

数据结构与算法分析

CS17-2Mw

数据结构与算法分析

C语言基础前置知识

ASCII码表常用符号

年份小知识

最大公约数和最小公倍数

文件的存储读写操作

格式化输出

数组以及链表

寻找两个有序数组的中位数

合并两个有序数组

有环链表

2019统考线性表题目

三数之和

最接近的三数之和

栈和队列

中缀表达式转后缀表达式

逆波兰表达式求值

柱状图中最大的矩形——单调栈

去除重复字母——单调栈

接雨水——单调栈，DP

树和森林

二叉树的创建

二叉树前中后序遍历

二叉树层序遍历

二叉树树高

二叉树最大树宽度

检验是否是完全二叉树

是否满二叉树

二叉树排序树BST

二叉平衡树BBT

B树

图

最小生成树

最短路径

拓扑排序

排序

插入排序

希尔排序

冒泡排序

快速排序

选择排序

堆排序

归并排序

基数排序

桶排序

递归与分治

全排列

斐波那契数列

整数划分

汉诺塔问题

二分查找

大整数乘法

strassen矩阵乘法

棋盘填充

快速排序

线性时间选择 (第K小的数)

最接近点对点 (未做)

循环赛日程表

合并k个排序链表

课后习题

分苹果问题

找零问题

动态规划DP

矩阵链乘法——区间DP

最长上升子序列

最长公共子序列

最大子段和

最大字段和 (升级版)

凸多边形最优三角剖分

流水调度作业

0-1背包问题

最优搜索二叉树

找零问题

数字组合问题

钢条切割

周年派对——树形DP

苹果树——树形背包问题

最小回文——区间DP

什么情况不能使用动态规划

课后习题

贪心算法

最优装载问题

活动安排问题

哈夫曼编码

单源最短路径

最小生成树

多机调度

课后习题

回溯法

Perm

最优装载(变形)

批处理作业调度

符号三角形

N皇后问题

最大团问题

旅行售货商问题

电路板排列问题

连续邮资问题

课后习题

分支限界法

单源最短路径

最优装载

旅行售货商问题

布线问题

最大团问题

批作业处理调度

串

朴素模式匹配

KMP算法

背包问题专题

01背包问题

完全背包问题

多重背包问题

混合背包问题

二维费用背包问题

分组背包问题

真题

2019-3 字典序

Tips

做题要诀

审清题目，代入数据，明白流程

理清思路，确定方法，特殊情况

记录要点，注意细节，编写代码

审查代码，检查要点，加入注释

C语言基础前置知识

ASCII码表常用符号

48-57 数字字符0-9

65-90 大写字母A-Z

97-122 小写字母 a-z

小写字母变大写 $x - 32 = X$

年份小知识

闰年

是4的倍数但不是100的倍数是闰年，对于是400的倍数且100的倍数为闰年。1900不是闰年，2000年是闰年。

```
bool isLeap(int n) {
    if((n%4==0&&n%100!=0) || n%400==0) return true;
    else return false;
}
```

第几年的第几天

耶稣诞生那天是星期一

最大公约数和最小公倍数

最大公约数(greatest common division)和最小公倍数(Least common multiple)的关系：

$$\text{lcm}(a, b) = \frac{a \cdot b}{\text{gcd}(a, b)}$$

最大公约数：

```
int gcd(int a, int b) {
    if(b) while((a%b)&&(b%a));
    return a+b;
}
```

最小公倍数：

```
int lcm(int a, int b) {
    return a*b/gcd(a, b);
}
```

文件的存储读写操作

```
#include<stdio.h>
int main() {
    FILE* f; //FILE 文件类型，大写
    f = fopen("a.txt", "w+"); //mode见以下介绍
    char s[] = "good night", s2[100];
    fwrite(s, sizeof(char), strlen(s), f); //写s字符串 或者是fputs(s, f);
    fseek(pFile, 0, SEEK_END); //指向文件末尾
    long lSize = ftell(f); //获取文件长度
    rewind(f); //将指针重新指向文件开头
    fread(s2, sizeof(char), lSize, f); //或者是规则的字符串使用fscanf(f, "%s", s2)
    fclose(f); //必须关闭文件，保证输出流成功写到文件中。或者使用fflush(f)强制写入输出流。
}
```

读取和写出字符使用 fgetc(char,FILE*) 和 fputc(char,FILE*)

读取和写出字符串使用 fgets(char*,FILE*) 和 fputs(char*,FILE*)

fopen() 函数 mode 解释：

- r 打开只读文件，该文件必须存在。
- r+ 打开可读写的文件，该文件必须存在，从文件头开始写，保留原文件中没有被覆盖的内容。
- w 打开只写文件，若文件存在则清楚文本内容。若文件不存在则建立该文件。
- w+ 打开可读写文件，若文件存在则清楚文本内容。若文件不存在则建立该文件。
- a 和 a+ 用于追加写文件，指针指向文本末尾。

格式化输出

flags (标识)	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
空格	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width (宽度)	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision	描述
.number	<p>对于整数说明符 (d、i、o、u、x、X)：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。</p> <p>对于 e、E 和 f 说明符：要在小数点后输出的小数位数。</p> <p>对于 g 和 G 说明符：要输出的最大有效位数。</p> <p>对于 s：要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。</p> <p>对于 c 类型：没有任何影响。</p> <p>当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。</p>
*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

数组以及链表

寻找两个有序数组的中位数

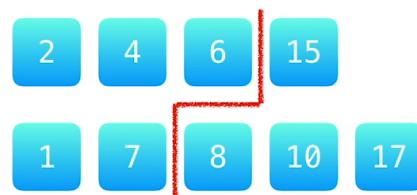
二分法 难度：☆☆☆☆☆

<https://leetcode-cn.com/problems/median-of-two-sorted-arrays>

我遇到的问题：

- 如何处理两个有序数组到最后剩余 $[a, b]$ 和 $[c]$ 三个左右的元素？
- 两个数组长度和在分为奇数和偶数的情况如何处理？
- 最后的处理又十分繁琐，如何省去这些繁琐的细节处理操作？

思想：



画一条【分割线】，这一个【分割线】使得两边的元素数量大致相同，并且满足【交叉小于等于】的关系：红线左边所有元素都要小于右边的元素。

Tips:

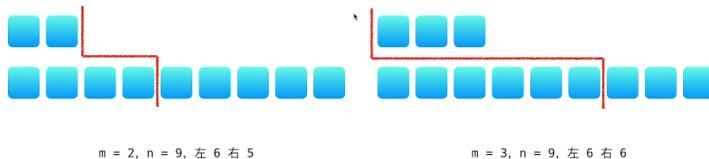
- 为了防止整形溢出一般使用 $mid = m + (n - m + 1)/2$
- 整数除法默认下取整，因此在取中位数的时候使用 $mid = (n + m + 1)/2$ 代替 $mid = (n + m)/2$ 。因为只剩下两个元素的时候，一直取中位数，由于默认向下取整，因此会一直选择第一个元素，从而陷入了死循环。

操作：

- 当 $m + n$ 为偶数的时候：中间位置为： $\frac{m+n}{2} = \frac{m+n+1}{2}$
- 当 $m + n$ 为奇数的时候：左边分得的数目多一个： $\frac{m+n+1}{2}$

可能遇到的极端情况：





```

class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        const int MIN = 0x80000000;
        const int MAX = 0x7fffffff;
        if(nums1.size() > nums2.size()){
            vector<int> tmp = nums1;
            nums1 = nums2;
            nums2 = tmp;
        }
        int m = nums1.size(), n = nums2.size();
        int totalLeft = m + (n - m + 1) / 2; //奇数情况 左边分得的数目要多一个
        int left = 0, right = m;
        // 必须保证nums1[i-1] <= nums2[j] && nums2[j-1] <= nums1[i]
        while(left < right){
            int i = left + (right - left + 1) / 2; // 加一就意味着nums1数组分割线左边始终有元素
            int j = totalLeft - i;
            if(nums1[i-1] > nums2[j]){
                right = i - 1; // nums1的分割先过右了, 下个搜索区间[left, i-1]
            } else{
                left = i; //下个搜索区间[i, right] 需要注意此处的死循环
            }
        }

        int i = left, j = totalLeft - i;
        int nums1LeftMax = i == 0 ? MIN : nums1[i-1];
        int nums2LeftMax = j == 0 ? MIN : nums2[j-1];
        int nums1RightMin = i == m ? MAX : nums1[i];
        int nums2RightMin = j == n ? MAX : nums2[j];
        if((m+n)%2 == 1){
            return max(nums1LeftMax, nums2LeftMax);
        } else{
            return 1.0 * (max(nums1LeftMax, nums2LeftMax) + min(nums1RightMin, nums2RightMin)) / 2;
        }
    }
};

```

合并两个有序数组

难度: ☆

【题目描述】

给你两个有序整数数组 `nums1` 和 `nums2`，请你将 `nums2` 合并到 `nums1` 中，使 `nums1` 成为一个有序数组。

初始化 `nums1` 和 `nums2` 的元素数量分别为 `m` 和 `n`。

你可以假设 `nums1` 有足够的空间（空间大小大于或等于 `m + n`）来保存 `nums2` 中的元素。

【输入】

```

nums1 = [1, 2, 3, 0, 0, 0], m = 3
nums2 = [2, 5, 6], n = 3

```

【输出】

```
[1, 2, 2, 3, 5, 6]
```

【思想】

很容易想出排序算法 $T = O\{(m+n)\log(m+n)\}$, 以及传统归并 $T = O(m+n)$, $S = O(m+n)$

由于 `nums1` 的数组后面有许多未使用的空间, 因此没必要从“头”开始归并, 换个思路可以从“尾”开始归并也是很好的解决方法。

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int s = m-- + n-- - 1;
    while (m >= 0 && n >= 0) {
        if (nums1[m] > nums2[n]) nums1[s--] = nums1[m--];
        else nums1[s--] = nums2[n--];
    }
    while (n >= 0) nums1[s--] = nums2[n--];
}
```

有环链表

难度: ☆☆

如何判断链表有环:

龟兔赛跑算法 (快慢指针) :

刚开始指针A和指针B指向链表头, A每向前走一步, B就向前走两步。

如果无环, B会首先变成null空指针, B指针不可能追上A。

一旦两者进入同一个环中, 无论两个点在哪个位置起步到最后都一定会相遇。

当B距离A有1个距离时, 下一次迭代两者一定会相遇。

当B距离A有2个距离的时候, 经过一次迭代就会变成第一种情况, 因此一定会相遇。

如何确定有环链表的起点:

41. (13分) 设线性表 $L = (\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_{n-2}, \alpha_{n-1}, \alpha_n)$ 采用带头结点的单链表保存, 链表中的

结点定义如下:

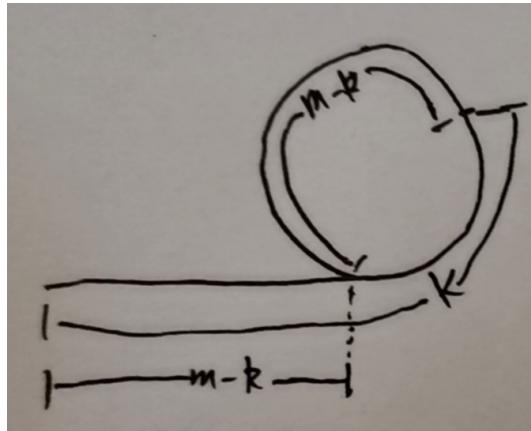
```
typedef struct node
{
    int data;
    struct node*next;
} NODE;
```

请设计一个空间复杂度为 $\mathcal{O}(1)$ 且时间上尽可能高效的算法, 重新排列 L 中的各结点, 得到线性表 $L' = (\alpha_1, \alpha_n, \alpha_2, \alpha_{n-1}, \alpha_3, \alpha_{n-2}, \dots)$ 。要求:

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想, 采用 C 或 C++语言描述算法, 关键之处给出注释。
- (3) 说明你所设计的算法的时间复杂度。

由上图可知, 当指针A和指针B第一次相遇, A再走过环的长度n时, B会和A再次相遇。因为当A走出n步时, B走了 $2n$ 步, 两者又回到了相同点, 第二次相遇的地点一定与上一次的地点相同。

记录A走过的距离 K , 即为环的长度 K 。

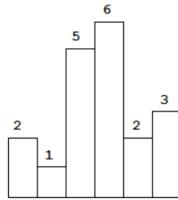


紧接这根据快慢指针的思想: 由于环的长度为 K , 因此让A和B同时回到链表起点, 先让B走 K 步; 然后A和B同时出发, 当到达环头的时候, B指针恰好与A指针重合, 即可得到环头。

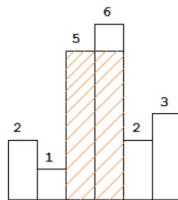
2019统考线性表题目

难度：☆☆☆

求在该柱状图中，能够勾勒出来的矩形的最大面积。



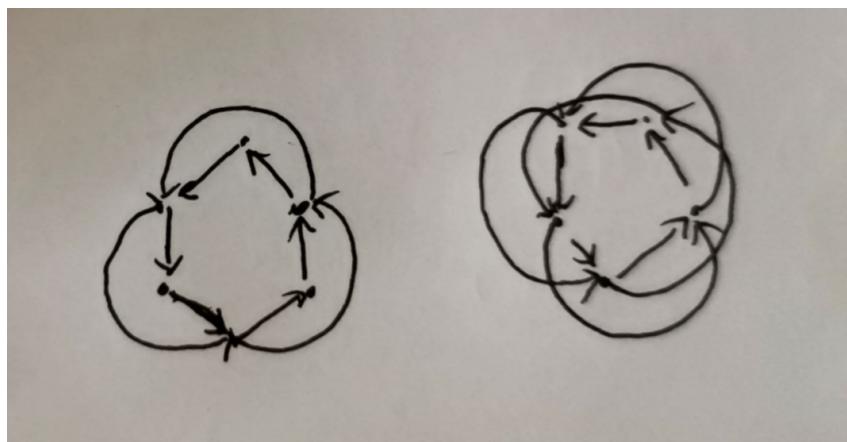
以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2, 1, 5, 6, 2, 3]。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

关键词：双指针，单链表逆置

思想：设 $b_n = \{a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}, \dots\}$ ，其中 b_{2n+1} 部分是 $\{a_1, a_2, a_3, \dots\}$ ，属于 $\{a_n\}$ 的前半部分， b_{2n} 部分是 $\{a_n, a_{n-1}, a_{n-2}\}$ 属于 $\{a_n\}$ 的后半部分， b_n 数列相当于是 $\{a_n\}$ 数列的前半部分和后半部分的合并。因此使用快慢指针，mid 指针每走一步，tail 指针向前走两步，当 tail 指向链表末尾的时候，mid 找到 a_n 的中间位置，tail 指向链表的末尾位置，将 a_n 的后半部分进行逆置。逆置之后，head 和 tail 分别指向两个链表的头部，两个链表就可以进行穿插合并了。



```
void change(node *head) {
    node* tail = head->next, *mid = head->next;
    while(tail->next != NULL) {
        mid = mid->next;
        tail=tail->next->next; //快慢指针，当tail指针到达尾部的时候，mid指向中间部分
    }
    node *tmp = mid, *mid_next;
    mid = mid->next; tmp->next=NULL;
    while(mid!=NULL){ //后半部分逆置
        mid_next = mid->next;
        mid->next = tmp;
        tmp = mid;
        mid = mid_next;
    }
    node* a = head->next, *b = tail;
    while(a->next!=NULL&&b->next!=NULL){ //开始依次连接
        node *tmp;
        if(b){
            tmp = a->next;
            a->next=b;
            b->next=tmp;
        }
    }
}
```

```

        a->next = b;
        a = tmp;
    }
    if(a) {
        tmp = b->next;
        b->next = a;
        b = tmp;
    }
}

```

三数之和

难度: ☆☆☆

三数之和 <https://leetcode-cn.com/problems/3sum/>

【题目描述】

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]
]
```

