算法思想

- 贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。
- 贪心选择是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素。
- 当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。运用贪心策略在每一次转化时都取得了最优解。问题的最优子结构性质是该问题可用贪心算法求解的关键特征。贪心算法的每一次操作都对结果产生直接影响。贪心算法对每个子问题的解决方案都做出选择，不能回退。
- 贪心算法的基本思路是从问题的某一个初始解出发一步一步地进行，根据某个优化测度，每一步都要确保能获得局部最优解。每一步只考虑一个数据，他的选取应该满足局部优化的条件。若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中，直到把所有数据枚举完，或者不能再添加算法停止。
- 实际上，贪心算法适用的情况很少。一般对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可以做出判断。

过程

1. 建立数学模型来描述问题；
2. 把求解的问题分成若干个子问题；
3. 对每一子问题求解，得到子问题的局部最优解；
4. 把子问题的局部最优解合成原来解问题的一个解。

用白话说，即假设一个问题比较复杂，暂时找不到全局最优解，那么我们可以考虑把原问题拆成几个小问题（分而治之思想），分别求每个小问题的最优解，再把这些“局部最优解”叠起来，就“当作”整个问题的最优解了。

该算法存在的问题

- 不能保证求得的最后解是最佳的
- 不能用来求最大值或最小值的问题
- 只能求满足某些约束条件的可行解的范围

案例

1 选择排序

```
/*
我们熟知的选择排序，其实采用的即为贪心策略。
它所采用的贪心策略即为每次从未排序的数据中选取最小值，并把最小值放在未排序数据的起始位置，直到未排序的数据为0，则结束排序。
*/

void swap(int* array, int i, int j)
{
    int tmp = array[i];
    array[i] = array[j];
    array[j] = tmp;
}
```

```
void selectSort(int* arr, int n){
    //i: 未排序数据的起始位置
    for(int i = 0; i < n; ++i)
    {
        int minIdx = i;
        //从所有未排序的数据中找最小值的索引
        for(int j = i + 1; j < n; ++j){
            if(arr[j] < arr[minIdx])
               minIdx = j;
        }
        swap(arr, minIdx, i);
    }
}

/*java*/
public static void swap(int[] arr, int i, int j){
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }

public static void selectSort(int[] arr){
        //i: 未排序数据的起始位置
        for(int i = 0; i < arr.length; ++i)
        {
            int minIdx = i;
            //从所有未排序的数据中找最小值的索引
            for(int j = i + 1; j < arr.length; ++j){
                if(arr[j] < arr[minIdx])
                    minIdx = j;
            }
            swap(arr, minIdx, i);
        }
    }
```

2.平衡字符串

在一个「平衡字符串」中，'L' 和 'R' 字符的数量是相同的。

给出一个平衡字符串 s，请你将它分割成尽可能多的平衡字符串。

返回可以通过分割得到的平衡字符串的最大数量。

```
/*
题目要求通过分割得到的最大数量，即尽可能多的分割。
贪心策略：不要有嵌套的平衡，只要达到平衡，就立即分割。
故可以定义一个变量balance，在遇到不同的字符时，向不同的方向变化，当balance为0时达到平衡，分割数更新。
比如：
从左往右扫描字符串s，遇到L,balance - 1，遇到R，balance + 1
当balance为0时即，更新记录cnt ++
如果最后cnt==0，说明s只需要保持原样，返回1
*/
```

```
class Solution {
public:
    int balancedStringSplit(string s) {
        int cnt = 0;
        int balance = 0;
        for(int i = 0; i < s.size(); i++){
            if(s[i] == 'L')
                balance--;
            if(s[i] == 'R')
                balance++;
            if(balance == 0)
                cnt++;
        }
        return cnt;
    }
};

/*java*/
class Solution {
    public int balancedStringSplit(String s) {
        int cnt = 0;
        int balance = 0;
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == 'L')
                balance--;
            if(s.charAt(i) == 'R')
                balance++;
            if(balance == 0)
                cnt++;
        }
        return cnt;
    }
}
```

