DP：

0-1背包:

    for (int i=1; i<=N; i++)

        for (int j=M; j>=weight[i]; j--)

        {

                f[j]=max(f[j],f[j-weight[i]]+value[i]);

        }

// 你有一堆石头质量分别为W1,W2,W3…WN.(W＜＝100000,N <30)现在需要你将石头合并为两堆，使两堆质量的差为最小。

//给一个整数的集合，要把它分成两个集合，要两个集合的数的和最接近

//有一个箱子容量为V（正整数，0≤V≤20000），同时有n个物品（0小于n≤30），每个物品有一个体积（正整数）。要求从n个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

int main(){

    int n;

    scanf("%d", &n);

    vector<int> stone(n + 1, 0);

    int sum = 0;

    for(int i = 1; i <= n; ++i){

        scanf("%d", &stone[i]);

        sum += stone[i];

    }

    vector<int> res(sum / 2 + 1, 0);

    for(int i = 0; i <= n; ++i)

        for(int j = sum / 2; j >= stone[i]; --j)

            res[j] = max(res[j], res[j - stone[i]] + stone[i]);

    printf("%d",sum - 2 \* res[sum / 2]);

    getchar();

    getchar();

    return 0;

}

//判断是否有和为100的组合

//注意这里需要判断是01背包还是完全背包，完全背包的话J从左往右遍历即可。

bool findSum100(vector<int>& a) {

        vector<bool> dp(a.size()+1,false);

        dp[0] = true;

        for (int i = 1; i < a.size(); i++) {

            for (int j = 100; j >= a[i]; j--) {

                if (dp[j - a[i]]) {

                    dp[j] = true;

                }

            }

            if (dp[100]) {

                return true;

            }

        }

        return false;

}

//打印和为n的所有组合,DFS法，O(2^n)

void dfs(int current, int sum, int n, vector<int>& nums, vector<int>& path, vector<vector<int>>& list){

    if(sum == n){

        list.push\_back(path);

        return;

    }

    if(sum > n || current == nums.size())

        return;

    for(int i = current; i < nums.size(); ++i){

        path.push\_back(nums[i]);

        dfs(i + 1, sum + nums[i], n, nums, path, list);

        path.pop\_back();

    }

}

完全背包:

    for(int i = 1; i <= N; ++i)

        for(int j = weight[i]; j <= M; ++j)

            f[j] = max(f[j], f[j - weight[i]] + value[i]);

//找零钱问题：有n种面额的硬币，每种硬币无限多，至少用多少枚硬币表示给定的面值M？

#include <algorithm>

#include <vector>

#define INF 0x3f3f3f3f

using namespace std;

int main(){

    int n, m;

    scanf("%d %d", &n, &m);

    vector<int> coin(n + 1, 0);

    for(int i = 1; i <= n; ++i){

        scanf("%d", &coin[i]);

    }

    vector<int> dp(m + 1, INF);

    dp[0] = 0;

    for(int i = 1; i <= n; ++i)

        for(int j = coin[i]; j <= m; ++j)

            dp[j] = min(dp[j], dp[j - coin[i]] + 1);

    printf("%d", dp[m] <= m ? dp[m] : -1);

    getchar();

    getchar();

    return 0;

}

// 100元钱，有几种零钱，比如1元，5元，10元，求有几种组合 : 121种

int main(){

    int money;

    int n;

    scanf("%d %d", &money, &n);

    vector<int> coin(n + 1);

    vector<int> dp(money + 1, 0);

    dp[0] = 1;

    for(int i = 1; i <= n; ++i)

        scanf("%d", &coin[i]);

    for(int i = 1; i <= n; ++i){

        for(int j = coin[i]; j <= money; ++j){

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

        }

    }

    printf("%d", dp[money]);

    return 0;

}

分组背包:

// 有N件物品和一个容量为VV的背包。第i件物品的费用是w[i]，价值是v[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

int main(){

    int n , m;//n是多少组,m是重量

    scanf("%d %d", &n, &m);

    vector<int> weight(n + 1, 0);

    vector<int> value(n + 1, 0);

    vector<int> f(m + 1, 0);

    for(int i = 1; i <= n; ++i){

        int s;//s为这一组内数量

        scanf("%d", &s);

        for(int q = 1; q <= s; ++q)

            scanf("%d %d", &weight[q], &value[q]);//输入这一组内重量，价值

        for(int j = m; j > 0; --j){

            for(int k = 1; k <= s; ++k){

                if(j >= weight[k])

                    f[j] = max(f[j], f[j - weight[k]] + value[k]);//最后输出的是f[m],因为每次只保留一组的数据，所以下一组是根据上一组来进行估算,所以在f[m]上每组最多只有一个；