方法1：

【思想】：为了不重复需要满足 $a+b+c=0$ ，且 $a < b < c$ 。先排序， $O(n^2 \log_2 n)$ 找两个数并且去重，用二分查找法找 c 。需要证明到 a 一定小于0，当 a 确定之后，由于 $b < c$ ，因此当 $b > (-a)/2$ 时就可以停止搜索了。

```

bool bin_search(int i, vector<int> nums, int target) {
    //二分查找。在leetcode中二分查找的函数调用时间耗费较多，还不如O(n)的暴力搜索
    int a = i, b = nums.size() - 1;
    while(a <= b) {
        int m = (a+b)/2;
        if(nums[m]==target) return true;
        else if(nums[m]<target) a = m + 1;
        else b = m - 1;
    }
    return false;
}

vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> ret;
    int n = nums.size();
    sort(nums.begin(), nums.end());
    if(n<3) return ret;
    int pa = nums[0] + 1;
    for(int i=0;i<n-2;i++) {
        if(nums[i]==pa) continue; //与第一个数字重复，跳过
        else pa = nums[i];
        int pb = nums[i+1] + 1;
        for(int j=i+1;j<n-1;j++) {
            if(nums[i]+nums[j]+nums[j+1]>0) break;
            //剪枝 如果a+b+c已经>0, 那后面的就不用搜索了
            if(nums[j]==pb) continue; //与第二个数字重复，跳过
            int pc = -pa - pb;
            if(pc<=j) break;
            if(pc==j) {
                vector<int> v{pa, pb, pc};
                sort(v.begin(), v.end());
                if(v!=ret.back()) ret.push_back(v);
            }
        }
    }
}

```

```

        else pb = nums[j];
        int tar = -nums[i]-nums[j]; //确定对象
        if(bin_search(j+1, nums, tar))ret.push_back({nums[i], nums[j], tar}); //如果存在则加入
    }
}
return ret;
}

```

时间复杂度分析：两层for循环，外加一个二分查找。 $T(n) = n * n * \log_2 n = O(n^2 \log n)$

方法2（推荐）：双指针法：

【思路】：使用双指针来进行查找，由于这个是三个数之和，如果不处理那就是“三指针”，因此我们首先应该对数组进行排序处理，处理之后每次确定一个最小的值，剩下两个值就交给双指针进行查找即可。

```

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

vector<vector<int>> threeSum(vector<int>& nums) {
    int n = nums.size();
    if(n<3) return {};
    sort(nums.begin(), nums.end());
    for(int i=0;i<n-2;i++) {
        if(i!=0&&nums[i]==nums[i-1])continue; //遇到重复的值跳过
        int low = i + 1, high = n - 1;
        while(low<high) {
            if(low!=i+1&&nums[low]==nums[low-1]) {
                low++; //遇到第二个数相同跳过
                continue;
            }
            int cnt = nums[i] + nums[low] + nums[high];
            if(cnt==0)ret.push_back({nums[i], nums[low], nums[high]});
            if(cnt>0)high--; //双指针的精髓
            else low++;
        }
    }
    return ret;
}

```

最接近的三数之和

<https://leetcode-cn.com/problems/3sum-closest/>

给定一个包括 n 个整数的数组 nums 和一个目标值 target 。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

【示例】：

输入： $\text{nums} = [-1,2,1,-4]$, $\text{target} = 1$

输出：2

解释：与 target 最接近的和是 2 ($-1 + 2 + 1 = 2$)。

【提示】：

```

3 <= nums.length <= 10^3
-10^3 <= nums[i] <= 10^3
-10^4 <= target <= 10^4

```

【思想】：同样是双指针，排序后确定一个最小的数值，后面两个大数进行双指针的“靠近”。

```

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

```

```

int threeSumClosest(vector<int>& nums, int target) {
    int n = nums.size();
    sort(nums.begin(), nums.end());
    int ret = nums[0] + nums[1] + nums[2];
    for(int i = 0; i < n - 2; i++) {
        int low = i + 1, high = n - 1;
        while(low < high) {
            int cnt = nums[i] + nums[low] + nums[high];
            ret = abs(cnt - target) < abs(ret - target) ? cnt : ret;
            if(cnt > target) high--;
            else if(cnt < target) low++;
            else return target;
        }
    }
    return ret;
}

```

栈和队列

关键词：“保持相对顺序”

中缀表达式转后缀表达式

和编译原理挂钩

逆波兰表达式求值

难度：☆

```

int evalRPN(vector<string>& tokens) {
    stack<int> digit;
    for(int i=0; i<tokens.size(); i++) {
        string s = tokens[i];
        if(isdigit(s[0]) || s.length() > 1) digit.push(stoi(tokens[i]));
        else {
            int b = digit.top(); digit.pop();
            int a = digit.top(); digit.pop();
            if(s[0] == '+') digit.push(a+b);
            else if(s[0] == '-') digit.push(a-b);
            else if(s[0] == '*') digit.push(a*b);
            else if(s[0] == '/') digit.push(a/b);
        }
    }
    return digit.top();
}

```

柱状图中最大的矩形——单调栈

单调栈、哨兵 难度：☆☆☆☆

<https://leetcode-cn.com/problems/largest-rectangle-in-histogram/>

给你一个仅包含小写字母的字符串，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

示例 1：

输入： "bcabc"
输出： "abc"

示例 2：

输入： "cbacdcbc"
输出： "acdb"

单调栈：单调递增的栈可以找到第一个比当前出栈元素小的元素

思想：如果后一个柱B的高度比前一个A高度短，则以A为高的最大面积矩形就已经确定，其面积=A*(当前下标-栈顶)，以此循环直到栈中没有比B还高的柱形即可。

```
int largestRectangleArea(vector<int>& heights) {
    int n = heights.size();
    if(n==0) return 0;
    if(n==1) return heights[0];
    vector<int> newHeight = heights;
    n+=2;
    newHeight.emplace(newHeight.begin(),0); //设置前面哨兵
    newHeight.push_back(0); //设置后哨兵
    stack<int> s;
    s.push(0);
    int ans = 0;
    for(int i=1;i<n;i++) {
        while(newHeight[i]<newHeight[s.top()]) { //设置单调栈
            int x = s.top();
            s.pop();
            int weight = i - s.top() - 1;
            ans=max(ans,weight*newHeight[x]);
        }
        s.push(i);
    }
    return ans;
}
```

去除重复字母——单调栈

字典序，单调栈 难度：☆☆☆☆

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。感谢 Marcos 贡献此图。

示例：

输入： [0,1,0,2,1,0,1,3,2,1,2,1]
输出： 6

题解：去除重复的字母，就表示每个字母至少出现一次。如果使用单调栈的话，中间还应该穿插对于不同字母数量的比对，如果字母的数量只有一个，那就在递增单调栈保留该元素；如果剩余数量大于一个，那就递增单调栈覆盖掉前一个。如果字母已经出现在单调栈中，那就忽略此字母并且让数量减一，因为排在前面相同的元素有更低的字典序。

代码：

```
bool used[27];
int cnt[27];
string removeDuplicateLetters(string s) {
    int n = s.length();
    for(char c:s){++cnt[c-'a'];
```

```

stack<char> m;
for(int i=0;i<n;i++) {
    while(!m.empty()&&s[i]<m.top()&&cnt[m.top()-'a']>1&&used[s[i]-'a']==0) {
        //栈非空, 逆序, 存在数量大于1, 并且未使用过
        used[m.top()-'a']=0;
        cnt[m.top()-'a']--;
        m.pop();
    }
    if(used[s[i]-'a']==0) {
        m.push(s[i]);
        used[s[i]-'a']=1;
    } else cnt[s[i]-'a']--;
}
string ans="";
while(!m.empty()) {
    ans.push_back(m.top()); //弹栈
    m.pop();
}
reverse(ans.begin(),ans.end()); //逆序
return ans;
}

```

接雨水——单调栈，DP

<https://leetcode-cn.com/problems/trapping-rain-water/>

难度: ★★★★☆

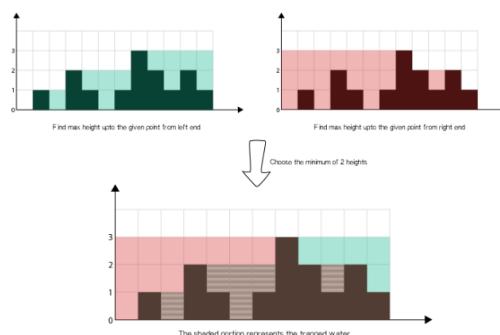
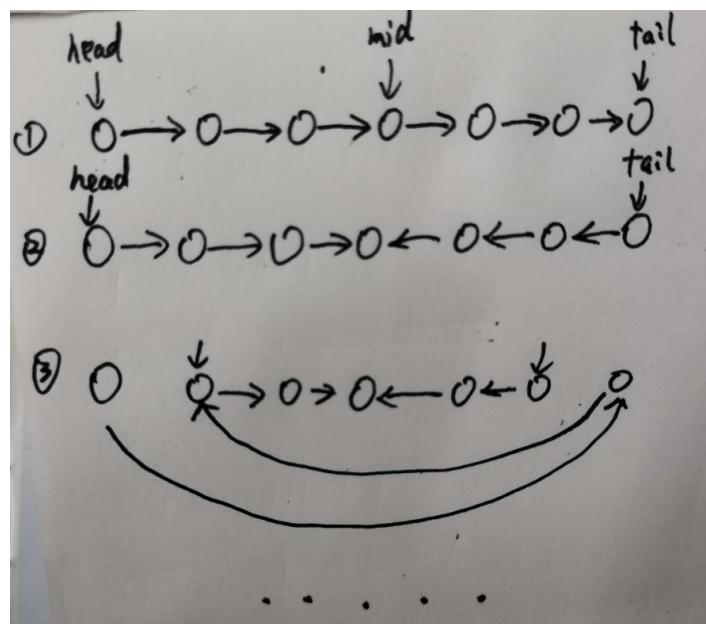
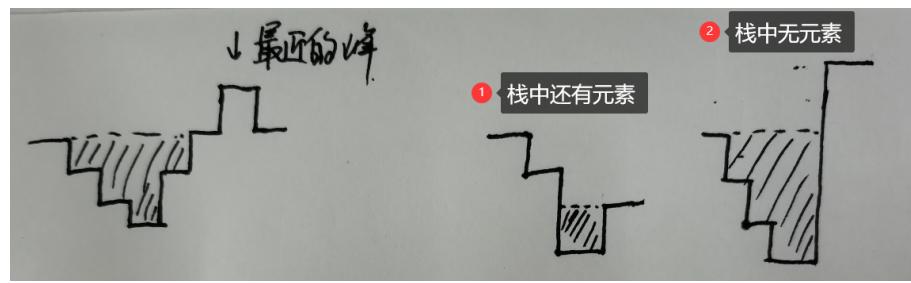


Fig: Dynamic Programming Approach

单调栈法：从第一个不是0的柱子开始构建一个单调不增栈，如果遇到一个大于栈顶的元素，则继续遍历数组找到右边最近的最凸的“峰”（元素），找到最近的凸峰之后就可以计算一个凹槽的雨水体积。还需要考虑多种情况，由于相当于一个“桶”，“桶”的容积由最低的“板”决定，因此需要考虑多种情况。



当遇到情况①的时候高度应为当前遍历元素的高度。当遇到②情况的时候高度为左边的最相应的元素。由于栈的特殊性，如果计算后不会保留以前计算的结果，因此会遇到一个难题：

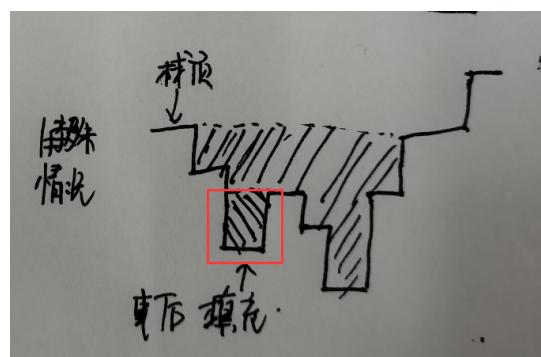


在途中划圈的部分会在后面重复计算，会导致结果有误。因此在每次计算完成之后将“坑”进行填充，就可以避免以后的重复计算了。时间复杂度O(n)，空间复杂度O(n)

代码如下：

```
class Solution {
public:
    int trap(vector<int>& height) {
        stack<int> s;
        int i = 0, n = height.size();
        if(n<=1) return 0;
        while(height[i]==0) i++;
        int ans=0;
        for(;i<n;i++) {
            int start=-1, h=0;
            while(!s.empty()&&height[i]>height[s.top()]) {
                while(i+1<n&&height[i+1]>height[i]) i++; //一直找到右侧最近的高峰
                start = s.top();
                h = max(h, height[start]); //记录当前遍历元素左边的最高高度
                s.pop();
            }
            if(start!=-1) {
                if(!s.empty()) {h=height[i];start=s.top();} //如果栈中还存在元素，表明属于①情况
                for(int j=start;j<i;j++) {
                    if(h-height[j]>0) {
                        ans+=h-height[j]; //加和
                        height[j]=h; //填充
                    }
                }
            }
            s.push(i);
        }
        return ans;
    }
};
```

DP法：



```
class Solution {
public:
    int trap(vector<int>& height) {
```

```

int n = height.size();
// left[i]表示i左边的最大值，right[i]表示i右边的最大值
vector<int> left(n), right(n);
for (int i = 1; i < n; i++)
    left[i] = max(left[i - 1], height[i - 1]);
for (int i = n - 2; i >= 0; i--)
    right[i] = max(right[i + 1], height[i + 1]);
int water = 0;
for (int i = 0; i < n; i++) {
    int level = min(left[i], right[i]);
    water += max(0, level - height[i]);
}
return water;
}
};


```

十分简洁明了，DP牛逼！

树和森林

概念：树的高度，树的深度，树的度=结点的最大度，分支结点，祖父，双亲，子孙

二叉树定义：

C语言定义方式：

```

typedef struct Node{
    int c;
    struct tree *left, *right;
}biNode,*biTree;

```

C++定义方式：

```

struct Node{
    int c;
    tree *left = NULL,*right = NULL;
};

```

二叉树的创建

根据前序遍历顺序和中序遍历顺序创建：

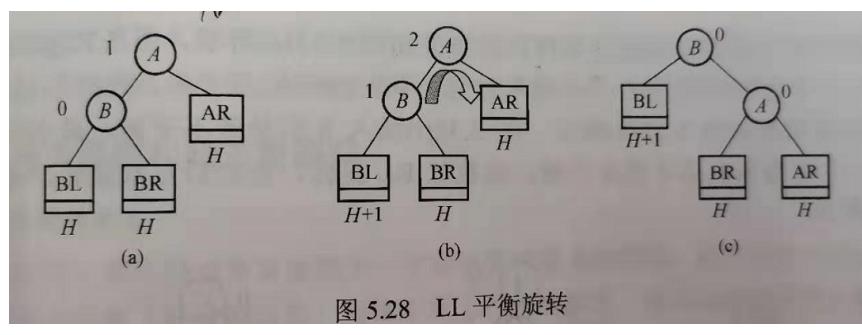


图 5.28 LL 平衡旋转

根据前序遍历和中序遍历的性质可以知道，前序遍历的第一个字母就是根节点，在对应的中序遍历中，根节点左边的序列就是其左子树的结点，右边的序列就是右子树的结点。因此在中序遍历的序列中根节点左侧序列“DBE”的数目就是左子树结点的数目 m ，在前序序列中根节点A后面的 m 位字母即为左子树结点的前序遍历序列“BDE”。其他的即为右子树的序列。可以使用递归的方式进行创造子树。

```

Node *createTree(string preSeq, string midSeq) {
    if (!preSeq.length()) return null;
    Node *node = new Node;
    node->c = preSeq[0];
    int index = midSeq.find(preSeq[0]); //找到根节点在中序遍历中的位置
    node->left = createTree(preSeq.substr(1, index), midSeq.substr(0, index));
    node->right = createTree(preSeq.substr(index + 1), midSeq.substr(index + 1));
    return node;
}

```

二叉树前中后序遍历

三种遍历方式，前序遍历，中序遍历和后序遍历

递归形式：思想——栈

```

void preTraverse(Node *t) {
    if (t == null) return;
    cout << t->c;
    preTraverse(t->left);
    preTraverse(t->right);
}

void midTraverse(Node *t) {
    if (t == null) return;
    midTraverse(t->left);
    cout << t->c;
    midTraverse(t->right);
}

void sufTraverse(Node *t) {
    if (t == null) return;
    sufTraverse(t->left);
    sufTraverse(t->right);
    cout << t->c;
}