3. 买卖股票的最佳时机 II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

```
/*
贪心策略
连续上涨交易日：  则第一天买最后一天卖收益最大，等价于每天都买卖
连续下降交易日：  则不买卖收益最大，即不会亏钱。
故可以遍历整个股票交易日价格列表，在所有上涨交易日都买卖（赚到所有利润），所有下降交易日都不买卖（永不亏钱）。
*/
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int ret = 0;
```

```
        for(int i = 1; i < prices.size(); ++i)
        {
            int curProfit = prices[i] - prices[i - 1];
            if(curProfit > 0)
                ret += curProfit;
        }
        return ret;
    }
};

/*java*/
class Solution {
    public int maxProfit(int[] prices) {
        int ret = 0;
        for(int i = 1; i < prices.length; ++i)
        {
            int curProfit = prices[i] - prices[i - 1];
            if(curProfit > 0)
                ret += curProfit;
        }
        return ret;
    }
}
```

4. 跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

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位置。

```
/*
贪心策略

设想一下，对于数组中的任意一个位置 y，我们如何判断它是否可以到达？根据题目的描述，只要存在一个位置 x，它本身可以到达，并且它跳跃的最大长度为 x + nums[x]，这个值大于等于 y，即 x+nums[x] ≥ y，那么位置y也可以到达。

换句话说，对于每一个可以到达的位置 x，它使得 x+1，x+2,…,x+nums[x] 这些连续的位置都可以到达。

这样以来，我们依次遍历数组中的每一个位置，并实时维护最远可以到达的位置。对于当前遍历到的位置x，如果它在 最远可以到达的位置的范围内，那么我们就可以从起点通过若干次跳跃到达该位置，因此我们可以用 x+nums[x]更新 最远可以到达的位置。

在遍历的过程中，如果最远可以到达的位置大于等于数组中的最后一个位置，那就说明最后一个位置可达，我们就可以直接返回True作为答案。反之，如果在遍历结束后，最后一个位置仍然不可达，我们就返回 False 作为答案。

以题目中的示例一
[2，3，1，1，4]
为例:

我们一开始在位置 0，可以跳跃的最大长度为 2，因此最远可以到达的位置被更新为 2;
```

我们遍历到位置 1，由于 1≤2，因此位置 1 可达。我们用 1加上它可以跳跃的最大长度3，将最远可以到达的位置更新为4。由于4大于等于最后一个位置4，因此我们直接返回 True。

我们再来看看题目中的示例二

[3, 2, 1, 0, 4]
我们一开始在位置0，可以跳跃的最大长度为3，因此最远可以到达的位置被更新为3；

我们遍历到位置1，由于1≤3，因此位置1可达，加上它可以跳跃的最大长度2得到3，没有超过最远可以到达的位置；

位置 2、位置 3 同理，最远可以到达的位置不会被更新；

我们遍历到位置4，由于4>3，因此位置4不可达，我们也就不考虑它可以跳跃的最大长度了。

在遍历完成之后，位置4仍然不可达，因此我们返回 False。
*/

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n = nums.size();
        int rightmost = 0;
        for (int i = 0; i < n; ++i) {
            //如果可以到达当前位置，则更新最大
            if (i <= rightmost) {
                //每次更新最大的位置
                rightmost = max(rightmost, i + nums[i]);
                //如果可以到达最后一个位置，则直接返回
                if (rightmost >= n - 1) {
                    return true;
                }
            }
        }
        return false;
    }
};

/*java*/
public class Solution {
    public boolean canJump(int[] nums) {
        int n = nums.length;
        int rightmost = 0;
        for (int i = 0; i < n; ++i) {
            //如果可以到达当前位置，则更新最大
            if (i <= rightmost) {
                //每次更新最大的位置
                rightmost = Math.max(rightmost, i + nums[i]);
                //如果可以到达最后一个位置，则直接返回
                if (rightmost >= n - 1) {
                    return true;
                }
            }
        }
        return false;
```

```
    }
}
```

5 钱币找零

假设1元、2元、5元、10元、20元、50元、100元的纸币分别有c0, c1, c2, c3, c4, c5, c6张。现在要用这些钱来支付K元，至少要用多少张纸币?

