【贪心】-虚拟游戏理财

题目描述与示例

题目描述

在一款虚拟游戏中生活，你必须进行投资以增强在虚拟游戏中的资产以免被淘汰出局。现有一家Bank，它提供有若干理财产品 `m` ，风险及投资回报不同，你有 `N` （元）进行投资，能接受的总风险值为 `X` 。

你要在可接受范围内选择最优的投资方式获得最大回报。

说明：

1、在虚拟游戏中，每项投资风险值相加为总风险值；

2、在虚拟游戏中，最多只能投资 `2` 个理财产品；

3、在虚拟游戏中，最小单位为整数，不能拆分为小数；

投资额*回报率=投资回报

输入描述

第一行：产品数(取值范围 `[1, 20]`)，总投资额(整数，取值范围 `[1, 10000]`)，可接受的总风险(整数，取值范围 `[1, 200]`)

第二行：产品投资回报率序列，输入为整数，取值范围 `[1,60]`

第三行：产品风险值序列，输入为整数，取值范围 `[1,100]`

第四行：最大投资额度序列，输入为整数，取值范围 `[1,10000]`

输出描述

每个产品的投资额序列

补充说明

1、在虚拟游戏中，每项投资风险值相加为总风险值；

2、在虚拟游戏中，最多只能投资 2 个理财产品；

3、在虚拟游戏中，最小单位为整数，不能拆分为小数；

投资额*回报率=投资回报

示例

输入

```
5 100 10
10 20 30 40 50
3 4 5 6 10
20 30 20 40 30
```

输出

```
0 30 0 40 0
```

说明：

投资第二项 `30` 个单位，第四项 `40` 个单位，总的投资风险为两项相加为 `4+6=10`

解题思路

本题难度不高，但是背景陌生以及条件较多，需要耐心读题完成。

首先观察本题的数据范围，产品数 `n` 最大只有 `20` 。由于最多只会取 `2` 个产品，所以枚举所有的产品对需要 `O(n^2)` 的时间复杂度，在题目所给的数据范围下是该复杂度是可以接受的。

枚举所有产品对非常容易实现，直接使用两个 `for` 循环即可实现。即

```
for i in range(n):
    for j in range(i+1, n):
        '''
        do something
        '''
        pass
```

当我们得到一个产品对的编号 `(i, j)` 时，需要考虑以下若干事情：

1. 判断两个产品的风险值的情况。如果两个产品风险值相加超过了 `X`，即 `risk_lst[i] + risk_lst[j] > X`，则必然不会选择这对二元组，直接跳过。
2. 两个产品的最大可投资额的相加结果 `max_amount_lst[i] + max_amount_lst[j]`，是否超过最大总投资额 `total_amount`。若
    - 否。则产品 `i` 和 `j` 的选择份额 `i_amount` 和 `j_amount` 即为它们各自的最大可投资额 `max_amount_lst[i]` 和 `max_amount_lst[j]`。这属于一种贪心策略。
    - 是。则进行后续的判断。
3. 产品 `i` 和 `j` 中选择单份份额回报率更高的产品，将其尽可能地选满。以如果单份产品 `i` 的投资回报率更高为例（`j` 回报率更高的情况也是类似的），即 `return_rate_lst[i] >= return_rate_lst[j]` 成立时，存在
    - 产品 `i` 的选择份额，为产品 `i` 的最大可投资额 `max_amount_lst[i]` 和总最大额 `total_amount` 之间的较小值，即 `i_amount = min(max_amount_lst[i], total_amount)`
    - 产品 `j` 的份额，为总最大额减 `total_amount` 去产品 `i` 的选择份额 `i_amount`，即 `j_amount = total_amount - i_amount`
    - 这也属于一种贪心策略
4. 根据选择份额 `i_amount` 和 `j_amount`，计算当前回报值 `cur_return`，更新全局的最大回报值和答案对。
    - 可以用以下格式储存具有最大回报值的产品编号以及选择份额
    - `pairs = [[i, i_amount], [j, j_amount]]`

上述过程可能会错误地掉一些可能的只选择单个产品的情况。当 `risk_lst[i]` 和 `risk_lst[j]` 的和大于 `X`，但 `risk_lst[i]` 或 `risk_lst[j]` 本身并不大于 `X` 时，无法获得选择单个产品的情况。