```

循环形式：

```

//先序遍历（循环）
/*
先序遍历的输出顺序是：父节点，左孩子，右孩子，使用栈的形式进行实现因此入栈的顺序是右孩子->左孩子->父节点
*/
void preTraverseL(Node *t) {
    stack<Node *> s;
    s.push(t);
    while (!s.empty()) {
        Node *tmp = s.top();
        s.pop();
        if (tmp->right) s.push(tmp->right); //右孩子入栈
        if (tmp->left) s.push(tmp->left); //左孩子入栈
        cout << tmp->c;
    }
}

//中序遍历（循环）
/*
中序遍历的输出顺序是：左孩子->父节点->右孩子，使用栈的形式进行实现因此入栈的顺序是右孩子->父节点->左孩子。由于在入栈的过程中，父节点是在右孩子之前入栈，因此需要将父节点的子节点们“取下来”，否则会导致重复遍历。
*/
void midTraverseL(Node *t) {
    stack<Node *> s;
    s.push(t);
    while (!s.empty()) {

```

```

        Node *tmp = s. top();
        s. pop();
        if (tmp->right)s. push(tmp->right); //右孩子入栈
        if (tmp->left || tmp->right) { //父节点存在子节点就“取下来”入栈
            Node *t = new Node;
            t->c = tmp->c;
            s. push(t);
        } else cout << tmp->c; //叶子结点直接输出。
        if (tmp->left)s. push(tmp->left); //左孩子入栈
    }
}

/*
后序遍历的输出顺序是：左孩子，右孩子，父节点，使用栈的形式进行实现因此入栈的顺序是父节点->右孩子->左孩子
*/
void sufTraverseL(Node *t) {
    stack<Node *> s;
    s. push(t);
    while (!s. empty()) {
        Node *tmp = s. top();
        s. pop();
        if (tmp->left || tmp->right) { //“取下来”
            Node *t = new Node; //父节点入栈
            t->c = tmp->c;
            s. push(t);
        } else cout << tmp->c; //叶子结点直接输出
        if (tmp->right)s. push(tmp->right); //右孩子入栈
        if (tmp->left)s. push(tmp->left); //左孩子入栈
    }
}

```

二叉树层序遍历

```

//层序遍历，每一层从左向右遍历
void levelTraverse(Node *t) {
    if (t == null) return;
    queue<Node*> q;
    q. push(t);
    while (!q. empty()) {
        Node *tmp = q. front();
        q. pop();
        cout << tmp->c;
        if (tmp->left != null)q. push(tmp->left);
        if (tmp->right != null)q. push(tmp->right);
    }
}

```

二叉树树高

```

int getTreeHeight(Node*t) {
    if(t==NULL) return 0;
    return max(getTreeHeight(t->left), getTreeHeight(t->right)) + 1;
}

```

二叉树最大树宽度

```

//简单易懂，不解释
int getHeight(TreeNode* root){
    if(root==NULL) return 0;
    return max(getHeight(root->left),getHeight(root->right)) + 1;
}

void dfs(TreeNode* root, vector<vector<int>> &grp, int o){
    if(root==NULL) return;

```

```

int level = (int)log2(o);
grp[level][0] = grp[level][0]==0?o:min(grp[level][0],o);
grp[level][1] = max(grp[level][1],o);
dfs(root->left, grp, o*2);
dfs(root->right, grp, o*2+1);
}

int widthOfBinaryTree(TreeNode* root) {
    int n = getHeight(root);
    vector<vector<int>> grp(n, vector<int>(2, 0));
    dfs(root, grp, 1);
    int _m = 0;
    for(int i=0; i<n; i++) _m = max(_m, grp[i][1]-grp[i][0]);
    return _m+1;
}

```

检验是否是完全二叉树

```

void f(Node*t, int index, bool *s, int &m) {
    if(t==NULL) return;
    m = max(m, index);
    s[index]=true;
    f(t->left, index*2, s, m);
    f(t->right, index*2+1, s, m);
}

bool isCompleteTree(Node *t) {
    int n = pow(2, getTreeHeight(t)) + 1;
    bool *s = new bool[n];
    memset(s, 0, sizeof(bool));
    int m = 0;
    f(t, 1, s, m);
    for(int i=1; i<=m; i++)
        if(!s[i]) return false;
    return true;
}

```

是否满二叉树

```

bool isFullTree(Node *t) {
    if(t->left==NULL&&t->right==NULL) return true;
    else if(t->left!=NULL&&t->right!=NULL) return isFullTree(t->left)&&isFullTree(t->right);
    else return false;
}

```

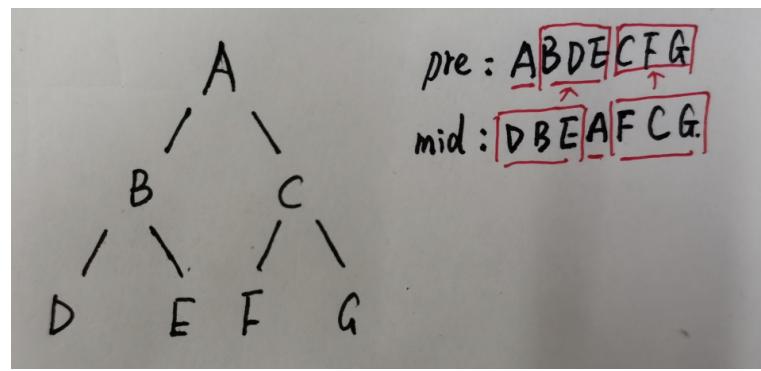
二叉树排序树BST

二叉排序（搜索）树binary search tree：根节点的左子树上的结点都要比根节点小，右子树都比根节点大。

同二叉平衡树的区别：只负责寻找元素，不负责平衡因子。

特征：BST的前序遍历是递增的序列。

如何删除结点：



上图还缺少的一种情况就是删除叶节点的情况，因此如果结点不是根节点，则需要另外一个变量来记录所删除结点的父节点情况。

二叉平衡树BBT

binary balance tree (AVL). 左子树和右子树之差叫做平衡因子，只能取-1, 0, 1。平衡树的左右子树高度差不能超过1。

与BST的关系：是一种特殊的BST

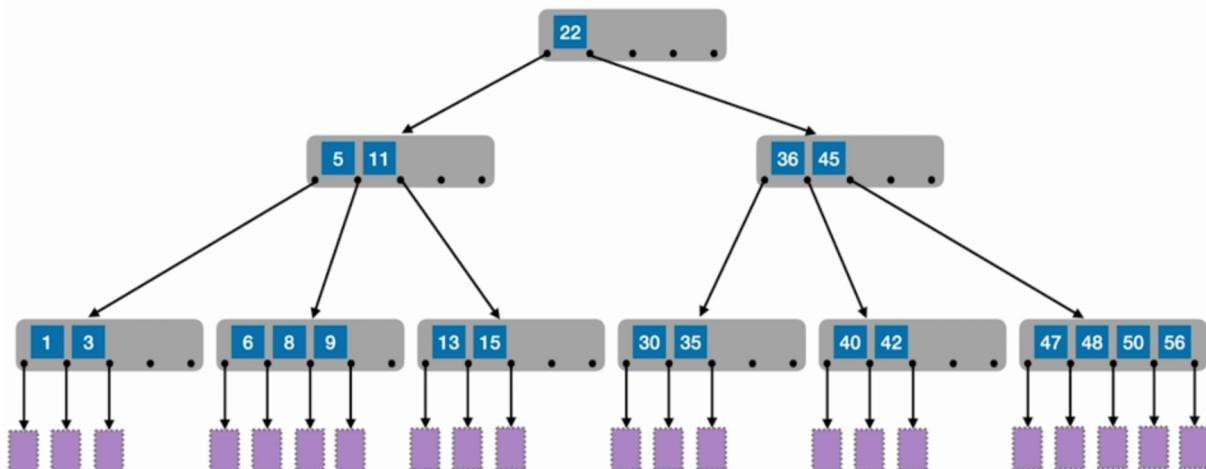
二叉树的结构定义

```
struct BBT{
    int data;
    int balance_factor;
    BBT *left=NULL, *right=NULL;
};
```

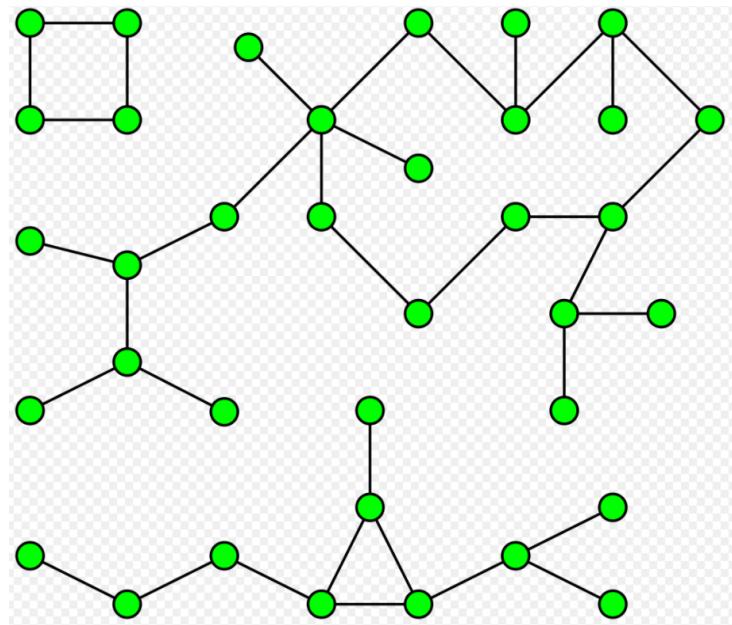
二叉树的调整：找到最小不平衡子树，四种调整方法 (LL, LR, RL, RR)

LL：在A的左孩子的左子树插入的结点导致的不平衡，其他类似。

LL型的旋转方式：



LR的旋转方式：



RR和RL的旋转方式跟LL和RL的旋转方式类似。

```

int getBalanceFactor(BBT *t) {
    return getHeight(t->left) - getHeight(t->right); //左子树高度减右子树高度
}

void updateBF(BBT *t) {
    if(!t) return;
    t->bf = getBalanceFactor(t); //更新平衡因子
    updateBF(t->left);
    updateBF(t->right);
}

void insertBBT(BBT* &t, int n) {
    if(!t) return;
    BBT* tmp = t, *pre = t, *target = new BBT;
    stack<BBT*> gfStack; //祖父栈, 用于找到最小不平衡子树
    // 将数值为n的结点插入子树
    target->data = n;
    target->bf = 0;
    while(tmp) {
        pre = tmp; //记录父节点
        gfStack.push(tmp);
        if(n>tmp->data) tmp = tmp->right;
        else tmp = tmp->left;
    }
    if(n>pre->data) pre->right=target;
    else pre->left = target;
    // 更新平衡因子 找到最小不平衡树
    while(!gfStack.empty()) {
        tmp = gfStack.top();
        updateBF(tmp); //更新平衡因子
        if(abs(tmp->bf)>1) {
            gfStack.pop();
            if(!gfStack.empty()) pre = gfStack.top(); //记录父节点
            else pre = NULL;
            break;
        }
        gfStack.pop();
    }
    BBT* b,*c;
    if(tmp->bf == 2) {
        b = tmp->left;
        if(b->bf == 1) {
            //LL型
        }
    }
}

```

```

        tmp->left = b->right;
        b->right = tmp;
        if(pre) {
            if(pre->left==tmp) pre->left=b;
            else pre->right = b;
        }else t=b;
    }else{
        //LR 型
        c = b->right;
        b->right = c->left;
        tmp->left = c->right;
        c->left = b;
        c->right = tmp;
        if(pre) {
            if(pre->left==tmp) pre->left=c;
            else pre->right=c;
        }else t = c;
    }
}
else if(tmp->bf == -2){
    b= tmp->right;
    if(b->bf == -1){
        //RR
        tmp->right = b->left;
        b->left = tmp;
        if(pre) {
            if(pre->left==tmp) pre->left=b;
            else pre->right=b;
        }else t= b;
    }else{
        //RL
        c = b->left;
        b->left = c->right;
        tmp->right = c->left;
        c->right = b;
        c->left = tmp;
        if(pre) {
            if(pre->left==tmp) pre->left=c;
            else pre->right=c;
        }else t=c;
    }
}
updateBF(t);
}

```

B树

又称多路平衡查找树，是一般化的二叉搜索树。这种数据结构常被应用在数据库和文件系统的实现上。这部分851考察的可能性较低，基础了解B树的结构和性质即可。

结构定义：

```

struct Node{      //5叉排序树
    ELEMType keys[4];   //最多4个关键字，递增或者递减
    struct Node* child[5]; //最多5个孩子
    int num;    //当前关键字的个数
}

```

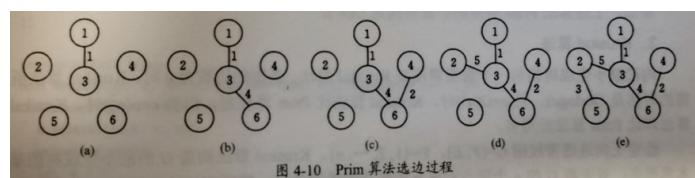


图 4-10 Prim 算法选边过程

B树的要求：

- 多路：为了保证查找的效率， m 叉查找树中除了根节点至少要有 $\lceil m/2 \rceil$ 个分叉。
- 平衡：所有的节点都要绝对平衡。叶节点（失败节点）都出现在同一层。

图

图的定义较为繁杂

图的定义： $G = (V, E)$ ，图的阶即顶点的个数 $|V|$ ， $|E|$ 表示图的边数。

简单图：不存在重复边，不存在顶点到自身的边。

多重图：与简单图相对

完全图（简单完全图）：对于无向图 n 个节点有 $n(n - 1)/2$ 个边的叫无向完全图。对于有向图 n 个节点有 $n(n - 1)$ 个边的叫有向完全图。

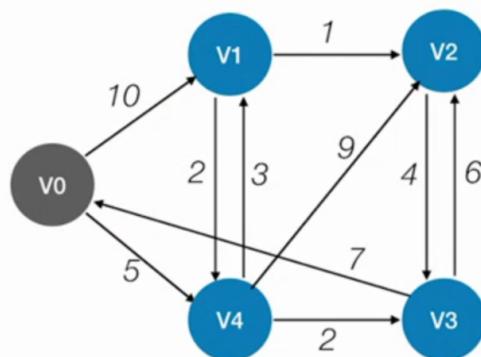
子图：对于 $G = (V, E)$ 存在 $G' = (V', E')$, $V' \in V, E' \in E$ ，则称 G' 为 G 的子图。如果 $V(G') = V(G)$ 则称 G' 为 G 的生成子图。

连通：顶点 v 到顶点 w 有路径存在则是连通的。

连通图：若 G 中任意两个顶点之间都是连通的则为连通图，否则为非连通图。

极大连通子图：连通子图中包含原图中所有的边，极小连通子图保证图连通但边最少。

连通分量：极大连通子图叫做连通分量，一个图可能有多个连通分量。



三个连通分量

强连通图：（针对有向图），两个顶点之间都有路径叫做强连通。任何两个顶点强连通叫做强连通图。

生成树：极小连通子图，去掉任何一条边都会变的不连通。

度：无向图的度=边数*2。有向图的度=出度+入度。

简单路径：不包含重复节点的路径

简单回路：除了起点之外其他顶点不重复。

最小生成树

包含所有顶点且边数最小的连通子图

(1) **最小生成树性质**：设 $G = (V, E)$ 是连通带权图， U 是 V 的真子集。如果 $(u, v) \in E$, 且 $u \in U, v \in V - U$ ，且在所有这样的边中， (u, v) 的权 $c[u][v]$ 最小，那么一定存在 G 的一棵最小生成树，它以 (u, v) 为其中一条边。这个性质有时也称为 MST 性质。

证明：设 G 的任何一棵最小生成树都不含边 (u, v) , 将边 (u, v) 添加到 G 的一棵最小生成树 T 上, 将产生含有边 (u, v) 的圈, 并且在这个圈上有一条不同于 (u, v) 的边 (u', v') 使 $u' \in U, v' \in V - U$, 将边 (u', v') 删除, 得到 G 的另一棵生成树 T' , 由于 $c[u][v] <= c[u'][v']$, 所以 T' 的耗费 $<= T$ 的耗费。于是 T' 是一棵包含边 (u, v) 的最小生成树, 与假设矛盾。

Prim 算法

难度：☆☆

设 $G = (V, E)$ 是连通带权图, $V = \{1, 2, \dots, n\}$ 。构造 G 的最小生成树的 Prim 算法的基本思想是：首先置 $S = \{1\}$, 然后, 只要 S 是 V 的真子集, 就作如下的贪心选择：选取满足条件 $i \in S, j \in V - S$, 且 $c[i][j]$ 最小的边, 将顶点 j 添加到 S 中。这个过程一直进行到 $S = V$ 时为止。在这个过程中选取到的所有边恰好构成 G 的一棵最小生成树。利用最小生成树性质和数学归纳法容易证明, 上述算法中的边集合 T 始终包含 G 的某棵最小生成树中的边。因此, 在算法结束时, T 中的所有边构成 G 的一棵最小生成树, Prim 算法所需要的计算时间为 $O(n^2)$ 。