            }

        }

    }

    printf("%d", f[m]);

    return 0;

}

// 3 5

// 2

// 1 2

// 2 4

// 1

// 3 4

// 1

// 4 5

多重背包:

int mult(int n, int m, vector<int>& weight, vector<int>& value, vector<int> nums){

    vector<int> dp(m + 1, 0);

    // for(int i = 1; i <= n ; ++i)

    //     for(int k = 0; k < nums[i]; ++k)

    //         for(int j = m; j >= weight[i]; --j)

    //             dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

    for(int i = 1; i <= n; ++i){

        for(int k = 1; nums[i]; k << 1){

            if(k > nums[i])

                k = nums[i];

            nums[i] -= k;

            int nw = k \* weight[i];

            int nv = k \* value[i];

            for(int j = m; j >= nw; j--)

                dp[j] = max(dp[j], dp[j - nw] + nv);

        }

    }

    return dp[m];

}

// 设有n种不同面值的硬币，各硬币的面值存于数组T［1:n］中。现要用这些面值的硬币来找钱。可以使用的各种面值的硬币个数存于数组Coins［1:n］中。对任意钱数0≤m≤20001，设计一个用最少硬币找钱m的方法。

// 对于给定的1≤n≤10，硬币面值数组T和可以使用的各种面值的硬币个数数组Coins，以及钱数m，0≤m≤20001，编程计算找钱m的最少硬币数。

#define LL long long int

#define INF 0x3f3f3f3f

int dp[20020];

int main(){

    int i, j, k, n, m;

    cin >> n;

    int coins[n];  //硬币个数

    int T[n];      //硬币面值

    for(i = 0; i<n; i++)

        cin >> T[i] >> coins[i];

    cin >> m;

    for(i=1;i<=m;i++) dp[i]=INF;    //赋极大值,表示不可达

    for(i=0;i<n;i++)    //遍历硬币种类

        for(j=1;j<=coins[i];j++)    //遍历硬币数量

            for(k=m; k>=T[i]; k--)  //此处较难理解

                                    //只能是由m到T[i]而不能相反

                                    //试想,初始态dp[k-T[i]]应当=INF,dp[0]=0

                                    //如果能组成m元的硬币,那么应当存在一条0->m的路径,此时

                                    //dp[m]就是需要的硬币数量

                                    //否则,dp[m]将不能链接到dp[0],从而dp[m]=INF输出-1

                dp[k] = min(dp[k], dp[k-T[i]]+1);

    cout << (dp[m]<m?dp[m]:-1) << endl;

    return 0;

}

// 问题

// 二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价1和代价2，第i件物品所需的两种代价分别为a[i]和b[i]。两种代价可付出的最大值（两种背包容量）分别为V和U。物品的价值为w[i]。

// 费用加了一维，只需状态也加一维即可。设f[i][v][u]表示前i件物品付出两种代价分别为v和u时可获得的最大价值。状态转移方程就是：

// f[i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+w[i]}

// 有时，“二维费用”的条件是以这样一种隐含的方式给出的：最多只能取M件物品。这事实上相当于每件物品多了一种“件数”的费用，每个物品的件数费用均为1，可以付出的最大件数费用为M。换句话说，设f[v][m]表示付出费用v、最多选m件时可得到的最大价值，则根据物品的类型（01、完全、多重）用不同的方法循环更新，最后在f[0..V][0..M]范围内寻找答案。

    for(int i = 1; i <= N; ++i){

        for(int j = M1; j >= weight1[i]; --j){

            for(int k = M2; k >= weight2[i]; --k)

                f[j][k] = max(f[j][k], f[j - weight1[i]][k - weight2[i]] + value[i]);

        }

}

[135. 分发糖果](https://leetcode-cn.com/problems/candy/)

老师想给孩子们分发糖果，有 *N* 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。你需要按照以下要求，帮助老师给这些孩子分发糖果：