```
/*
贪心策略:

用贪心算法的思想，很显然，每一步尽可能用面值大的纸币即可。在日常生活中我们自然而然也是这么做的。在程序中已经事先将Value按照从小到大的顺序排好。

*/
#include<iostream>
#include<algorithm>
using namespace std;

int solve(int money,const vector<pair<int,int>>& moneyCount)
{
    int num = 0;
    //首先选择最大面值的纸币
    for (int i = moneyCount.size() - 1; i >= 0; i--)
    {
        //需要的当前面值与面值数量取最小
        int c = min(money / moneyCount[i].first, moneyCount[i].second);
        money = money - c * moneyCount[i].first;
        num += c;
    }
    if (money > 0)
        num = -1;
    return num;
}

int main()
{
    //存放纸币与数量: first:纸币, second:数量
    vector<pair<int, int>> moneyCount = { { 1, 3 }, { 2, 1 }, { 5, 4 }, { 10, 3 }, { 20, 0
}
    ,{50, 1}, { 100, 10 } };

    int money;
    cout << "请输入要支付的钱" << endl;
    cin >> money;

    int res = solve(money, moneyCount);
    if (res != -1)
        cout << res << endl;
    else
        cout << "NO" << endl;
    return 0;
}
```

```java
/*java*/
public static int solve(int money, int[][] moneyCount)
    {
        int num = 0;
        //首先选择最大面值的纸币
        for (int i = moneyCount.length - 1; i >= 0; i--)
        {
            //需要的当前面值与面值数量取最小
            int c = Math.min(money / moneyCount[i][0], moneyCount[i][1]);
            money = money - c * moneyCount[i][0];
            num += c;
        }
        if (money > 0)
            num = -1;
        return num;
    }

    public static void main(String[] args) {
        //存放纸币与数量: first:纸币, second:数量
        int[][] moneyCount = { { 1, 3 }, { 2, 1 }, { 5, 4 }, { 10, 3 }, { 20, 0 }
                ,{50, 1}, { 100, 10 } };
        Scanner scanner = new Scanner(System.in);
        int money;
        System.out.println("请输入要支付的钱");
        money = scanner.nextInt();

        int res = solve(money, moneyCount);
        if (res != -1)
            System.out.println(res);
        else
            System.out.println("No");
    }
```

6 多机调度问题

某工厂有n个独立的作业，由m台相同的机器进行加工处理。作业i所需的加工时间为ti，任何作业在被处理时不能中断，也不能进行拆分处理。现厂长请你给他写一个程序：算出n个作业由m台机器加工处理的最短时间

输入

第一行T（1<T<100)表示有T组测试数据。每组测试数据的第一行分别是整数n，m（1<=n<=10000，1<=m<=100），接下来的一行是n个整数ti（1<=t<=100)。

输出

所需的最短时间

```java
/*
这个问题还没有有效的解法(求最优解)，但是可以用贪心选择策略设计出较好的近似算法(求次优解)。

贪心策略
```

当n<=m时，只要将作业分给每一个机器即可；当n>m时，首先将n个作业从大到小排序，然后依此顺序将作业分配给空闲的处理机。也就是说从剩下的作业中，选择需要处理时间最长的，然后依次选择处理时间次长的，直到所有的作业全部处理完毕，或者机器不能再处理其他作业为止。如果我们每次是将需要处理时间最短的作业分配给空闲的机器，那么可能就会出现其它所有作业都处理完了只剩所需时间最长的作业在处理的情况，这样势必效率较低。

例如

n =6 m =3

ti : 2 5 13 15 16 20

speed数组: 20 16 15 13 5 2

这里重点是mintime数组的变化:

初始数组为空

0 0 0 (因为m = 3 ，所以3个是有效的，其余为0不考虑)

20 16 15 (依次存入)

20 16 15+13 (下面代码中的min_element函数功能) 15最小

20 16+5 28 (16最小，新数组中)

20+2 21 28 (20最小，新数组中)

--

28 (所有作业完成的最短时间)
*/

```
bool cmp(const int &x, const int &y)
{
    return x > y;
}

int findMax(const vector<int>& machines)
{
    int ret = machines[0];
    for (const auto& cur : machines)
    {
        if (ret < cur)
            ret = cur;
    }
    return ret;
}
int greedStrategy(const vector<int>& works, vector<int>& machines) {
    // 按作业时间从大到小排序
    sort(works.begin(), works.end(), cmp);
    int workNum = works.size();
    int machineNum = machines.size();
```