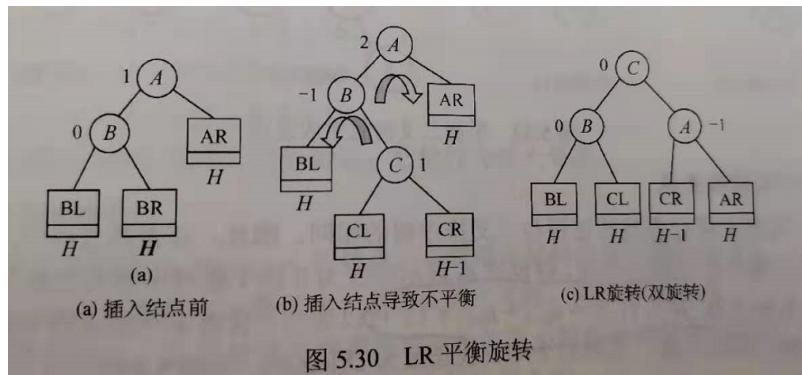


图 5.30 LR 平衡旋转

用 S 表示生成树的点集合, V 表示所有的点集, 用 $V - S$ 表示未加入生成树的点集。

用 $c[i][j]$ 来表示点 i 到点 j 之间的权值。使用 $\text{closest}[i]$ 表示集合 S 到节点 i 加入到集合时候连接的顶点; 用 $\text{lowcost}[i]$ 表示生成树点集到节点 i 的最小权值。

这种思路是将整个 S 集合当成一个点, 不需要再遍历所有节点之间的边找到最小值, 使用 lowcost 数组一并解决。

```
#define INF 0x3f3f3f3f
void prim(int n, int **c) { //c 表示每个结点之间边的耗费
    int lowcost[n+1]; //V-S集合中与生成树点集S各点之间的最小值
    int closest[n+1]; //V-S集合中与生成树点集S各点最近点
    bool s[n]; //用于标记节点是否在S中
    s[0] = true; //先将初始结点加入到S集中
    for(int i=1;i<n;i++) { //初始化
        lowcost[i]=c[0][i];
        closest[i]=0;
    }
    for(int i=1;i<n;i++) { //一共再构建n-1条边
        int j = 1, min = INF;
        for(int k=1;k<n;k++) {
            if(k!=i&&!s[k]&&lowcost[k]<min) // 找到V-S点集中且与生成树点集S最近的点
                min = lowcost[k];
            j = k;
        }
        s[j] = true; //把点j加入到S中
        for(int k=1;k<n;k++) {
            if(!s[k]&&c[j][k]<lowcost[k]) {
                lowcost[k]=c[j][k]; // 更新V-S结点中与生成树点集S之间的最小值
                closest[k] = j; // 记录V-S集合中离S集合中最近的点j
            }
        }
    }
}
```

时间复杂度 $O(n^2)$, 空间复杂度 $O(n)$

Kruskal算法

Kruskal 算法构造 G 的最小生成树的基本思想是，首先将 G 的 n 个顶点看成 n 个孤立的连通分支。将所有的边按权从小到大排序。然后从第一条边开始，依边权递增的顺序查看每一条边，并按上述方法连接 2 个不同的连通分支：当查看到第 k 条边(v,w)时，如果端点 v 和 w 分别是当前 2 个不同的连通分支 T1 和 T2 中的顶点时，就用边(v,w)将 T1 和 T2 连接成一个连通分支，然后继续查看第 k+1 条边；如果端点 v 和 w 在当前的同一个连通分支中，就直接再查看第 k+1 条边。这个过程一直进行到只剩下一个连通分支时为止。

并查集+优先队列（并查集超纲了？）

最短路径

带权单源最短路径——Dijkstra算法

不适合负权值带权图 难度：☆☆

思想：

设置一个顶点集合 S 并不断地作贪心选择来扩充这个集合。一个顶点属于集合 S 当且仅当从源到该顶点的最短路径长度已知。初始时，S 中仅含有源。设 u 是 G 的某个顶点，从源到 u 且中间只经过 S 中顶点的路称为从源到 u 的特殊路径，并用数组 dist 来记录当前每个顶点所对应的最短特殊路径长度。该算法每次从 V-S 中取出具有最短特殊路长度的顶点 u，将 u 添加到 S 中，同时对数组 dist 作必要的修改。S 包含所有 V 中顶点时，dist 就记录了从源到所有其他顶点的最短路径。

2. 贪心选择性质：Dijkstra 算法所做的贪心选择是从 V-S 中选择具有最短特殊路径的顶点 u，从而确定从源到 u 的最短路径长度 dist[u]。这种贪心选择能导致最优解，是因为，如果存在一条从源到 u 且长度比 dist[u]更短的路，设这条路初次走出 S 之外到达的顶点为 x \in V-S，然后徘徊于 S 内外若干次，最后离开 S 到达 u。在这条路径上，分别记 d(v, x), d(x, u) 和 d(v, u) 为顶点 v 到顶点 x, 顶点 x 到顶点 u 和顶点 v 到顶点 u 的路长，那么，dist[x]≤d(v, x), d(v, x) + d(x, u) = d(v, u)

3. 最优子结构性质：如果 $P(i, j) = V_i \dots V_k \dots V_s \dots V_j$ 是从顶点 i 到 j 的最短路径，k 和 s 是这条路径上的一个中间顶点，那么 $P(k, s)$ 必定是从 k 到 s 的最短路径。下面证明该性质的正确性。证明：假设 $P(i, j) = V_i \dots V_k \dots V_s \dots V_j$ 是从顶点 i 到 j 的最短路径，则有 $P(i, j) = P(i, k) + P(k, s) + P(s, j)$ 。而 $P(k, s)$ 不是从 k 到 s 的最短距离，那必存在另一条从 k 到 s 的最短路径 $P'(k, s)$ ，那么 $P'(i, j) = P(i, k) + P'(k, s) + P(s, j)$

需要记录三个信息，是否已经被遍历 final 数组，最短路径的数值 dist 数组以及路径前驱 path (非必要)。

	v0	v1	v2	v3	v4
final	✓	✗	✗	✗	✗
dist	0	8	14	7	5
path	-1	4	4	4	0

过程：

	v0	v1	v2	v3	v4
final	✓	✗	✗	✗	✓
dist	0	8	14	7	5
path	-1	4	4	4	0

名称	数据对象	稳定性	时间复杂度		额外空间复杂度
			平均	最坏	
冒泡排序	数组	✓	$O(n^2)$		$O(1)$
选择排序	数组 链表	✗ ✓	$O(n^2)$		$O(1)$
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$
堆排序	数组	✗	$O(n \log n)$		$O(1)$
归并排序	数组 链表	✓	$O(n \log^2 n)$	$O(1)$	
			$O(n \log n)$	$O(n) + O(\log n)$	如果不是从下到上 $O(1)$
快速排序	数组 链表	✗ ✓	$O(n \log n)$	$O(n^2)$	$O(\log n)$
希尔排序	数组	✗	$O(n \log^2 n)$	$O(n^2)$	$O(1)$
计数排序	数组、链表	✓	$O(n + m)$		$O(n + m)$
桶排序	数组、链表	✓	$O(n)$		$O(m)$
基数排序	数组、链表	✓	$O(k \times n)$	$O(n^2)$	

代码：可以在找后续节点的时候可以使用堆的思想

```
void dijkstra(vector<vector<int>> map, vector<int>& dist, vector<int>& path) {
    // map可以为邻接矩阵或者是邻接表
    // dist表示节点A0到其他节点的距离 path表示每个节点的前驱
```

```

int n = map.size();
vector<bool> final(n, false);
dist = vector<int>(n, -1); // -1 表示不可达
path = vector<int>(n, -1);
dist[0] = 0;
int p = 0; //当前的起点
while(true) {
    int len = dist[p];
    for (int i = 0; i < n; ++i) {
        if(final[i]==false&&map[p][i]!=-1) {//未遍历过并且与p连接的点
            if(map[p][i] + len < dist[i] || dist[i]==-1) {
                dist[i] = map[p][i] + len;
                path[i] = p;
            }
        }
    }
    final[p] = true;
    //检查节点是否全部遍历完毕，如果未完毕并且找到下一个起点
    bool over = true;
    int length = INF;
    for (int i = 0; i < n; ++i)
        if(final[i]==false) {
            over = false;
            if(dist[i]!=-1&&dist[i]<length) {
                length=dist[i];
                p = i;
            }
        }
    if(over)break;
}
}

```

► 点击展开调试代码：

带权多源最短路径——Floyd算法

思想：动态规划，很典型的区间DP—— $O(n^3)$

```

//有向图
int floyd(int **graph, int **path, int n) {
    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++) {
            for(int k=0;k<n;k++) { //中转点
                if(graph[i][k]+graph[k][j]<graph[i][j]) {
                    graph[i][j]=graph[i][k]+graph[k][j];
                    path[i][j] = k; //path数组记录i到j的中转点
                }
            }
        }
    }
}

```

拓扑排序

难度：☆☆

课程表 II <https://leetcode-cn.com/problems/course-schedule-ii/>

【题目】现在你总共有 n 门课需要选，记为 0 到 $n-1$ 。

在选修某些课程之前需要一些先修课程。例如，想要学习课程 0，你需要先完成课程 1，我们用一个匹配来表示他们：
[0,1]

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

【示例】

输入：4, [[1,0], [2,0], [3,1], [3,2]]

输出：[0,1,2,3] or [0,2,1,3]

解释：总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。

因此，一个正确的课程顺序是 [0,1,2,3]。另一个正确的排序是 [0,2,1,3]。

```
vector<int> topology(int n, vector<vector<int>>& prerequisites) {
    vector<int> f(n, 0);
    queue<int> sons[n];
    for(int i=0;i<prerequisites.size();i++) {
        f[prerequisites[i][0]]++;
        sons[prerequisites[i][1]].push(prerequisites[i][0]);
    }
    vector<int> ret;
    int s = n, p=0, t;
    while(s) {
        p=0;
        while(p<n&&f[p]!=0) p++;
        if(p==n) return {};//无解
        while(!sons[p].empty()) {
            t = sons[p].front();
            sons[p].pop();
            f[t]--;
        }
        f[p] = -1;//标记为已经用过
        ret.push_back(p);
        s--;
    }
    return ret;
}
```

排序

基数排序，时间复杂度 = O(d(n+r))

$\approx O(30000)$

若采用 $O(n^2)$ 的排序， $\approx O(10^8)$

若采用 $O(n \log_2 n)$ 的排序， $\approx O(140000)$

插入排序

难度：☆

```

void insertSort(int *a, int n) {
    for (int i = 1; i < n; ++i) {
        int t = a[i], j = i - 1;
        for (; j >= 0; j--) {
            if (t < a[j]) a[j+1] = a[j];
            else break;
        }
        a[j+1] = t;
    }
}

```

希尔排序

插入排序的升级版。难度：☆

稳定性：不稳定

每一次减少增量d，分成不同的序列来进行插入排序

$d = n/2 \dots$

时间复杂度： $O(n^{1.3})$ 最坏 $O(n^2)$

```

void shellSort(int *a, int n) {
    int d = n/2, op = true;
    while(op){ //设置开关
        if(d==1)op=false;
        for(int i=d;i<n;i+=d){
            int t = a[i], j = i-d;
            for(;j>=0;j-=d){
                if(t < a[j]) a[j+d] = a[j];
                else break;
            }
            a[j+d] = t;
        }
        d/=2; //每次增量减半
    }
}

```

冒泡排序

难度：☆

EZ，可以用于链表， $O(n^2)$

```

void bubbleSort(int *a, int n) {
    for(int i=n;i>0;i--){
        for(int j=1;j<i;j++)
            if(a[j]<a[j-1]) swap(a[j],a[j-1]);
    }
}

```

快速排序

冒泡排序进阶版——不稳定。难度：☆☆

从列表中选取一个基准pivot，小于的都在pivot左边，大的在右边，依次递归。

时间复杂度 $O(n \times \text{递归层数})$, $\min = O(n \log_2 n)$

如果是逆序或者顺序的，复杂度会更高

解决方法：

- 随机选取一个元素
- 比较头、中、尾三个值进行对比，取中间值。

平均性能最优秀

无随机化代码 (deprecated) :

```
int partition(int s, int l, int *a) {
    int t = a[s];
    while(s<l) {
        while(s<l&&a[s]>t) l--;
        a[s] = a[l];
        while(s<l&&a[s]<=t) s++;
        a[l] = a[s];
    }
    a[l] = t;
    return l;
}

void quickSort(int *a, int s, int l) {
    if(s>=l) return;
    int pivot = partition(s, l, a);
    quickSort(a, s, pivot-1);
    quickSort(a, pivot+1, l);
}
```

随机化取中值代码 (推荐) :

```
void quicksort(int *a, int low, int high) {
    if(low >= high) return;
    int x = a[(low + high) / 2], i = low - 1, j = high + 1;
    while(i < j) {
        do i++; while(a[i] < x); //与x相同的情况会直接略过
        do j--; while(a[j] > x);
        if(i < j) swap(a[i], a[j]);
    }
    quicksort(low, j); //最终a[j] < x, a[i] > x, 因此分界点为j
    quicksort(j+1, high);
}
```

选择排序

每次选择序列中的最小值放到头部——不稳定

$O(n^2)$

堆排序

选择排序的进阶版

大根堆和小根堆

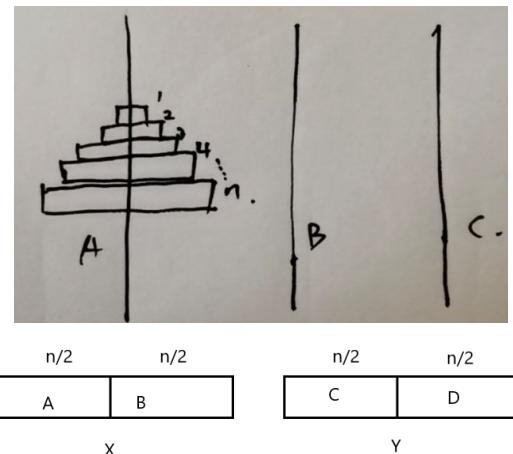
建堆的时间复杂度 $O(n)$, 排序的过程 $O(n \log_2 n)$

归并排序

将两个有序的序列合并为一个, 如果遇到n重归并排序需要使用到败者树。

基数排序

稳定，通常用链表实现



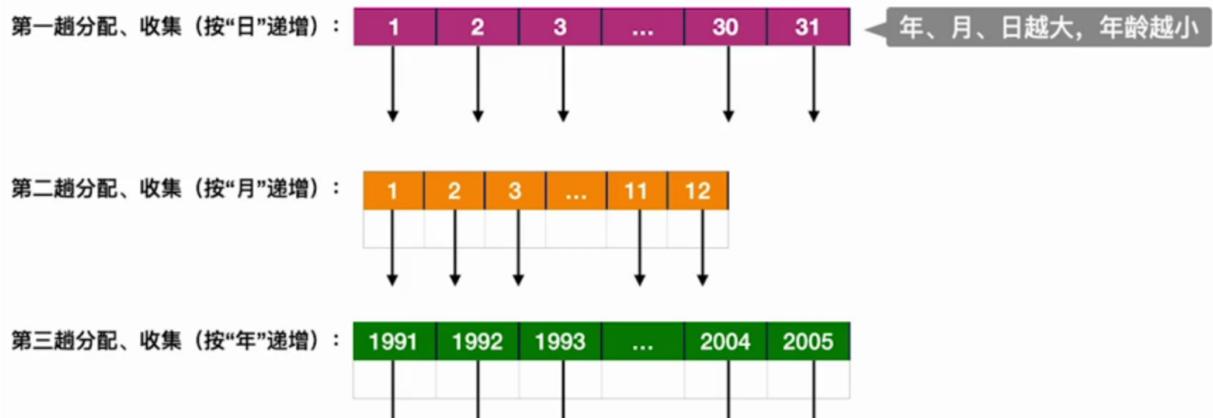
应用情况：

基数排序的应用

中国大学MOOC

某学校有 10000 学生，将学生信息按年龄递减排序

生日可拆分为三组关键字：年(1991~2005)、月(1~12)、日(1~31) 权重：年>月>日



桶排序

使用前提：数据均匀分布。桶的数量 = \sqrt{n}

```
void bucketSort(int *a, int n){  
    int bucketCounts = sqrt(n) + 1;  
    vector<node*> buckets(bucketCounts, new node(0));  
    for(int i=0;i<n;i++){  
        node* head = buckets[a[i]/bucketCounts];  
        head = insert(head, a[i]); //这里是有序插入  
    }  
    node* h;  
    for(int i=0;i<n;i++) h=merge(buckets[i]); //有序合并各个桶的链表  
    while(h!=NULL){ //输出  
        cout<<h->data<<" "  
        h = h->next;  
    }  
}
```