* 每个孩子至少分配到 1 个糖果。
* 相邻的孩子中，评分高的孩子必须获得更多的糖果。

那么这样下来，老师至少需要准备多少颗糖果呢？

**示例 1:**

**输入:** [1,0,2]

**输出:** 5

**解释:** 你可以分别给这三个孩子分发 2、1、2 颗糖果。

**示例 2:**

**输入:** [1,2,2]

**输出:** 4

**解释:** 你可以分别给这三个孩子分发 1、2、1 颗糖果。

第三个孩子只得到 1 颗糖果，这已满足上述两个条件。

class Solution {

public:

    int candy(vector<int>& ratings) {

        int res = 0;

        vector<int> tmp(ratings.size(), 1);

        for(int i = 1; i < ratings.size(); ++i){

            if(ratings[i] > ratings[i - 1])

                tmp[i] = tmp[i - 1] + 1;

        }

        for(int i = ratings.size() - 1; i > 0; --i){

            if(ratings[i - 1] > ratings[i] && tmp[i - 1] < tmp[i] + 1)

                tmp[i - 1] = tmp[i] + 1;

        }

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

            res += tmp[i];

        }

        return res;

    }

};

字符串: KMP

未改进版:

rank -1 0 1 2 3 4 5 6 7 8 9

p[] \* C H I N C H I L L A

next[] N/A -1 0 0 0 0 1 2 3 0 0

T 0 0 0 1 0 0 0 0 1 0

P 0 0 0 0 1 0

-1 0 1 2 3 0

未改进版:

rank -1 0 1 2 3 4 5

p[] \* 0 0 0 0 1 0

next[] N/A -1 0 1 2 3 0

改进版:

rank -1 0 1 2 3 4 5

p[] \* 0 0 0 0 1 0

next[] N/A -1 -1 -1 -1 3 -1

int\* buildNext(string P){

    int m = P.size();

    int j = 0;

    int\* N = new int[m];

    int t = N[0] = -1;//模式串指针

    while(j < m - 1){

        if(0 > t || P[j] == P[t]){//匹配

            j++;

            t++;

            //N[j] = t; //此句可改进

            N[j] = P[j] != P[t] ? t : N[t];

        }else//失配

        {

            t = N[t];

        }

    }

    return N;

}

int kmpMatch(string T, string P){

    int\* next = buildNext(P);//构造next表

    int n = T.size(), i = 0;//文本串指针

    int m = P.size(), j = 0;//模式串指针

    while(i < n && j < m){//自左向右逐个比对字符

        if(0 > j || T[i] == P[j]){//若匹配，或P已移出最左侧(两个判断的次序不可交换)

            i++;

            j++;//则转到下一字符

        }else//否则

        {

            j = next[j];//模式串右移（注意:文本串不用回退）

        }

    }

    delete[] next;//释放next表

    if(j == m)

        return i - j;

    return -1;

}

栈、队列、数组、链表、map

[1. 两数之和](https://leetcode-cn.com/problems/two-sum/)

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

**示例:**

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[**0**] + nums[**1**] = 2 + 7 = 9

所以返回 [**0, 1**]

 vector<int> twoSum(vector<int>& nums, int target) {

        vector<int> res;

        map<int, int> mp;

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

            int cur = target - nums[i];

            if(mp.find(cur) != mp.end()){

                res.push\_back(mp[cur]);

                res.push\_back(i);

                return res;

            }

            mp[nums[i]] = i;

        }

        return res;

    }

[15. 三数之和](https://leetcode-cn.com/problems/3sum/)

给你一个包含 *n* 个整数的数组 nums，判断 nums 中是否存在三个元素 *a，b，c ，*使得 *a + b + c =*0 ？请你找出所有满足条件且不重复的三元组。

**注意：**答案中不可以包含重复的三元组。

**示例：**

给定数组 nums = [-1, 0, 1, 2, -1, -4]，

满足要求的三元组集合为：

[

[-1, 0, 1],

[-1, -1, 2]

]

class Solution {

public:

    vector<vector<int>> twoSum(vector<int>& nums, int start, int end, int target, int value){

        vector<vector<int>> ans;

        while(start < end){

            int sum = nums[start] + nums[end];

            if(sum == target){

                vector<int> res;

                res.push\_back(value);

                res.push\_back(nums[start]);

                res.push\_back(nums[end]);

                ans.push\_back(res);

                while(start < end && nums[start] == nums[start + 1])

                    start++;

                start++;

                while(start < end && nums[end] == nums[end - 1])

                    end--;

                end--;