因此还需要再加一个 `for` 循环判断只选择一种产品的情况。

```python
for i in range(n):
    if risk_lst[i] > X:
        continue
    i_amount = min(max_amount_lst[i], total_amount)
    cur_return = i_amount * return_rate_lst[i]
    if cur_return > max_return:
        max_return = cur_return
        pairs = [[0, 0], [i, i_amount]]
```

代码

Python

```python
# 题目：【贪心】2024E-虚拟游戏理财
# 分值：200
# 作者：闭着眼睛学数理化
# 算法：贪心
# 代码看不懂的地方，请直接在群上提问


# 输入产品数量n，最大总投资额total_amount，最高风险系数X
n, total_amount, X = map(int, input().split())

# 输入长度为n的回报率列表
return_rate_lst = list(map(int, input().split()))

# 输入长度为n的风险值列表
risk_lst = list(map(int, input().split()))

# 输入长度为n的最大投资额序列
max_amount_lst = list(map(int, input().split()))

# 初始化总的最大回报值为0
max_return = 0
# 储存当前最大会值对应的产品编号以及所选取的份额数
pairs = [[0, 0], [0, 0]]

# 双重循环，遍历所有产品对（二元组）
for i in range(n):
    for j in range(i+1, n):
        # 考虑风险值的影响，如果两个产品的风险值加起来超过了X，则这组二元组不能选择
        # 直接跳过
        if risk_lst[i] + risk_lst[j] > X:
            continue
        # 如果两个产品的最大可投资额加起来，也不超过最大总投资额total_amount
        # 那么会贪心地将两个产品都选满
        # 即产品i选择max_amount_lst[i]份，产品j选择max_amount_lst[j]份
        if max_amount_lst[i] + max_amount_lst[j] <= total_amount:
            i_amount, j_amount = max_amount_lst[i], max_amount_lst[j]
        # 否则，我们必须在两个产品之间选择【单份产品的投资回报率】更高的产品
        # 贪心地尽可能选择它
        else:
```

```
            # 如果单份产品i的投资回报率更高
            if return_rate_lst[i] >= return_rate_lst[j]:
                # 产品i的份额，为产品i的最大额和总最大额之间的较小值
                i_amount = min(max_amount_lst[i], total_amount)
                # 产品j的份额，为总最大额减去产品i的份额
                j_amount = total_amount - i_amount
            # 如果单份产品j的投资回报率更高
            else:
                # 产品j的份额，为产品j的最大额和总最大额之间的较小值
                j_amount = min(max_amount_lst[j], total_amount)
                # 产品j的份额，为总最大额减去产品i的份额
                i_amount = total_amount - j_amount

        # 计算得到对应的当前回报值cur_return
        cur_return = i_amount * return_rate_lst[i] + j_amount *
return_rate_lst[j]
        # 如果当前计算得到的回报值比之前记录过的最大回报值更大
        # 则更新最大回报值以及pairs
        if cur_return > max_return:
            max_return = cur_return
            pairs = [[i, i_amount], [j, j_amount]]


# 考虑只选择1种产品i的情况
for i in range(n):
    if risk_lst[i] > X:
        continue
    # 产品i的份额，为产品i的最大额和总最大额之间的较小值
    i_amount = min(max_amount_lst[i], total_amount)
    cur_return = i_amount * return_rate_lst[i]
    # 如果当前计算得到的回报值比之前记录过的最大回报值更大
    # 则更新最大回报值以及pairs
    if cur_return > max_return:
        max_return = cur_return
        pairs = [[0, 0], [i, i_amount]]


# 构建答案序列，除了最终的i和j位置需要调整为最优的i_amount和j_amount，
# 其他位置所选取的份额都是0
ans = [0] * n
ans[pairs[0][0]] = pairs[0][1]
ans[pairs[1][0]] = pairs[1][1]

print(" ".join(str(num) for num in ans))
```

Java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();
        int totalAmount = scanner.nextInt();
        int X = scanner.nextInt();

        int[] returnRateLst = new int[n];
        int[] riskLst = new int[n];
        int[] maxAmountLst = new int[n];

        for (int i = 0; i < n; i++) {
            returnRateLst[i] = scanner.nextInt();
        }

        for (int i = 0; i < n; i++) {
            riskLst[i] = scanner.nextInt();
        }

        for (int i = 0; i < n; i++) {
            maxAmountLst[i] = scanner.nextInt();
        }

        int maxReturn = 0;
        int[][] pairs = new int[2][2];