```
    // 作业数如果小于机器数，直接返回最大的作业时间
    if (workNum <= machineNum) {
        int minimalTime = works[0];
        return minimalTime;
    }
    else {
        // 为每一个作业选择机器
        for (int i = 0; i < workNum; i++) {
            // 选择最小的机器
            int flag = 0;
            //首先假设用第一个机器处理
            int tmp = machines[flag];
            // 从剩余机器中选择作业时间最小的
            for (int j = 1; j < machines.size(); j++) {
                if (tmp > machines[j]) {
                    flag = j;
                    tmp = machines[j];
                }
            }
            // 将当前作业交给最小的机器执行
            machines[flag] += works[i];
        }
        // 从所有机器中选出最后执行完作业的机器
        int minimalTime = findMax(machines);
        return minimalTime;
    }
}

int main()
{
    int n, m;
    cout << "请输入作业数和机器数" << endl;
    cin >> n >> m;
    vector<int> works(n);
    vector<int> machines(m,0);

    for (int i = 0; i < n; ++i)
        cin >> works[i];

    cout << greedStrategy(works, machines) << endl;
    return 0;
}

/*java*/
class myComparator implements Comparator<int[]> {
        //按作业时间从大到小排序
        public int compare(int a, int b) {
            return b - a;
        }
    }

public static int findMax(int[] machines){
        int ret = machines[0];
```

```
        for (int cur : machines)
        {
            if (ret < cur)
                ret = cur;
        }
        return ret;
    }

private static int greedStrategy(int[] works, int[] machines) {
        // 按作业时间从大到小排序
        Arrays.sort(works, new myComparator());
        // 作业数如果小于机器数，直接返回最大的作业时间
        if (workNum <= machineNum) {
            minimalTime = works[0];
            return minimalTime;
        } else {
            // 为每一个作业选择机器
            for (int i = 0; i < workNum; i++) {
                // 选择最小的机器
                int flag = 0;
                //首先假设用第一个机器处理
                int tmp = machines[flag];
                // 从剩余机器中选择作业时间最小的
                for (int j = 1; j < machines.length; j++) {
                    if (tmp > machines[j]) {
                        flag = j;
                        tmp = machines[j];
                    }
                }
                // 将当前作业交给最小的机器执行
                machines[flag] += works[i];
            }
            // 从所有机器中选出最后执行完作业的机器
            minimalTime = findMax(machines);
            return minimalTime;
        }
    }

public static void main(String[] args) {
        int n, m;
        System.out.println("请输入作业数和机器数");
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();
        m = scanner.nextInt();

        int[] works = new int[n];
        int[] machines = new int[m];

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

        System.out.println(greedStrategy(works, machines));
```

```
    }
```

7 活动选择

有n个需要在同一天使用同一个教室的活动a1, a2, ..., an，教室同一时刻只能由一个活动使用。每个活动a[i]都有一个开始时间s[i]和结束时间f[i]。一旦被选择后，活动a[i]就占据半开时间区间[s[i],f[i])。如果[s[i],f[i])和[s[j],f[j])互不重叠，a[i]和a[j]两个活动就可以被安排在这一天。求使得尽量多的活动能不冲突的举行的最大数量。

贪心策略
1．每次都选择开始时间最早的活动，不能得到最优解。

2．每次都选择持续时间最短的活动，不能得到最优解。

3．每次选取结束时间最早的活动，可以得到最优解。按这种方法选择，可以为未安排活动留下尽可能多的时间。如下图所示的活动集合S，其中各项活动按照结束时间单调递增排序。

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

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

bool cmp(const pair<int,int>& a, const pair<int,int>& b)
{
    return a.second < b.second;
}

int greedyActivitySelector(const vector<pair<int,int>>& act)
{
    //贪婪策略：每次选择最早结束的活动
    int num = 1, i = 0;
    for (int j = 1; j < act.size(); j++)
    {
```

```
        if (act[j].first >= act[i].second)
        {
            i = j;
            num++;
        }
    }
    return num;
}

int main()
{
    int number;
    cin >> number;
    vector<pair<int, int>> act(number);
    int idx = 0;
    for (int i = 0; i < act.size(); ++i)
    {
        cin >> act[i].first >> act[i].second;
    }

    //按照活动截止时间从小到大排序
    sort(act.begin(), act.end(), cmp);

    int ret = greedyActivitySelector(act);
    cout << ret << endl;
}

/*java*/
class myComparator implements Comparator<int[]> {
    //按起点递增排序
    public int compare(int[] a, int[] b) {
        return a[1] - b[1];
    }
}

public static int greedyActivitySelector(int[][] act)
    {
        //贪婪策略：每次选择最早结束的活动
        int num = 1, i = 0;
        for (int j = 1; j < act.length; j++)
        {
            if (act[j][0] >= act[i][1])
            {
                i = j;
                num++;
            }
        }
        return num;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int number = scanner.nextInt();
```

```
        int[][] act = new int[number][2];
        int idx = 0;
        for (int i = 0; i < act.length; ++i)
        {
            act[i][0] = scanner.nextInt();
            act[i][1] = scanner.nextInt();
        }
        //按照活动截止时间从小到大排序
        Arrays.sort(act, new myComparator());

        int ret = greedyActivitySelector(act);
        System.out.println(ret);
    }
```