            }else if(sum < target){

                start++;

            }else{

                end--;

            }

        }

        return ans;

    }

    vector<vector<int>> threeSum(vector<int>& nums) {

        vector<vector<int>> ans;

        sort(nums.begin(), nums.end());

        int size = nums.size();

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

            if(i > 0 && nums[i] == nums[i - 1])

                continue;

            auto res = twoSum(nums, i + 1, size - 1, -nums[i], nums[i]);

            ans.insert(ans.end(), res.begin(), res.end());

        }

        return ans;

    }

};

[23. 合并K个升序链表](https://leetcode-cn.com/problems/merge-k-sorted-lists/)

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

**示例 1：**

**输入：**lists = [[1,4,5],[1,3,4],[2,6]]

**输出：**[1,1,2,3,4,4,5,6]

**解释：**链表数组如下：

[

1->4->5,

1->3->4,

2->6

]

将它们合并到一个有序链表中得到。

1->1->2->3->4->4->5->6

class Solution {

public:

    ListNode\* mergeTwoLists(ListNode \*a, ListNode \*b) {

        if ((!a) || (!b)) return a ? a : b;

        ListNode head, \*tail = &head, \*aPtr = a, \*bPtr = b;

        while (aPtr && bPtr) {

            if (aPtr->val < bPtr->val) {

                tail->next = aPtr; aPtr = aPtr->next;

            } else {

                tail->next = bPtr; bPtr = bPtr->next;

            }

            tail = tail->next;

        }

        tail->next = (aPtr ? aPtr : bPtr);

        return head.next;

    }

    ListNode\* merge(vector <ListNode\*> &lists, int l, int r) {

        if (l == r) return lists[l];

        if (l > r) return nullptr;

        int mid = (l + r) >> 1;

        return mergeTwoLists(merge(lists, l, mid), merge(lists, mid + 1, r));

    }

    ListNode\* mergeKLists(vector<ListNode\*>& lists) {

        return merge(lists, 0, lists.size() - 1);

    }

};

[146. LRU缓存机制](https://leetcode-cn.com/problems/lru-cache/)

运用你所掌握的数据结构，设计和实现一个  [LRU (最近最少使用) 缓存机制](https://baike.baidu.com/item/LRU)。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果关键字 (key) 存在于缓存中，则获取关键字的值（总是正数），否则返回 -1。  
写入数据 put(key, value) - 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字/值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。在**O(1)** 时间复杂度内完成这两种操作

**示例:**

LRUCache cache = new LRUCache( 2 /\* 缓存容量 \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // 返回 1

cache.put(3, 3); // 该操作会使得关键字 2 作废

cache.get(2); // 返回 -1 (未找到)

cache.put(4, 4); // 该操作会使得关键字 1 作废

cache.get(1); // 返回 -1 (未找到)

cache.get(3); // 返回 3

cache.get(4); // 返回 4

struct DLinkedNode {

    int key, value;

    DLinkedNode\* prev;

    DLinkedNode\* next;

    DLinkedNode(): key(0), value(0), prev(nullptr), next(nullptr) {}

    DLinkedNode(int \_key, int \_value): key(\_key), value(\_value), prev(nullptr), next(nullptr) {}

};

class LRUCache {

private:

    unordered\_map<int, DLinkedNode\*> cache;

    DLinkedNode\* head;

    DLinkedNode\* tail;

    int size;

    int capacity;

public:

    LRUCache(int \_capacity): capacity(\_capacity), size(0) {

        // 使用伪头部和伪尾部节点

        head = new DLinkedNode();

        tail = new DLinkedNode();

        head->next = tail;

        tail->prev = head;

    }

    int get(int key) {

        if (!cache.count(key)) {

            return -1;

        }

        // 如果 key 存在，先通过哈希表定位，再移到头部

        DLinkedNode\* node = cache[key];

        moveToHead(node);

        return node->value;

    }

    void put(int key, int value) {

        if (!cache.count(key)) {

            // 如果 key 不存在，创建一个新的节点

            DLinkedNode\* node = new DLinkedNode(key, value);

            // 添加进哈希表

            cache[key] = node;

            // 添加至双向链表的头部

            addToHead(node);

            ++size;

            if (size > capacity) {

                // 如果超出容量，删除双向链表的尾部节点

                DLinkedNode\* removed = removeTail();

                // 删除哈希表中对应的项

                cache.erase(removed->key);