        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (riskLst[i] + riskLst[j] > X) {
                    continue;
                }

                int iAmount, jAmount;
                if (maxAmountLst[i] + maxAmountLst[j] <= totalAmount) {
                    iAmount = maxAmountLst[i];
                    jAmount = maxAmountLst[j];
                } else {
                    if (returnRateLst[i] >= returnRateLst[j]) {
                        iAmount = Math.min(maxAmountLst[i], totalAmount);
                        jAmount = totalAmount - iAmount;
                    } else {
                        jAmount = Math.min(maxAmountLst[j], totalAmount);
                        iAmount = totalAmount - jAmount;
```

```
                    }
                }

                int curReturn = iAmount * returnRateLst[i] + jAmount *
returnRateLst[j];

                if (curReturn > maxReturn) {
                    maxReturn = curReturn;
                    pairs[0][0] = i;
                    pairs[0][1] = iAmount;
                    pairs[1][0] = j;
                    pairs[1][1] = jAmount;
                }
            }
        }


        for (int i = 0; i < n; i++){
            if (riskLst[i] > X) {
                    continue;
            }
            int iAmount = Math.min(maxAmountLst[i], totalAmount);
            int curReturn = iAmount * returnRateLst[i];
            if (curReturn > maxReturn) {
                maxReturn = curReturn;
                pairs[0][0] = 0;
                pairs[0][1] = 0;
                pairs[1][0] = i;
                pairs[1][1] = iAmount;
            }
        }


        int[] ans = new int[n];
        ans[pairs[0][0]] = pairs[0][1];
        ans[pairs[1][0]] = pairs[1][1];

        for (int i = 0; i < n; i++) {
            System.out.print(ans[i] + " ");
        }
    }
}

```

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, totalAmount, X;
    cin >> n >> totalAmount >> X;

    vector<int> returnRateLst(n);
    vector<int> riskLst(n);
    vector<int> maxAmountLst(n);

    for (int i = 0; i < n; i++) {
        cin >> returnRateLst[i];
    }

    for (int i = 0; i < n; i++) {
        cin >> riskLst[i];
    }

    for (int i = 0; i < n; i++) {
        cin >> maxAmountLst[i];
    }

    int maxReturn = 0;
    vector<vector<int>> pairs(2, vector<int>(2, 0));

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (riskLst[i] + riskLst[j] > X) {
                continue;
            }

            int iAmount, jAmount;
            if (maxAmountLst[i] + maxAmountLst[j] <= totalAmount) {
                iAmount = maxAmountLst[i];
                jAmount = maxAmountLst[j];
            } else {
                if (returnRateLst[i] >= returnRateLst[j]) {
                    iAmount = min(maxAmountLst[i], totalAmount);
                    jAmount = totalAmount - iAmount;
                } else {
                    jAmount = min(maxAmountLst[j], totalAmount);
                    iAmount = totalAmount - jAmount;
                }
            }

```

```
            int curReturn = iAmount * returnRateLst[i] + jAmount *
returnRateLst[j];

            if (curReturn > maxReturn) {
                maxReturn = curReturn;
                pairs[0][0] = i;
                pairs[0][1] = iAmount;
                pairs[1][0] = j;
                pairs[1][1] = jAmount;
            }
        }
    }

    for (int i = 0; i < n; i++){
        if (riskLst[i] > X) {
                continue;
        }
        int iAmount = min(maxAmountLst[i], totalAmount);
        int curReturn = iAmount * returnRateLst[i];
        if (curReturn > maxReturn) {
            maxReturn = curReturn;
            pairs[0][0] = 0;
            pairs[0][1] = 0;
            pairs[1][0] = i;
            pairs[1][1] = iAmount;
        }
    }



    vector<int> ans(n, 0);
    ans[pairs[0][0]] = pairs[0][1];
    ans[pairs[1][0]] = pairs[1][1];

    for (int i = 0; i < n; i++) {
        cout << ans[i] << " ";
    }

    return 0;
}

```

时空复杂度

时间复杂度： `O(N^2)` 。双重循环所需时间复杂度

空间复杂度： `O(1)` 。除了输入的序列，仅需若干常数变量维护遍历过程。