

}

for (int i = 0; i < n; i++) { // 枚举每个技能

for (int j = talent; j >= skills[i].cost; j--) { // 枚举天赋值

dp[j] = max(dp[j], dp[j - skills[i].cost] + skills[i].defense); // 更新dp值

}

}

return dp[talent]; // 返回角色的最大防御力

}

运行结果截图：