递归与分治

有的题需要分清楚组合问题还是排列问题

(1) 将一个规模为 n 的问题分解为 k 个规模较小的子问题，这些子问题相互独立且与原问题相同。递归地解这些问题，然后将各子问题的解合并得到原问题的解。

(2) 分治法所能解决的问题一般具有以下几个特征：问题的规模缩小到一定的程度就可以容易地解决；该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质；该问题分解出的子问题的解可以合并为该问题的解；该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。在用分治法设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治算法的时间复杂度递归式：

$$T(n) = \begin{cases} O(1) & n = 1 \\ kT(n/m) + f(n) & n > 1 \end{cases}$$

规模为 n 的问题分为 k 个规模为时间复杂度为 n/m 的子问题的时间复杂度。

全排列

全排列模板 难度：☆☆

```
void permutation(int *a, int low, int high){  
    if(low==high){  
        for (int i = 0; i <= high; ++i) printf("%d ", a[i]);  
        printf("\n");  
    } else{  
        for (int i = low; i <= high; ++i) {  
            swap(&a[low], &a[i]);  
            permutation(a, low +1, high);  
            swap(&a[low], &a[i]); //再换回来  
        }  
    }  
}
```

斐波那契数列

$$\text{递推式: } f(n) = \begin{cases} 0, & n \leq 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n \geq 2 \end{cases}$$

递归法：难度：☆

```
int f(int n){  
    if(n<=0) return 0;  
    else if(n==1) return 1;  
    else return f(n-1)+f(n-2);  
}
```

时间复杂度 $O(2^n)$

循环法：难度：☆

```

int fabonacci(int n) {
    if(n<=1) return n;
    int a = 0, b = 1;
    while(n>=2) {
        b = a + b;
        a = b - a;
        n--;
    }
    return b;
}

```

时间复杂度: $O(n)$

线性代数斐波那契求解: 难度: $\star\star$

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

```

int m2[][] = {{1,1}, {1,0}};

void matrixMulti(int a[][], int b[][]) { //矩阵乘法
    int x = a[0][0]*b[0][0]+a[0][1]*b[1][0];
    int y = a[0][0]*b[0][1]+a[0][1]*b[1][1];
    int z = a[1][0]*b[0][0]+a[1][1]*b[1][0];
    int q = a[1][0]*b[0][1]+a[1][1]*b[1][1];
    a[0][0]=x, a[0][1]=y, a[1][0]=z, a[1][1]=q;
}

void matrixPower(int m[][], int n) { //矩阵幂运算
    if(n<=1) return;
    matrixPower(m, n/2); //只求解一半 logn的主要解决方式
    matrixMulti(m, m);
    if(n%2==1) matrixMulti(m, m2);
}

int fib(int n) {
    if(n<=1) return 1;
    int m[][] = {{1,1}, {1,0}};
    matrixPower(m, n-1);
    return m[0][0];
}

```

时间复杂度 $O(\log n)$

公式法:

$$F(n) = \frac{\sqrt{5}}{5} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right]$$

整数划分

有使用到动态规划的思想

<http://bailian.openjudge.cn/practice/4117>

【题目】整数划分，是指把一个正整数n表示成系列正整数之和：例如正整数6有如下11种不同的划分，所有p(6)=11，为以下11种情况

```

5+1
4+2, 4+1+1
3+3, 3+2+1, 3+1+1+1
2+2+2, 2+2+1+1, 2+1+1+1+1
1+1+1+1+1+1

```

代码：

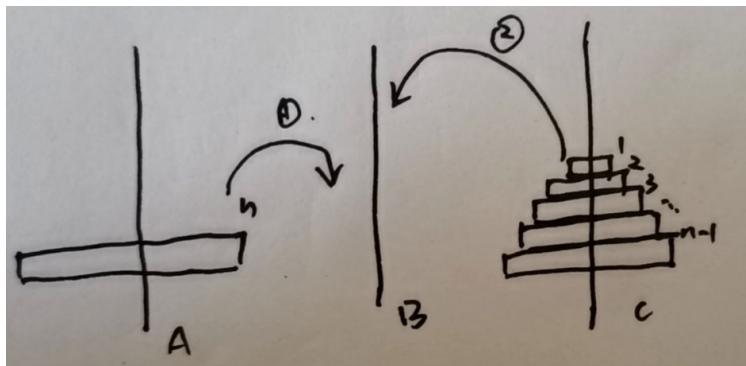
```

//m为需要划分的整数，n为划分的最大值
int digitDivision(int m, int n) {
    if(m<1 || n<1) return 0;
    if(m==1 || n==1) return 1;
    if(m==n) return digitDivision(m, n-1) + 1;
    else if(m<n) return digitDivision(m, m);
    return digitDivision(m, n-1)+digitDivision(m-n, n);
}

```

汉诺塔问题

难度：☆



基数排序，时间复杂度 = $O(d(n+r))$

基数排序擅长解决的问题：

- ①数据元素的关键字可以方便地拆分为 d 组，且 d 较小 反例：给5个人的身份证号排序
- ②每组关键字的取值范围不大，即 r 较小 反例：给中文人名排序
- ③数据元素个数 n 较大 擅长：给十亿人的身份证号排序

XXXXXXXXXXXXXXXXXXXX

赵 钱 孙 ... 杰 伦 龚 谌 或 ...

【要求】

将A上所有的盘子都移到B上，并且大盘子不允许在小盘子上面。

【思想】

把上面N-1层都送到C，再把第N层送到B，再将C的N-1层送到B。

由于已经做好第N层的操作之后，第N层的盘子是最大的，所有的盘子都比其小，因此第N层盘子可以被看作是“隐形”的，其他小盘子可以在大盘子上任意移动。

```

void hanoi(int n, int a, int b, int c) {
    //n表示盘子的数量，表示把盘子都从a送到b，其中c为辅助。a, b, c表示盘子的数量。
    if(n>0) {
        hanoi(n-1, a, c, b);
        move(a, b); //将a上面最大的移动到b
        hanoi(n-1, c, b, a);
    }
}

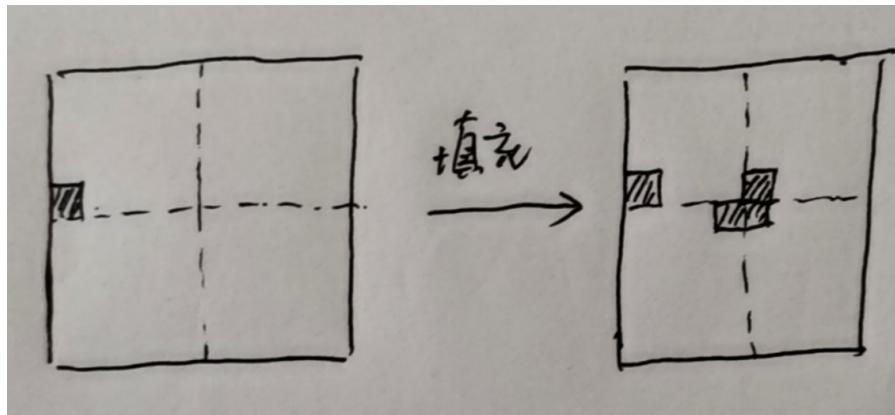
```

二分查找

用于查找已经升序或者降序的序列中的元素 难度: ☆

```
int binSearch(int *a, int len, int num) {
    int i = 0, j = len - 1, mid;
    while(i <= j) {
        mid = (i+j)/2;
        if(a[mid]==num) return mid;
        if(a[mid]<num) i = mid + 1;
        else j = mid - 1;
    }
    return -1; //未找到返回-1
}
```

大整数乘法



两个大整数X和Y相乘，可分为两个部分相互相乘，X和Y可以分解为以下部分：

$$X = A \times 2^{\frac{n}{2}} + B$$

$$Y = C \times 2^{\frac{n}{2}} + D$$

$$XY = AC \times 2^n + (AD + BC) \times 2^{\frac{n}{2}} + BD$$

最终需要进行4次 $n/2$ 幂次的乘法 (AC, AD, BC, BD) , 2次移位运算，三次加法运算

时间复杂度：

$$T(n) = \begin{cases} O(1) & n=1 \\ 4T(n/2) + O(n) & n>1 \end{cases}$$

可知时间复杂度为 $O(n^{\log 4})=O(n^2)$

改良之后：

$$XY = AC \times 2^n + ((A - B)(D - C) + AC + BD) \times 2^{\frac{n}{2}} + BD$$

改良之后只需要计算三次乘法：AC, BD, (A - B)(D - C)

改进后复杂度：

$$T(n) = \begin{cases} O(1) & n=1 \\ 3T(n/2) + O(n) & n>1 \end{cases}$$

可知时间复杂度为 $O(n^{\log 3})=O(n^{1.59})$

strassen矩阵乘法

8次乘法运算减到7次，使用特定的公式，减少重复计算。需要记住特殊的公式，感觉考的可能性较小。

$$T(n) = O(n^3)$$

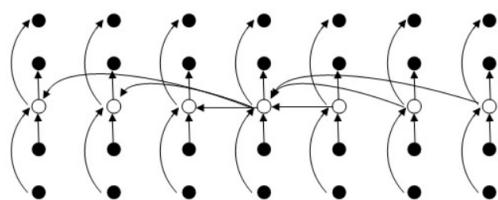
$$T(n) = \begin{cases} O(1), & n = 2 \\ 7T(n/2) + O(n^2) & n > 2 \end{cases}$$

$$T(n) = O(n \log 7)$$

棋盘填充

理解分治的最好样例。难度：☆

当 $k > 0$ 时，将 $2k \times 2k$ 棋盘分割为 4 个 $2k - 1 \times 2k - 1$ 子棋盘。特殊方格必位于 4 个较小子棋盘之一中，其余 3 个子棋盘中无特殊方格。为了将这 3 个无特殊方格的子棋盘转化为特殊棋盘，可以用一个 L 型骨牌覆盖这 3 个较小棋盘的会合处，从而将原问题转化为 4 个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘 1×1 。



填充一个L型方块之后，将原问题又化为了4个相同的子问题，然后继续划分。

快速排序

难度：☆☆

```
int partition(int *a, int low, int high) {
    int tmp = a[low];
    while (low < high) {
        while (high > low && a[high] >= tmp) high--;
        a[low] = a[high];
        while (low < high && a[low] < tmp) low++;
        a[high] = a[low];
    }
    a[low] = tmp;
    return low;
}

void quickSort(int *grp, int low, int high) {
    if (low >= high) return;
    int pivot = partition(grp, low, high);
    quickSort(grp, pivot + 1, high);
    quickSort(grp, low, pivot - 1);
}
```

随机快速排序

线性时间选择（第K小的数）

难度：☆☆

普通线性时间选择：

由快速排序算法而启发，在数组 $a[p:r]$ 中想要找第 k 小的数，使用 partition 函数将 $a[p:r]$ 分为两个部分 $a[p:i]$ 和 $a[i+1:r]$ 两个部分， $a[p:i]$ 中所有的数都小于 $a[i+1:r]$ 中的数。

```

int selectPartition(int *a, int low, int high) {
    int t = a[low];
    while (low < high) {
        while (low < high && a[high] > t) high--;
        a[low] = a[high];
        while (low < high && a[low] <= t) low++;
        a[high] = a[low];
    }
    a[low] = t;
    return low;
}

int selectIndexK(int* a, int low, int high, int k) {
    int pivot = selectPartition(a, low, high);
    if (k == pivot) return a[k];
    if (k < pivot) return selectIndexK(a, low, pivot - 1, k);
    else return selectIndexK(a, pivot + 1, high, k);
}

```

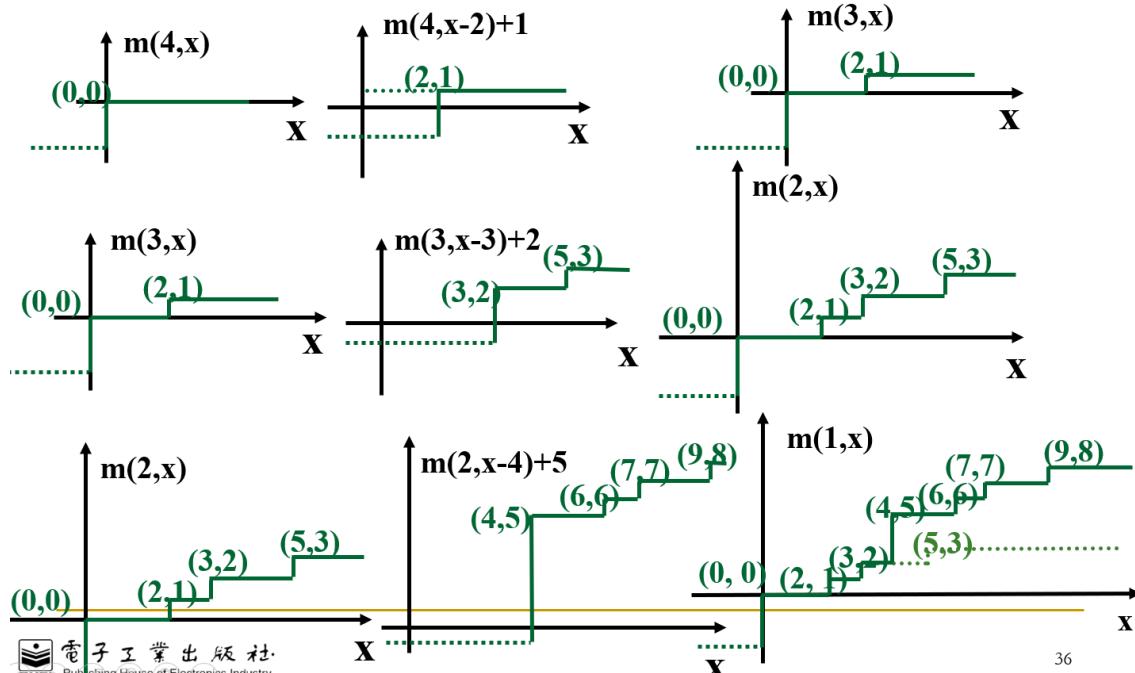
进阶版：

在普通情况下，线性时间选择最坏情况下时间复杂度仍然是 $O(n^2)$ 。为了改进这种情况。对于75个数量以下的数组直接进行排序(那我还用你干什么)。当大于75时分为5组找其中位数，找到的中位数数组中再找中位数，以此类推。

将 n 个输入元素划分成 $\lceil n/5 \rceil$ 个组，每组 5 个元素，只可能有一个组不是 5 个元素。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共 $\lceil n/5 \rceil$ 个。递归调用 select 来找出这 $\lceil n/5 \rceil$ 个元素的中位数。如果 $\lceil n/5 \rceil$ 是偶数，就找它的 2 个中位数中较大的一个，以这个元素作为划分基准。设所有元素互不相同。在这种情况下，找出的基准 x 至少比 $3(n-5)/10$ 个元素大，因为在每一组中有 2 个元素小于本组的中位数，而 $n/5$ 个中位数中又有 $(n-5)/10$ 个小于基准 x 。同理，基准 x 也至少比 $3(n-5)/10$ 个元素小。而当 $n \geq 75$ 时， $3(n-5)/10 \geq n/4$ 所以按此基准划分所得的 2 个子数组的长度都至少缩短 $1/4$ 。

一个例子

$n=3, c=6, w=\{4, 3, 2\}, v=\{5, 2, 1\}$ 。



最接近点对点 (未做)