                // 防止内存泄漏

                delete removed;

                --size;

            }

        }

        else {

            // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部

            DLinkedNode\* node = cache[key];

            node->value = value;

            moveToHead(node);

        }

    }

  void addToHead(DLinkedNode\* node) {

        node->prev = head;

        node->next = head->next;

        head->next->prev = node;

        head->next = node;

    }

    void removeNode(DLinkedNode\* node) {

        node->prev->next = node->next;

        node->next->prev = node->prev;

    }

    void moveToHead(DLinkedNode\* node) {

        removeNode(node);

        addToHead(node);

    }

    DLinkedNode\* removeTail() {

        DLinkedNode\* node = tail->prev;

        removeNode(node);

        return node;

    }

};

[150. 逆波兰表达式求值](https://leetcode-cn.com/problems/evaluate-reverse-polish-notation/)

根据[逆波兰表示法](https://baike.baidu.com/item/%E9%80%86%E6%B3%A2%E5%85%B0%E5%BC%8F/128437)，求表达式的值。

有效的运算符包括 +, -, \*, / 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

**说明：**

* 整数除法只保留整数部分。
* 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

**示例 1：**

**输入:** ["2", "1", "+", "3", "\*"]

**输出:** 9

**解释:** 该算式转化为常见的中缀算术表达式为：((2 + 1) \* 3) = 9

**示例 2：**

**输入:** ["4", "13", "5", "/", "+"]

**输出:** 6

**解释:** 该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6

**示例 3：**

**输入:** ["10", "6", "9", "3", "+", "-11", "\*", "/", "\*", "17", "+", "5", "+"]

**输出:** 22

**解释:**

该算式转化为常见的中缀算术表达式为：

((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5

= ((10 \* (6 / (12 \* -11))) + 17) + 5

= ((10 \* (6 / -132)) + 17) + 5

= ((10 \* 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

class Solution {

public:

    int evalRPN(vector<string>& tokens) {

        stack<int> stack;

        int num1, num2;

        for(auto sym : tokens){

            if(sym == "+"){

                num1 = stack.top();

                stack.pop();

                num2 = stack.top();

                stack.pop();

                stack.push(num2 + num1);

            }

            else if(sym == "-"){

                num1 = stack.top();

                stack.pop();

                num2 = stack.top();

                stack.pop();

                stack.push(num2 - num1);

            }

            else if(sym == "\*"){

                num1 = stack.top();

                stack.pop();

                num2 = stack.top();

                stack.pop();

                stack.push(num2 \* num1);

            }

            else if(sym == "/"){

                num1 = stack.top();

                stack.pop();

                num2 = stack.top();

                stack.pop();

                stack.push(num2 / num1);

            }

            else

                stack.push(atoi(sym.c\_str()));

        }

        return stack.top();

    }

};

贪心；

[122. 买卖股票的最佳时机 II](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-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）。

    int maxProfit(vector<int>& prices) {

        int max\_profit = 0;

        for(int i = 0; i < prices.size() - 1; i++)

        {

            if(prices[i + 1] > prices[i])

            // 只要上涨就进行买卖股票

                max\_profit += (prices[i + 1] - prices[i]);

        }

        return max\_profit;

    }

[452. 用最少数量的箭引爆气球](https://leetcode-cn.com/problems/minimum-number-of-arrows-to-burst-balloons/)

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足  xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

**Example:**

**输入:**

[[10,16], [2,8], [1,6], [7,12]]

**输出:**

2

**解释:**

对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（射爆另外两个气球）。

根据 x\_end 将气球进行排序。初始化 first\_end 为第一个气球结束的坐标 points[0][1]。

初始化箭的数量 arrows = 1。遍历所有的气球：如果气球的开始坐标大于 first\_end：

则增加箭的数量。将 first\_end 设置为当前气球的 x\_end。返回 arrows。

  int findMinArrowShots(vector<vector<int>>& points) {

    if (points.size() == 0) return 0;

    sort(begin(points), end(points),

         [](const vector<int> &o1, const vector<int> &o2) {

      return (o1[1] < o2[1]);

    });

    int arrows = 1;

    int xStart, xEnd, firstEnd = points[0][1];

    for (auto p : points) {

      xStart = p[0];

      xEnd = p[1];

      if (firstEnd < xStart) {

        arrows++;

        firstEnd = xEnd;

      }

    }

    return arrows;