相似题目：

最多可以参加的会议数目

8. 无重叠区间

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

可以认为区间的终点总是大于它的起点。 区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

如果我们按照起点对区间进行排序，贪心算法就可以起到很好的效果。
贪心策略：
当按照起点先后顺序考虑区间的时候。我们利用一个 prev 指针追踪刚刚添加到最终列表中的区间。遍历的时候，可能遇到图中的三种情况：

情况一

当前考虑的两个区间不重叠：

在这种情况下，不移除任何区间，将 prev 赋值为后面的区间，移除区间数量不变。

情况二

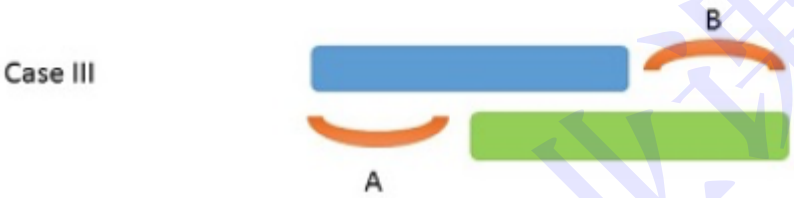
两个区间重叠，后一个区间的终点在前一个区间的终点之前。

这种情况下，我们可以简单地只用后一个区间。这是显然的，因为后一个区间的长度更小，可以留下更多的空间（AA 和 BB），容纳更多的区间。因此， prev 更新为当前区间，移除区间的数量 + 1。

情况三

两个区间重叠，后一个区间的终点在前一个区间的终点之后。

这种情况下，我们用贪心策略处理问题，直接移除后一个区间。这样也可以留下更多的空间，容纳更多的区间。因此，prev 不变，移除区间的数量 + 1。




```
bool cmp(const vector<int>& a, const vector<int>& b)
{
    //按起点递增排序
    return a[0] < b[0];
}
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.size() == 0) {
            return 0;
        }
        //按起点递增排序
        sort(intervals.begin(), intervals.end(), cmp);

        int end = intervals[0][0], prev = 0, count = 0;
        for (int i = 1; i < intervals.size(); i++) {
            if (intervals[prev][1] > intervals[i][0]) {
                if (intervals[prev][1] > intervals[i][1]) {
                    //情况2
                    prev = i;
                }
                //情况3
                count++;
            } else {
                //情况1
                prev = i;
            }
        }
        return count;
```

```
    }
};

/*java*/
public class Solution {
    class myComparator implements Comparator<int[]> {
        //按起点递增排序
        public int compare(int[] a, int[] b) {
            return a[0] - b[0];
        }
    }
    public int eraseOverlapIntervals(int[][] intervals) {
        if (intervals.length == 0) {
            return 0;
        }
        //起点排序
        Arrays.sort(intervals, new myComparator());

        int end = intervals[0][0], prev = 0, count = 0;
        for (int i = 1; i < intervals.length; i++) {
            if (intervals[prev][1] > intervals[i][0]) {
                if (intervals[prev][1] > intervals[i][1]) {
                    //情况2
                    prev = i;
                }
                //情况3
                count++;
            } else {
                //情况1
                prev = i;
            }
        }
        return count;
    }
}
```