循环赛日程表

难度：☆ 能看出来就是最简单的，看不出来那就难了：(

左上角和右下角是相同的，复制即可

循环形式：

```
void agenda(int k) {
    int len = 1<<k;
    int ** a = array(len+1, len+1);
    for (int i = 1; i <= len; ++i)a[i][1] = i;
    for (int size = 1; size <= len/2; size*=2) {      //需要复制的方块的尺寸
        for (int num = 0; num < len / size / 2; ++num) {      // 每个相同大小需要复制方块的对数
            for (int m = 1; m <= size; ++m) {          // 方块复制
                for(int n = 1;n<= size;++n) {
                    a[m+size*(1+num*2)][n+size] = a[m+size*(num*2)][n];           // 行变列不变
                    a[m+size*(num*2)][n+size] = a[m+size*(1+num*2)][n];
                }
            }
        }
    }
}
```

合并k个排序链表

<https://leetcode-cn.com/problems/merge-k-sorted-lists>

【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

分治法：

这种递归的方式值得学习

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

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(a->val<=b. 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 h) {
    if(l>h) return NULL;
    if(l==h) return lists[l];
    int mid = (l+h)/2;
    return mergeTwoLists(merge(lists, l, mid), merge(lists, mid+1, h));
}

ListNode* mergeKLists(vector<ListNode*>& lists) {
    return merge(lists, 0, list.size()-1);
}

```

课后习题

算法分析：

2-3

```

int binSearch(int a[], int n, int &i, int &j, int x) {
    int low = 0, high = n-1;
    i=j=-1;
    while(low<=high) {
        int mid = (low + high)/2;
        if(a[mid]==x) {
            i=j=mid;
            return 1;
        }
        if(a[mid]<x) low = mid +1;
        else high = mid -1;
    }
    if(low>=0&&low<n) j=low;
    if(high>=0&&high<n) i=high;
    return 0;
}

```

算法实现

2-1众数问题

【Description】

给定含有n个元素的多重集合S，每个元素在S中出现的次数称为该元素的重数。多重集S中重数最大的元素称为众数。例如，S={1, 2, 2, 2, 3, 5}。多重集S的众数是2，其重数为3。对于给定的由n个自然数组成的多重集S，计算S的众数及其重数。**如果出现多个众数，请输出最小的那个。**

【Input】

输入数据的第1行是多重集S中元素个数n (n<1300000)；接下来的n行中，每行有一个最多含有5位数字的自然数。

【Output】

输出数据的第1行给出众数，第2行是重数。

【Input】

```

6
1
2
2
2
3
5

```

【 Output 】

```
2  
3
```

分治法：

哈希表法O(n)：难度：☆

```
#include<bits/stdc++.h>  
using namespace std;  
const int N = 1000+10;  
int a[N], t;  
int main() {  
    int n;  
    while(~scanf("%d", &n)) {  
        memset(a, 0, N*4);  
        for (int i = 0; i < n; ++i) {  
            scanf("%d", &t);  
            a[t]++;  
        }  
        int m = 0, k;  
        for (int i = 0; i < N; ++i)  
            if(a[i]>m) {  
                m=a[i];  
                k=i;  
            }  
        printf("%d\n", k);  
    }  
    return 0;  
}
```

【注意】摩尔投票法找的不是众数，而是多数元素（数目大于 $n/2$ 的元素，可以改为选举法）

2-3半数集问题

难度：☆

<https://acm.sdu.edu.cn/onlinejudge3/problems/1713>

```
int halfNum(int n) {  
    int ans = 1;  
    for (int i = 1; i <= n/2; ++i) {  
        ans += halfNum(i);  
    }  
    return ans;  
}
```

2-4半数单集问题

难度：☆☆

<https://acm.sdu.edu.cn/onlinejudge3/problems/1714>

特例：比如24会遇到重复的数字，12·24和1·2·24，那遇到重复的怎么办？那就删掉重复的呗

```

int a[200+10];

int halfNum(int n) {
    if(n<=1) return n;
    int ans = 1;
    if(a[n]>0) return a[n];
    for (int i = n/2; i >=1 ; --i) {
        ans += halfNum(i);
        if((i>10)&&2*(i/10)<=i%10)ans-=a[i/10];
    }
    a[n]=ans;
    return ans;
}

```

2-5 有重复元素的排列问题

一般的全排列都是abcdef的形式，没有重复元素。如果含有重复元素比如aab，则 a_1a_2b 和 a_2a_1b 就属于同一种元素。由于全排列是按序从左向右递归的形式一层一层递归，因此需要保证每次的递归“头”只保证出现一次即可。

```

bool isRepeating(char *a, int i, int k) {
    //检验a[i:k-1]中是否有与a[k]相同的元素
    for(;i<k;i++)
        if(a[i]==a[k]) return true;
    return false;
}

void perm(char *a, int i, int n) {
    if(i==n) cout<<a<<endl;
    else{
        for(int k=i;k<n;k++) {
            if(isRepeating(a, i, k)) continue;//a[k]已经当过“头”了，则跳过
            swap(a[i], a[k]);
            perm(a, i+1, n);
            swap(a[i], a[k]);
        }
    }
}

```

2-6 排列的字典序

关键词：字典序 难度：☆☆☆

A. 暴力搜索法 (超时1010ms)：

```

#include <bits/stdc++.h>
#define MAX 20+5
using namespace std;

int a[MAX], b[MAX];
int cnt=0, sign=-2;

bool perm(int i, int n) {
    if(i==n) {
        bool s = true;
        for(int m=0;m<n;m++)
            if(a[m]!=b[m]) {      //不对应则跳出循环
                s=false;
                break;
            }
        if(s){ //完全相等，则记录字典序
            sign=cnt;
            cout<<cnt<<endl;
        }
        if(sign==sign+1){
            cout<<a[0];      //成功找到下一个序列，输出序列
        }
    }
}

```

```

        for(int m=1;m<n;m++) cout<<" "<<a[m];
        cout<<endl;
        return true;
    }
    cnt++;
    return false;
}

for (int j = i; j < n; ++j) {
    int tmp = a[j];
    for (int k = j; k > i; k--) a[k]=a[k-1];
    a[i]=tmp; //保证字典序，不用swap
    if(perm(i+1,n))return true; //如果找到下一个序列就退出，剪枝
    tmp = a[i]; //复原数组
    for(int k=i;k<j;k++) a[k]=a[k+1];
    a[j]=tmp;
}
return false;
}

int main() {
    int n;
    scanf("%d", &n);
    for(int i=0;i<n;i++) {
        a[i]=i+1;
        scanf("%d", &b[i]);
    }
    perm(0,n);
    return 0;
}

```

B. 公式法计算字典序：

步骤一：由排列计算出字典序值：

设给定的 $\{1, 2, \dots, n\}$ 的排列为 π ，设其字典序的值为 $\text{rank}(\pi, n)$ ，那么其对应的字典序值显然存在：

$$(\pi[1] - 1) \cdot (n - 1)! \leq \text{rank}(\pi, n) \leq \pi[1] \cdot (n - 1)! - 1$$

设 r 为 π 在以 $\pi[1]$ 开头的所有排列中的序号，那么就可以得到 r 也是集合 $\{1, 2, \dots, n\} - \{\pi[1]\}$ 中序列为 $[\pi[2], \pi[3], \dots, \pi[n]]$ 的序号。那么将 π 中所有大于 $\pi[1]$ 的元素都减一就会得到集合 $\{1, 2, \dots, n - 1\}$ 的排列 π' ，其字典序也是 r ，由此就得到计算 $\text{rank}(\pi, n)$ 的递归公式：

$$\text{rank}(\pi, n) = (\pi[1] - 1) \cdot (n - 1)! + \text{rank}(\pi', n - 1)$$

$$\text{其中 } \pi'[i] = \begin{cases} \pi[i + 1] - 1, & \pi[i + 1] > \pi[i] \\ \pi[i + 1], & \pi[i + 1] < \pi[i] \end{cases}$$

步骤二：由排列计算出下一个排列：

由一个字典序找到下一个字典序，比如序列 26458173，首先改变的数值一定是从个位开始。不改变个位以上的值能否将序列增大？不能；不改变十位以上的值只改变 “73” 能否将序列的值增大？不能；不改变百位以上的值只改变 “173” 能否将序列的值增大？可以，因为存在序列 “317, 371...” > “173”。观察可以得到以下规律，即找到下标为 i 的数字满足 $a[i] < a[i + 1]$ ，且对于 $i + 1$ 之后的序列都有 $a[i + 1] > a[i + 2] > \dots > a[n]$ 。找到下标 i 之后如何从 “317, 371, 713, 731” 中找到最适合的序列？显然这个序列是 “317”。因此还应该找到下标 j 满足 $a[j]$ 刚好大于 $a[i]$ 即 $a[j + 1], a[j + 2], \dots, a[n] < a[i], a[j] > a[i]$ ，然后 $\text{swap}(a[i], a[j])$ ，将 $a[i + 1 : n]$ 进行逆序处理即可得到下一个数列。

程序如下：

```

#include <bits/stdc++.h>
#define MAX 20+5
using namespace std;

int a[MAX], b[MAX], fac[MAX];

int dictIndex(int a[], int n){ //计算字典序
    int s = 0;

```

```

        for (int j = 0; j < n; ++j) {
            for(int i=j+1;i<n;i++) if(a[i]>a[j])a[i]--; //对a[j+1:n]大于a[j]元素进行-1
            s+=(a[j]-1)*fac[n-j-1]; // 加每层的序号 (π[i]-1)*(n-i-1) !
        }
        return s;
    }

int main() {
    int n;
    scanf("%d", &n);
    fac[0]=1;
    for(int i=0;i<n;i++) {
        fac[i+1]=(i+1)*fac[i];
        scanf("%d", &b[i]);
        a[i]=b[i];
    }
    cout<<dictIndex(a, n)<<endl; //得到字典序
    int i=n-2;
    while(b[i]>b[i+1]) i--; //找到可以增大的序号
    int j=i+1;
    while(j+1<n&&b[j+1]>b[i]) j++;
    swap(b[i], b[j]); //找到可以交换的序号
    i++, j=n-1;
    while(i<j) swap(b[i++], b[j--]); //对a[i+1:n]进行逆序处理
    for (int k = 0; k < n; ++k) cout<<b[k]<<" "; //输出
    return 0;
}

```

2-7 集合划分问题

<https://acm.sdu.edu.cn/onlinejudge3/problems/1718> (! OJ为设置溢出)

知识点: Bell 数

贝尔数适合递推公式:

$$B_{n+1} = \sum_{k=0}^n \binom{n}{k} B_k = \sum_{k=0}^n C_n^k B_k.$$

上述组合公式的证明:

可以这样来想, B_{n+1} 是含有 $n+1$ 个元素集合的划分的个数, 考虑元素 b_{n+1}

假设他被单独划分到一类, 那么还剩下 n 个元素, 这种情况下划分个数为 $\binom{n}{n} B_n$

假设他和某一个元素被划分为一类, 那么还剩下 $n-1$ 个元素, 这种情况下划分个数为 $\binom{n}{n-1} B_{n-1}$

假设他和某两个元素被划分为一类, 那么还剩下 $n-2$ 个元素, 这种情况下划分个数为 $\binom{n}{n-2} B_{n-2}$

依次类推, 得到了上述组合公式。

```

#include <bits/stdc++.h>
#define MAX 20+5
using namespace std;
int fac[MAX], b[MAX];

int main() {
    int n;
    while(cin>>n) {
        memset(b, 0, 100);
        memset(fac, 0, 100);
        fac[0]=1;
        for (int i = 0; i < n; ++i) fac[i+1]=(i+1)*fac[i]; //计算阶乘
        b[0]=1;
        for (int i = 1; i <= n; ++i) {
            for(int j=0;j<i;j++)
                b[i]+=(fac[i-1]/(fac[j]*fac[i-j-1]))*b[j];
        }
    }
}

```

```

        }
        cout<<b[n]<<endl;
    }
    return 0;
}

```

补充——贝尔三角形：

1								
1	2							
2	3	5						
5	7	10	15					
15	20	27	37	52				
52	67	87	114	151	203			
203	255	322	409	523	674	877		
877	1080	1335	1657	2066	2589	3263	4140	

每行首项是贝尔bell数。每行之和是第二类Stirling数。

2-11 整数因子分解问题

不考虑性能的情况：难度：☆

时间复杂度 $O(n^{1.5})$

```

int intDivision(int n) {
    int sum = 1;
    for(int i=n-1;i>1;i--) {
        if(n%i==0)
            sum+=intDivision(n/i);
    }
    return sum;
}

```

考虑性能的情况：难度：☆☆

【注意点】：

- 这种情况下，自顶向下的递归要比自底向上的循环要好，由于除数的特殊性，不是每一个值都能用的到，因此应该选择自顶向下方式进行
- 这种方法不是动态规划
- 由于 $0 < n < 2E9$ ，并且对于自顶向下来说，n是越来越小的，因此N设置为一个较小的值就可以，没必要存储，大的n值直接由临时变量返回即可。

时间复杂度 $O(n^{1.5})$

```

#include<bits/stdc++.h>
using namespace std;
#define N 1000010
int dp[N]={0};

int ac(int n){
    if(n<N&&dp[n]!=0) return dp[n];
    int s = 1;
    for(int i=2;i<=sqrt(n);i++){
        if(n%i==0) {
            if(i*i==n) s+=ac(i);
            else s+=ac(n/i)+ac(i);
        }
    }
    if(n<N) dp[n]=s;
    return s;
}

int main() {
    int n;

```

```

    cin>>n;
    cout<<ac(n)<<endl;
    return 0;
}

```

分苹果问题

【题目描述】把M个同样的苹果放在N个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？(用k表示)
5, 1, 1和1, 5, 1是同一种分法。

【输入】第一行是测试数据的数目t ($0 \leq t \leq 20$)。以下每行均包含二个整数M和N，以空格分块。

【输出】对输入的每组数据M和N，用一行输出相应的K。

【输入样例】

```

1
7 3

```

【输出样例】

```

8

```

【解析】

其实使用到了动态规划的思想。

设问题题解为 $f(m, n)$,

递推关系式可以写为：

- 至少有一个盘子空着: $f(m, n - 1)$
- 所有盘子都至少有一个苹果: $f(m - n, n)$

结合起来: $f(m, n) = f(m, n - 1) + f(m - n, n)$

找零问题

<https://leetcode-cn.com/problems/coin-change>

【题目】：

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

```

输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1

```

递归法(最优)：

由于要求的硬币最少，所以使用贪心+分治法首先对面值大的硬币进行选择，遍历当前硬币的数量从 $[0, l]$, $l \leq \frac{n}{x_i}$ 。由于可能出现一些特殊情况，比如[8,5,1]，总面值为10的情况单纯的贪心就无法解决答案，因此应该保留不选该硬币的情况。

```

int coinChange(vector<int>& coins, int amount) {
    int res = INT_MAX;
    sort(coins.begin(), coins.end()); // 将硬币从小到大排序
    helper(coins, amount, coins.size() - 1, 0, res);
    return res == INT_MAX ? -1 : res;
}

void helper(vector<int>& coins, int amount, int start, int cur, int& res) {
    // cur表示当前硬币的数量，start表示遍历的硬币索引
    if (start < 0) return;
    if (amount % coins[start] == 0) {
        res = min(res, cur + amount / coins[start]);
        return;
    }
}

```

```

    }
    for(int i = amount/coins[start]; i >=0; i--) {
        if(cur+i>=res-1)break;
        helper(coins, amount-i*coins[start], start-1, cur+i, res);
    }
}

```

DP法，相当于完全背包问题 $dp[i] = \min\{dp[i], dp[i - j] + 1\}$:

```

int coinChange(vector<int>& coins, int amount) {
    int dp[amount+1];
    fill(dp, dp+amount+1, 0x3fffffff);
    dp[0] = 0;
    for(int i:coins) {
        for(int j=i;j<=amount;j++)
            dp[j]=min(dp[j-i]+1, dp[j]);
    }
    if(dp[amount]==0x3fffffff) return -1;
    else return dp[amount];
}

```

加减法总没有乘除法快，因此dfs的方法时间复杂度更低。

动态规划DP

思想：将问题分为多个小问题，求解多个小问题的最优解

重要属性：

最优子结构：一个问题的最优解包含其子问题的最优解

重叠子问题：例如斐波那契数列会多次求相同的子问题

无后效性：与后续阶段无关

每一步决策都依赖于前一步的决策

动态规划一般步骤：

- 刻画一个最优解的结构特征，寻找最优子结构（通常采用反证法+“复制-粘贴”法）。
- 递归定义最优解的值
- 计算最优解的值，通常采用自底向上的方式
- 构造最优解（如果有要求，即加上）

设计要素：

优化的目标函数是什么？约束条件是什么？

如何划分子问题（边界）？

原始问题的优化函数值和子问题的优化函数值有什么关系（递推方程）？

是否满足优化原则？

最小子问题如何界定？即初值是什么？

DP类型：

线性DP、树形DP、区间DP

树形DP：由叶子结点到根节点的自底向上的状态转移，一般用到回溯法。

区间DP：属于线性DP的一种，以区间长度作为DP的阶段，区间的左右端点作为区间的维度。

矩阵链乘法——区间DP

目的：选定一个最好的运行次序，使得矩阵相乘的次数最少。

思路：由于矩阵的运算满足乘法结合律，因此在不同矩阵之间插入括号乘法运算的次数也不相同。设 $A_i A_{i+1} \dots A_j$ 的加括号方案的最优解分割点在 A_k 和 A_{k+1} 之间，同样在求解 $A_i A_{i+1} \dots A_k$ 的时候也存在解，假设在求解 $A_i A_{i+1} \dots A_k$ 存在一个更优的解比原问题的解所得的代价更小，则使用更优的解替换在求解 $A_i A_{i+1} \dots A_k$ 的解，那么原问题 $A_i A_{i+1} \dots A_j$ 的解的代价会变小，与假设相矛盾，因此得到 $A_i A_{i+1} \dots A_j$ 的最优解中也包含 $A_i A_{i+1} \dots A_k$ 的最优解，则此问题具有最优子结构性质。因此设 $f(i, j)$ 表示矩阵链从 A_i 到 A_j 的最小代价，所以最小代价可以表示为 $f(i, j) = f(i, k) + f(k+1, j) + a[i] * a[k+1] * a[t+1]$ ，得到状态转移方程如下：

$$f(i, j) = \begin{cases} 0, & i \geq j \\ \min_{i \leq k \leq j} \{f(i, k) + f(k+1, j) + a[i] * a[k+1] * a[t+1]\} & i < j \end{cases}$$

最后依次类推，递归计算到不可再分的子问题为止，由于会重复计算子问题，所以采用备忘录的方式 $\text{cost}[i][j]$ 数组来记录 $f(i, j)$ 的值，最后返回 $\text{cost}[0][n]$ 为整体问题的最优解。

```
#include <bits/stdc++.h>
#define MAX 20+5
using namespace std;

int cost[MAX][MAX], path[MAX][MAX];

int matrixChain(int *a, int n) {
    for (int k = 1; k < n; ++k) {
        for (int i = 0; i < n - k; ++i) {
            int t = i + k; // 设置循环重点
            for (int j = i; j <= t; ++j) {
                int c = cost[i][j] + cost[j+1][t] + a[i]*a[j+1]*a[t+1]; // 计算 f(i, j)
                if(cost[i][t]==0||c<cost[i][t]) { // 更新最优解
                    cost[i][t] = c;
                    path[i][t] = j; // 更新路径
                }
            }
        }
    }
    return cost[0][n-1];
}

// 写出运算的表达式
void traceback(int n, int i, int j){ // 构造最优解
    if(i==j) cout<<"A"<<i+1;
    else{
        int k = path[i][j];
        cout<<"(";
        traceback(n, i, k);
        traceback(n, k+1, j);
        cout<<")";
    }
}
}
```

```
输入:
6
30 35 15 5 10 20 25
输出:
15125
((A1(A2A3))((A4A5)A6))
```

递归形式：

```
int MatrixChain(int *arr, int len) {
    if(len==1) return arr[0]*arr[1];
    int MatrixMultiplication(int *arr, int **res, int first, int last);
    // 创建记录数组防止重复运算子问题
    int **res = (int **) malloc(sizeof(int *) * len);
```

```

for (int i = 0; i < len; ++i) {
    res[i] = (int *) malloc(sizeof(int) * len);
    memset(res[i], 0, sizeof(int) * len);
}
return MatrixMultiplication(arr, res, 0, len-1);
}

int MatrixMultiplication(int *arr, int **res, int first, int last) {
    if(last<=first) return 0;
    if(res[first][last]!=0) return res[first][last]; //子问题重叠问题
    int minimum = -1;
    for (int mid = first; mid < last; ++mid) {
        res[first][mid] = MatrixMultiplication(arr, res, first, mid); //分为子问题
        res[mid+1][last] = MatrixMultiplication(arr, res, mid+1, last); //分为子问题
        int sum = res[first][mid] + res[mid+1][last] + arr[first]*arr[mid+1]*arr[last+1]; //递推公式
        if(minimum== -1 || sum < minimum) minimum = sum;
    }
    return minimum;
}

```

最长上升子序列

线性DP

<https://leetcode-cn.com/problems/longest-increasing-subsequence/>

思想: $dp[i]$ 表示以 $a[i]$ 为结尾的 $seq[0 : i - 1]$ 的最长上升子序列。 $T=O(n^2)$

$dp[i] = \max(dp[i], dp[j] + 1), j < i$, 最长长度就是 $LIS = \min\{dp[i]\}, 0 \leq i < n$, 和平常的dp有所不同

```

int lengthOfLIS(vector<int>& nums) {
    int len = nums.size(), *dp = new int[len];
    if(len<=1) return len;
    int fmax = 1;
    for (int i = 0; i < len; ++i) {
        dp[i] = 1;
        for (int j = 0; j < i; ++j)
            if(nums[j]<nums[i]&&dp[j]>=dp[i])
                dp[i] = dp[j]+1;
        if(dp[i]>fmax) fmax=dp[i];
    }
    return fmax;
}

```

进阶版:

思想: $T=O(n\log n)$ 虽然已经不属于DP了

Description

Given a set of n integers: $A=\{a_1, a_2, \dots, a_n\}$, we define a function $d(A)$ as below:

$$d(A) = \max_{1 \leq s_1 \leq t_1 < s_2 \leq t_2 \leq n} \left\{ \sum_{i=s_1}^{t_1} a_i + \sum_{j=s_2}^{t_2} a_j \right\}$$

Your task is to calculate $d(A)$.

Input

The input consists of $T(\leq 30)$ test cases. The number of test cases (T) is given in the first line of the input.

Each test case contains two lines. The first line is an integer $n(2 \leq n \leq 50000)$. The second line contains n integers: a_1, a_2, \dots, a_n . ($|a_i| \leq 10000$). There is an empty line after each case.

Output

Print exactly one line for each test case. The line should contain the integer $d(A)$.

Sample Input

```
1
10
1 -1 2 2 3 -3 4 -4 5 -5
```

Sample Output

```
13
```

元素越小，后面可以排的数字越多。

```
#include <bits/stdc++.h>
#define N 1000+5
using namespace std;

int a[N], dp[N];

void binChange(int *a, int n, int t){ //二分更新
    int low = 0, high = n - 1, mid;
    while(low < high){
        mid = (low + high) / 2;
        if(a[mid] >= t) high = mid;
        else low = mid + 1;
    }
    a[high] = t;
}

//这里不同与普通的二分查找，由于数组a中不存在全部小于t的值，且二分查找是找准确的值，这里是找一个模糊的值，因此保证l指向的位置是大于t的一个数值，并且根据二分，找到的是大于t的最小数值

int main(){
    int n; cin >> n;
    for (int i = 0; i < n; ++i) cin >> a[i];
    int len = 1;
    dp[0] = a[0];
```

```

for(int i=1;i<n;i++) {
    if(a[i]>dp[len-1]) dp[len++]=a[i];
    else if(a[i]<dp[len-1]) binChange(dp, len, a[i]);
}
if(n!=0) cout<<len<<endl;
else cout<<0<<endl;
return 0;
}

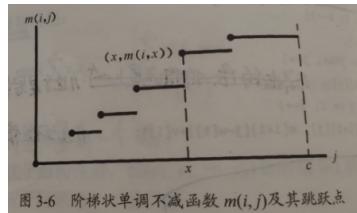
```

最长公共子序列

POJ2250 <http://poj.org/problem?id=2250>

思路：设字符串 $p[0 : m], q[0 : n]$ 的有最优解公共子序列 k , 那么在任意 $i \leq m, j \leq n$ 中两字符串的子序列 $p[0 : i], q[0 : j]$ 也存在解 s ; 假设在子序列 $p[0 : i], q[0 : j]$ 存在一个更优的解 s' , 那么可以使用新解 s' 替换掉原来的解 s , 得到的新的公共子序列 k' 就比原问题的公共 k 长度更长, 与问题假设的最优化相悖。因此该问题有最优子结构性质。根据问题写出DP方程如下：

$$LCS(i, j) = \begin{cases} LCS(i - 1, j - 1) + 1, & p[i] = q[j] \\ \max\{LCS(i - 1, j), LCS(i, j - 1)\}, & p[i] \neq q[j] \end{cases}$$



最优子结构性质：

设序列 $X = x_1, x_2, \dots, x_m$ 和 $Y = y_1, y_2, \dots, y_n$ 的最长公共子序列为 $Z = z_1, z_2, \dots, z_k$, 则

- (1) 若 $x_m = y_n$, 则 $z_k = x_m = y_n$, 且 z_{k-1} 是 x_{m-1} 和 y_{n-1} 的最长公共子序列。
- (2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$, 则 Z 是 x_{m-1} 和 Y 的最长公共子序列。
- (3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$, 则 Z 是 X 和 y_{n-1} 的最长公共子序列。

证明： (1) 用反证法。若 $z_k \neq x_m$, 则 $\{z_1, z_2, \dots, z_k, x_m\}$ 是 X 和 Y 的长度为 $k+1$ 的公共子序列。这与 Z 是 X 和 Y 的一个最长公共子序列矛盾。因此, 必有 $z_k = x_m = y_k$ 。由此可知 z_{k-1} 是 x_{m-1} 和 y_{n-1} 的一个长度为 $k-1$ 的公共子序列。若 x_{m-1} 和 y_{n-1} 有一个长度大于 $k-1$ 的公共子序列 W , 则将 x_m 加在其尾部产生 X 和 Y 的一个长度大于 k 的公共子序列。此为矛盾。故 z_{k-1} 是 x_{m-1} 和 y_{n-1} 的一个最长公共子序列。

(2) 由于 $z_k \neq x_m$, Z 是 x_{m-1} 和 Y 的一个公共子序列。若 x_{m-1} 和 Y 有一个长度大于 k 的公共子序列 W , 则 W 也是 X 和 Y 的一个长度大于 k 的公共子序列。这与 Z 是 X 和 Y 的一个最长公共子序列矛盾。由此即知, Z 是 X_{m-1} 和 Y 的一个最长公共子序列

(3) 证明与 (2) 类似。

上述性质告诉我们, 两个序列的最长公共子序列包含了这两个序列的前缀的最长公共子序列。因此, 最长公共子序列问题具有最有子结构性质

单纯计算长度：

```

int longestCommonSubsequence(string text1, string text2) {
    int m=text1.length(), n = text2.length();
    vector<vector<int>> record(m+1, vector<int>(n+1, 0));
    for(int i=0;i<m;i++) {
        for(int j=0;j<n;j++) {
            if(text1[i]==text2[j]) record[i+1][j+1]=record[i][j] + 1;
            else record[i+1][j+1]=max(record[i][j+1], record[i+1][j]);
        }
    }
    return record[m][n];
}

```

改进空间复杂度版：

```

int longestCommonSubsequence(string text1, string text2) {
    int m=text1.length(), n = text2.length();
    vector<int> record(n+1, 0);
    for(int i=0;i<m;i++) {
        int last = 0; //相当于record[i-1][j-1]
        for(int j=0;j<n;j++) {
            int tmp = record[j+1]; // 相当于记录record[i][j-1]
            if(text1[i]==text2[j]) record[j+1]=last + 1;
            else record[j+1] = max(tmp, record[j]); //record[j]相当于record[i-1][j]
            last = tmp;
        }
    }
    return record[n];
}

```

求LCS构造最优解：

```

string getLCS(string a, string b, vector<vector<int>> map) {
    int m = a.length(), n=b.length();
    string ret = "";
    while(m>0&&n>0) {
        if(a[m-1]==b[n-1]) {
            ret = a[--m] + ret;
            n--;
        } else if(map[m][n]==map[m-1][n]) m--;
        else n--;
    }
    return ret;
}

```

最大子段和

最大子段：

```

int maxSubArray(vector<int>& nums) {
    int p = 0, m=nums[0];
    for(int i:nums) {
        p=p+i>p+i:i;
        if(p>m) m=p;
    }
    return m;
}

```

最大字段和（升级版）

<http://poj.org/problem?id=2479>

改进版：

根据上升子序列的特性，设置一个辅助数组 d[] 记录最长上升子序列，

d[] 表示最长上升序列的元素，len 表示最长上升序列的长度；

算法步骤：

- 1) 初始化: d[1]=a[1], len=1;
 - 2) 依次判断序列中的每个元素，将 a[i] 与 d[len] (d[] 的最后一个元素) 比较；
 - 3) 如果 a[i]==d[len]，什么也不做，继续下一次循环；
 - 4) 如果 a[i]>d[len]，则将 a[i] 直接添加到 d[] 的末尾，即 d[++len]=a[i];
 - 5) 如果 a[i]<d[len]，则将 a[i] 覆盖 d[] 中第一个大于 a[i] 的数；在 d[] 中查找第一个大于 a[i] 的数可以采用二分查找 (d[] 本身有序)，也可以直接调用 upper_bound() 函数，该函数也是采用二分查找实现的，每次查找的时间复杂度为 $O(\log n)$ 。
- *upper_bound(d+1,d+len+1,a[i])=a[i];

分成序列两段分别求两者的大字段和。

```

const int MAXN = 50010;
int s[MAXN], lt[MAXN], rt[MAXN];

```

```

int main() {
    int n;
    cin>>n;
    while(n--) {
        int nums;
        scanf("%d", &nums);
        for (int i = 0; i < nums; ++i) scanf("%d", &s[i]);
        lt[0] = s[0], rt[nums-1] = s[nums-1];
        int ret = 0x80000000;
        for (int j = 1; j < nums; ++j) {
            lt[j] = max(lt[j-1]+s[j], s[j]); //从左到右求 s[0:i] 的字段和
            rt[nums-j-1] = max(rt[nums-j]+s[nums-j-1], s[nums-j-1]); //从右到左求 s[i:n] 的字段和
        }
        for (int k = 1; k < nums; ++k) {
            lt[k] = max(lt[k], lt[k-1]); //合并称为 s[0:i] 的最大字段和
            ret = max(ret, lt[k-1]+rt[k]); //求 s[0:n] 的最大字段和
        }
        cout<<ret<<endl;
    }
    return 0;
}

```

凸多边形最优三角剖分

(与矩阵连乘问题类似) 时间 $O(n^3)$ 空间 $O(n^2)$

(1) 最优子结构性质：若凸($n+1$)边形 $P = v_0, v_1, \dots, v_{n-1}$ 的最优三角剖分 T 包含三角形 v_0, v_k, v_n , $1 \leq k \leq n-1$, 则 T 的权为 3 个部分权的和：三角形 v_0, v_k, v_n 的权，子多边形 v_0, v_1, \dots, v_{k-1} 和 v_k, v_{k+1}, \dots, v_n 的权之和。可以断言，由 T 所确定的这 2 个子多边形的三角剖分也是最优的。因为若有 v_0, v_1, \dots, v_{k-1} 和 v_k, v_{k+1}, \dots, v_n 的更小权的三角剖分将导致 T 不是最优三角剖分的矛盾。

(2) 递归结构： $t[i][j]$, $1 \leq i < j \leq n$

流水调度作业

$T(S, t)$ 表示机器开始加工 S 中的作业时候，还有 t 时间需要等待：

$$T(S, t) = \min_{i \in S} \{a_i + T(S - \{i\}, b_i + \max\{t - a_i, 0\})\}$$

其中 $T(S - \{i\}, b_i + \max\{t - a_i, 0\})$ 表示作业中假如 a_i 之后，还需要再等待的时间为 $b_i + \max\{t - a_i, 0\}$ ，由原先还剩下的 t 时间再减去 a_i 即可。

Johnson 调度法则：

	i	$i + 1$
A	a_i	a_{i+1}
B	b_i	b_{i+1}

需要满足 Johnson 不等式：作业 i 和作业 $i + 1$ 之间需要满足以下关系： $\min\{b_i, a_{i+1}\} \geq \min\{a_i, b_{i+1}\}$ ，可以保证调度为最优。换句话说，当作业 i 和作业 $i + 1$ 之间不满足 Johnson 不等式的时候，交换两者的工作顺序之后，作业 i 和作业 $i + 1$ 之间满足 Johnson 不等式，那么得到的最新工作调度不会增加工时。（具体证明见书上，挺晦涩的，这里直接记住结论）

算法流程：

1. 令 $N_1 = \{i \mid a_i < b_i\}$, $N_2 = \{i \mid a_i \geq b_i\}$
2. 将 N_1 中按照非减序排序，将 N_2 中按照非增序排序。
3. N_1 中的作业接 N_2 中的作业即可满足 Johnson 法则的最优调度。