  }

[455. 分发饼干](https://leetcode-cn.com/problems/assign-cookies/)

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子 i ，都有一个胃口值 gi ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 sj。如果 sj >= gi，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

**注意：**你可以假设胃口值为正。一个小朋友最多只能拥有一块饼干。

**示例 1:**

**输入:** [1,2,3], [1,1]

**输出:** 1

**解释:**

你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。

虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。

所以你应该输出1。

**示例 2:**

**输入:** [1,2], [1,2,3]

**输出:** 2

**解释:**

你有两个孩子和三块小饼干，2个孩子的胃口值分别是1,2。

你拥有的饼干数量和尺寸都足以让所有孩子满足。

所以你应该输出2.

int findContentChildren(vector<int>& g, vector<int>& s) {

    sort(g.begin(), g.end());sort(s.begin(), s.end());

    int i = 0, j = 0;int result = 0;

    while (i != g.size() && j != s.size())

    {

        if (g[i] <= s[j])

        {

            result++;i++;j++;

        }

        else

            j++;

    }

    return result;

}

[605. 种花问题](https://leetcode-cn.com/problems/can-place-flowers/)

假设你有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个数组包含0和1，其中0表示没种植花，1表示种植了花），和一个数 **n**。能否在不打破种植规则的情况下种入 **n**朵花？能则返回True，不能则返回False。

**示例 1:**

**输入:** flowerbed = [1,0,0,0,1], n = 1

**输出:** True

**示例 2:**

**输入:** flowerbed = [1,0,0,0,1], n = 2

**输出:** False

贪心算法一般用来解决需要 “找到要做某事的最小数量” 或 “找到在某些情况下适合的最大物品数量” 的问题，且提供的是无序的输入。

贪心算法的思想是每一步都选择最佳解决方案，最终获得全局最佳的解决方案。

我们从左到右扫描数组 flowerbed，如果数组中有一个 0，并且这个 0 的左右两侧都是 0，那么我们就可以在这个位置种花，即将这个位置的 0 修改成 1，并将计数器 count 增加 1。对于数组的第一个和最后一个位置，我们只需要考虑一侧是否为 0。

在扫描结束之后，我们将 count 与 n 进行比较。如果 count >= n，那么返回 True，否则返回 False

首先为了避免首位和尾位的特殊情况，可在首位和尾位都填上一个0，这样就可以一视同仁了，只需寻找到有多少个 0 的左右两边都是0即可，该个数即为可插花朵的最大值

class Solution {

public:

    bool canPlaceFlowers(vector<int>& flowerbed, int n) {

        int count = 0;

        flowerbed.insert(flowerbed.begin(),0);

        flowerbed.insert(flowerbed.end(),0);

        for(int i=1;i<flowerbed.size()-1;i++){

            if(flowerbed[i]==0&&flowerbed[i-1]==0&&flowerbed[i+1]==0){

                flowerbed[i]=1;

                count++;

            }

            if(count>=n)

                return true;

        }

        return n<=count;

    }

};

假设1元、2元、5元、10元、20元、50元、100元的纸币分别有a,b,c,d,e,f,g张。现在要用这些钱来支付m元，至少要用多少张纸币？用贪心算法的思想,每一次选择最大面值的钱币。

vector<int> Num{ 3,0,2,1,0,3,5 };

vector<int> Value{ 1,2,5,10,20,50,100 };

int BagsQues(int money) {

    int sum = 0;

    for (int i = Value.size() - 1; i >= 0; --i) {

        int N = min(money / Value[i], Num[i]);

        money = money - N \* Value[i];

        sum += N;

        if (money == 0)

            return sum;

    }

    return -1;

}

求出每张面额，用了多少张：

vector<int> Num{ 3,0,2,1,0,3,5 }, Value{ 1,2,5,10,20,50,100 };

vector<tuple<int, int> > BagsQues(int money) {

    int sum = 0;

    vector<tuple<int, int> > ch;

    for (int i = Value.size() - 1; i >= 0; --i) {

        int N = min(money / Value[i], Num[i]);

        money = money - N \* Value[i];

        sum += N;

        if (N != 0) {

            ch.push\_back({ Value[i], N });

        }

        if(money == 0)

            return ch;

    }

    ch.clear();

    ch.push\_back({ -1, -1 });

    return ch;

}

排序:

堆排序:

class Solution {

public:

    vector<int> getLeastNumbers(vector<int>& arr, int k) {

        int endIndex = arr.size() - 1;

        buildHeap(arr, endIndex);

        while(endIndex){

            swap(arr[0], arr[endIndex]);

            endIndex--;

            heap(arr, 0, endIndex);

        }

        vector<int> res(k, 0);

        for(int i = 0; i < k; i++){

            res[i] = arr[i];

        }

        return  res;

    }

    //初始化建堆的时间复杂度为O(n)

    void buildHeap(vector<int>& arr, int endIndex){

        for(int i = (endIndex - 1)/2; i >= 0; i--){

            heap(arr, i, endIndex);

        }

    }

    void heap(vector<int>& arr, int startIndex, int endIndex){

        while(true){

            int finalIndex = startIndex;

            if((startIndex \* 2 + 1) <= endIndex && arr[finalIndex] < arr[startIndex \* 2 + 1])

                finalIndex = startIndex \* 2 + 1;

            if((startIndex \* 2 + 2) <= endIndex && arr[finalIndex] < arr[startIndex \* 2 + 2])

                finalIndex = startIndex \* 2 + 2;

            if(finalIndex == startIndex)

                break;

            swap(arr[startIndex], arr[finalIndex]);

            startIndex = finalIndex;

        }

    }

};

// 重建堆一共需要n-1次循环，每次循环的比较次数为log(i)，则相加为：log2+log3+…+log(n-1)+log(n)≈log(n!)。可以证明log(n!)和nlog(n)是同阶函数：

// 初始化建堆的时间复杂度为O(n)，排序重建堆的时间复杂度为nlog(n)，所以总的时间复杂度为O(n+nlogn)=O(nlogn)。

快排:

int getStandard(int array[], int i, int j) {

    //基准数据

    int key = array[i];

    while (i < j) {

            //因为默认基准是从左边开始，所以从右边开始比较

            //当队尾的元素大于等于基准数据 时,就一直向前挪动 j 指针

            while (i < j && array[j] >= key) {

                j--;

            }

            //当找到比 array[i] 小的时，就把后面的值 array[j] 赋给它

            if (i < j) {

                array[i] = array[j];

            }

            //当队首元素小于等于基准数据 时,就一直向后挪动 i 指针

            while (i < j && array[i] <= key) {

                i++;

            }

            //当找到比 array[j] 大的时，就把前面的值 array[i] 赋给它

            if (i < j) {

                array[j] = array[i];

            }

        }

    //跳出循环时 i 和 j 相等,此时的 i 或 j 就是 key 的正确索引位置

    //把基准数据赋给正确位置

    array[i] = key;

    return i;

}

void QuickSort(int array[], int low, int high) {

    //开始默认基准为 low

    if (low < high) {

        //分段位置下标

        int standard = getStandard(array, low, high);

        //递归调用排序

        //左边排序

        QuickSort(array, low, standard - 1);

        //右边排序

        QuickSort(array, standard + 1, high);

    }

}

归并排序

void merge(vector<int>& arr, int lo, int mi, int hi){

    int\* temp = new int[hi - lo + 1];

    int p1 = lo, p2 = mi + 1, i = 0;

    while(p1 <= mi && p2 <= hi){

        temp[i++] = arr[p1] > arr[p2] ? arr[p2++] : arr[p1++];

    }

    while(p1 <= mi)

        temp[i++] = arr[p1++];

    while(p2 <= hi)

        temp[i++] = arr[p2++];

    for(int i = 0; i < hi - lo + 1; i++){

        arr[lo + i] = temp[i];

    }

    delete[] temp;

}

void mergeSort(vector<int>& arr, int lo, int hi){

    if(lo < hi){

        int mi = (lo + hi) >> 1;

        mergeSort(arr, lo, mi);

        mergeSort(arr, mi + 1, hi);

        merge(arr, lo, mi, hi);

    }

}

二分查找

int binary\_search(int arr[], int low, int high, int key)

{

    //查找某元素是否存在于数组中 若存在，则返回下标；否则返回-1

    while (low <= high) {

        // 检查key是否在中间

        int mid = low + (high - low) / 2;

        if (arr[mid] == key)

            return mid;

        // 如果x大于arr[mid] 则low = mid+1;否则high=mid-1

        if (arr[mid] < key)

            low = mid + 1;

        else

            high = mid - 1;

    }

    return -1;

}