```

int FlowShop(int n, int* a, int* b, int* c) {
    class Jobtype{
        public:
            int operator <= (Jobtype a) const {return (key<=a.key);}
    };

```

```

int key, index;
bool job;
};

Jobtype *d = new Jobtype[n];
for(int i=0;i<n;i++) {
    d[i].key = min(a[i], b[i]);
    d[i].job = a[i] <= b[i];
    d[i].index = i;
}
sort(d,n); //这里程序将N1和N2进行合并排序，按照job来进行分开
int j = 0, k = n - 1;
for(int i=0;i<n;i++) {
    if(d[i].job)c[j++]=d[i].index; //N1中的数据进行正序放置
    else c[k--] = d[i].index; //N2中的数据进行逆序放置
}
j = a[c[0]];
k = j + b[c[0]];
for(int i=1;i<n;i++) {
    j+=a[c[i]];
    k = max(k + b[c[i]], j+b[c[i]]); //记录完成的最后时刻
}
return k;
}

```

0-1背包问题

可以看后面的背包问题专题，更加简单明了

难度：☆☆☆☆☆

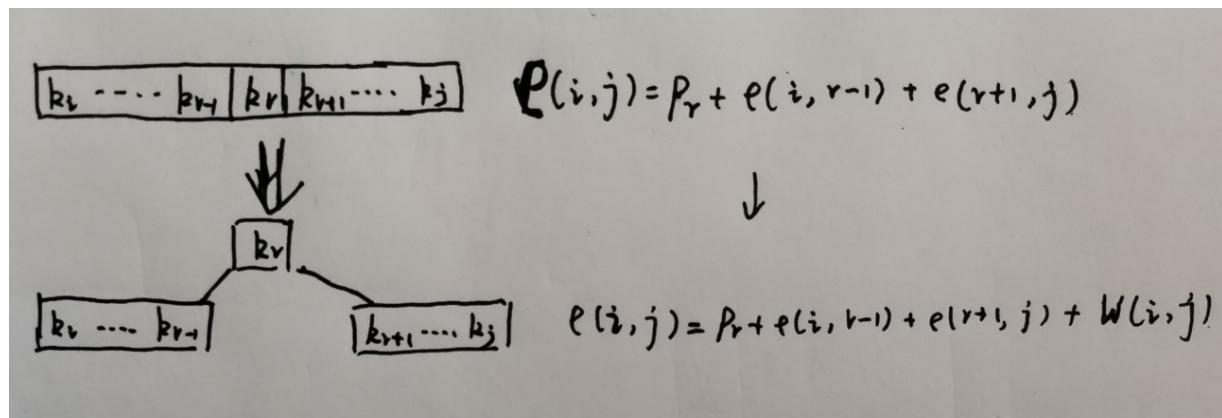
对于0-1背包问题就是一个选不选的问题，在不进行任何优化剪枝的情况下，时间复杂度有 $O(2^n)$ 规模，因此处理时间会很长。递推公式如下：

$m(i, j)$ 表示可用物品*i*, $i+1 \dots i+n$ 在剩余空间可以携带物品的最大价值。

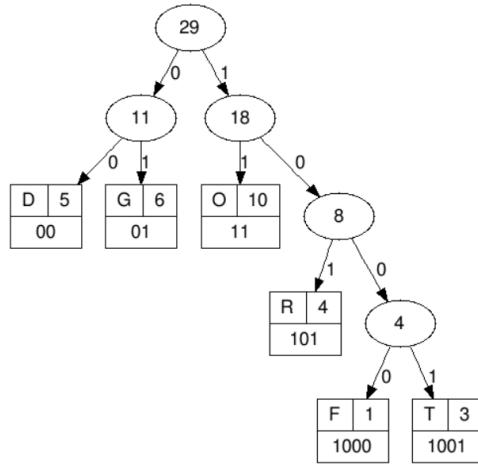
$$m(i, j) = \begin{cases} \max\{m(i+1, j), m(i+1, j - w_i) + v_i\} & j \geq w_i \\ m(i+1, j) & 0 < j < w_i \end{cases}$$

如何优化原始求解空间下的复杂度？

可以了解到，在物品 *i* 数量不变的情况下，*j* 越大， $m(i, j)$ 单调不减。（废话）



通过不同的 $m(i, j)$ 进行结合：



```

#include <iostream>
using namespace std;

int **newMetrix(int m, int n) {
    int **a = new int*[m];
    for (int i = 0; i < m; i++) a[i] = new int[n];
    return a;
}

int knapSack(int n, int c, int w[], int v[], int **p, int *x) {
    p[0][0] = 0;
    p[0][1] = 0;
    //这个是初始化跳跃点
    int *head = new int[n + 2];//第0个不用
    head[n] = 1;
    int left = 0, right = 0, next = 1; // 这是一个新创建的过程，把原来的数组又复制和新建一个
    //left表示每次p[i]的起点 right表示p[i]的结束点 next用于指向下一个p[i]的index
    //从最后一个物品开始拿取
    for (int i = n; i >= 1; i--) {
        int k = left;
        //这个循环是为了将p[i]中的数据和新增的点相加，并且判断是否是受控点等
        for (int j = left; j <= right; j++)
        {
            if (p[j][0] + w[i] > c) break;//如果大于C,继续判断下个物品
            int y = p[j][0] + w[i], m = p[j][1] + v[i];
            while (k <= right && p[k][0] < y) {
                //如果小于y的话，说明(y,m)点不是受控点，可以安全copy
                p[next][0] = p[k][0];
                p[next++][1] = p[k++][1];
            }
            while (k <= right && p[k][0] == y) {
                //到后期总结的时候会有很多y相同的值，这个也是用来消除受控点
                if (m < p[k][1]) m = p[k][1];
                k++;
            }
            // 小于和等于的情况都已经讨论过了，该讨论增加合并节点的情况了
            //这种情况下，要么是后面还有原来的节点没有被copy，要么就是k>right,后面没有节点了，所以我们判断(y,m)就可以了，不是受控点就加入
            //要比较的是跟上一个加入的节点
            while (m > p[next - 1][1]) {
                p[next][0] = y;
                p[next++][1] = m;
            }
            //如果添加新的点之后，还要看后面的点是不是受控点
            while (k <= right && p[k][1] <= p[next-1][1] )k++;
        }
        left = right + 1;
    }
}

```

```

        right = next - 1;
        head[i - 1] = next;
    }
    return p[next - 1][1];
}

int main()
{
    int c = 10, w[] = { 0, 2, 2, 6, 5, 4 }, v[] = { 0, 6, 3, 5, 4, 6 }, **p, *x;
    //c 是指 背包容量 w是指物品质量 v指物品价值 **p是用来保存p[i]和q[i]的数组 *x是记录的是那个物品被取
    //我们这里还需要一个数组，因为**p为了保证m(i, j)的单调不减性质，需要不停赋值和合并，所以我们需要的这个数组就是用来记录
    每个p[i]起始的index
    int n = sizeof(w) / 4 - 1;
    x = new int[n];
    p = newMatrix((n*2)*n*2, 2);
    cout<<knapSack(n, c, w, v, p, x);
    return 0;
}

```

最优搜索二叉树

大致类似矩阵链乘法，最重要的就是需要读懂题目想要干什么。

题目：在一个文章中，每个单词出现的频率不同。为了加快检索的目的，目标要求使使用频率越高的词越靠近树根位置，频率越小的词越远离树根。给定 n 个不同关键字的已排序的序列 $K = \langle k_1, k_2, \dots, k_n \rangle$, $k_1 < \dots < k_n$, 每个关键字都有一个 p_i 表示其搜索频率。对于 n 个关键字还有 $n + 1$ 个 d_0, d_1, \dots, d_n 表示不在表 K 中的值, d_{i-1} 表示小于 k_i 的值, d_i 表示大于 k_i 的值, 每个关键字都有一个 q_i 表示其搜索频率。并且有以下公式：

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$$

最优化结构的证明：假设已经有最优搜索二叉树 T , 包含一颗关键字为 $k_i \dots k_j$ 的子树 T' , 那么 T' 必是 $k_i \dots k_j$ 以及“伪结点” $d_{i-1}, d_i \dots d_j$ 的最优解, 如果存在一个也包含关键字为 $k_i \dots k_j$ 的子树 T'' , 且搜索期望比 T' 要低, 使用剪切——粘贴法替换 T' , 从而得到一个整体比 T 期望还要低的树, 与 T 最优的假设相矛盾。

最优二叉树不一定是最矮的。

如何分割子问题：

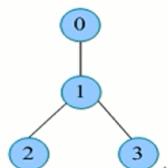
战略游戏

描述 (POJ1463)

鲍勃喜欢玩电脑游戏，特别是战略游戏，但有时他找不到足够快的解决方案，然后他非常伤心。现在他有以下问题。他必须保卫一座中世纪的城市，城市的道路形成一棵树。他必须把最小数量的士兵放在节点上，这样他们才能观察到所有的边缘。你能帮助他吗？

你的程序应该找到 bob 为一棵树放置的最小士兵数。

例如，对于树：



解决方案是一个士兵（在节点 1）。

```

double OBST(vector<double> p, vector<double> q, int n, vector<vector<int>> root) {
    vector<vector<double>> w(n+2, vector<double>(n+1, 0))           // w[i, j] 是从 i 到 j 增加的概率
                                                , e(n+2, vector<double>(n+1, -1));   // e[i, j] 是从 i 到 j 的期望
    root = vector<vector<int>>(n+2, vector<int>(n+1, -1));
    for (int i = 1; i <= n+1; ++i) {
        w[i][i-1] = q[i-1];
        e[i][i-1] = q[i-1];
    }
}

```

```

for(int gap=1;gap<=n;gap++) {
    for(int i=1;i<=n-gap+1;i++) {
        int j = i + gap - 1;
        w[i][j] = w[i][j - 1] + p[j] + q[j];
        for(int r = i; r <= j; r++) {
            double tmp = e[i][r - 1] + e[r + 1][j] + w[i][j];
            if(e[i][j] == -1 || tmp < e[i][j]) {
                e[i][j]=tmp;
                root[i][j]=r;
            }
        }
    }
    cout<<root[1][n]<<endl;
    return e[1][n];
}

```

找零问题

见递归与分治法中找零问题

数字组合问题

线性DP

【描述】

有n个正整数，找出其中和为t(t也是正整数)的可能的组合方式。如：

$n = 5$, 5个数分别为1, 2, 3, 4, 5, $t = 5$;

那么可能的组合有 $5 = 1 + 4$ 和 $5 = 2 + 3$ 和 $5 = 5$ 三种组合方式。

【输入】

输入的第一行是两个正整数n和t，用空格隔开，其中 $1 \leq n \leq 20$ ，表示正整数的个数，t为要求的和($1 \leq t \leq 1000$)

接下来的一行是n个正整数，用空格隔开。

【输出】

和为t的不同的组合方式的数目。

【样例输入】

```

5 5
1 2 3 4 5

```

【样例输出】

```
3
```

思路：使用动态规划，用 $dp[i][j]$ 来表示前 i 位数字可表示和为 j 的组合数。列的DP方程为
 $dp[i][j] = dp[i - 1][j] + dp[i - 1][j - a[i]]$ ，由于可以第 i 位完全取决于前 i-1 位的数字，因此可以缩减备忘录的空间由 $O(n^2)$ 至 $O(n)$ 。

代码：

```

int numericAssembly(int *a, int n, int res) {
    int *dp = new int[res];
    memset(dp, 0, (res+1)*4);
    dp[0]=1;
    for (int i = 0; i < n; ++i) {
        for (int j = res; j >= a[i]; j--) //由于是求res的组合数，后面的数字完全没必要去管
            dp[j] += dp[j-a[i]];
    }
    return dp[res];
}

```

```

int main() {
    int n, res;
    while(cin>>n) {
        cin>>res;
        int *a = new int[n];
        for(int i=0;i<n;i++)cin>>a[i];
        cout<<numericAssembly(a, n, res)<<endl;
    }
}

```

钢条切割

Serling公司购买长钢条，将其切割为短钢条出售。切割工序本身没有成本支出。公司管理层希望知道最佳的切割方案。假定我们知道Serling公司出售一段长为*i*英寸的钢条的价格为 p_i ($i=1,2,\dots$ ，单位为美元)。钢条的长度均为整英寸。图给出了一个价格表的样例。

长度 <i>i</i>	1	2	3	4	5	6	7	8	9	10
价格 p_i	1	5	8	9	10	17	17	20	24	30

图 15-1 钢条价格表样例。每段长度为 i 英寸的钢条为公司带来 p_i 美元的收益

$$dp[i] = \max(\text{price}[i - m] + dp[m], dp[i])$$

```

int ironSlice(int * p, int n) {
    int *dp = new int[n+1], *rt = new int[n+1];
    memset(rt, 0, (n+1)*4);
    for(int i=1;i<=n;i++) dp[i]=p[i-1];
    dp[0]=0;
    for(int i=1;i<n;i++) {
        for (int j = 0; j <= i; ++j) {
            if(dp[j] + p[i-j]>dp[i+1]) {
                dp[i+1] = dp[j] + p[i-j]; //dp
                rt[i+1] = j;           //构造最优解
            }
        }
        for(int i=1;i<=n;i++) cout<<rt[i]<<" ";
        cout<<endl;
        return dp[n];
    }
}

```

周年派对——树形DP

<http://poj.org/problem?id=2342>

Description

There is going to be a party to celebrate the 80-th Anniversary of the Ural State University. The University has a hierarchical structure of employees. It means that the supervisor relation forms a tree rooted at the rector V. E. Tretyakov. In order to make the party funny for every one, the rector does not want both an employee and his or her immediate supervisor to be present. The personnel office has evaluated conviviality of each employee, so everyone has some number (rating) attached to him or her. Your task is to make a list of guests with the maximal possible sum of guests' conviviality ratings.

Input

Employees are numbered from 1 to N. A first line of input contains a number N. $1 \leq N \leq 6\,000$. Each of the subsequent N lines contains the conviviality rating of the corresponding employee. Conviviality rating is an integer number in a range from -128 to 127. After that go $N - 1$ lines that describe a supervisor relation tree. Each line of the tree specification has the form:

L K

It means that the K-th employee is an immediate supervisor of the L-th employee. Input is ended with the

line
0 0

Output

Output should contain the maximal sum of guests' ratings.

Sample Input

```
7  
1  
1  
1  
1  
1  
1  
1  
1 3  
2 3  
6 4  
7 4  
4 5  
3 5  
0 0
```

Sample Output

```
5
```

$0 < N, V \leq 100$
 $0 < v_i, w_i, s_i \leq 100$

```
const int N = 6000 + 10;  
int val[N], fa[N], dp[N][2]; //记录结点值，父亲，dp大小  
vector<int> E[N]; //记录每个结点的儿子结点  
void dfs(int u) {  
    dp[u][0] = 0, dp[u][1] = val[u]; //边界条件  
    for (int i = 0; i < E[u].size(); ++i) {  
        int v = E[u][i];  
        dfs(v); //自下而上求解  
        dp[u][0] += max(dp[v][0], dp[v][1]);  
        dp[u][1] += dp[v][0];  
    }  
}  
  
int main() {  
    int n;  
    scanf("%d", &n);  
    for (int i=1; i<=n; i++) {  
        scanf("%d", &val[i]);  
        E[i].clear();  
        fa[i]=-1;  
    }  
    int f, s;  
    for (int i=0; i<n; i++) {  
        scanf("%d %d", &s, &f);  
        fa[s]=f;  
        E[f].push_back(s);  
    }  
    int rt = fa[1];  
    while (fa[rt] != -1) rt = fa[rt]; //找到根节点  
    dfs(rt);  
    printf("%d", max(dp[rt][0], dp[rt][1]));  
}
```

类似的题：

<http://poj.org/problem?id=1463>

数据范围

$0 < N \leq 1000$
 $0 < V \leq 2000$
 $0 < v_i, w_i, s_i \leq 2000$

```
#include<cstdio>
#include<vector>
#include<memory.h>
#include<algorithm>
using namespace std;
const int N = 1500 + 10;
int dp[N][2], fa[N];
vector<int> E[N];

void dfs(int u) {
    dp[u][0] = 0;
    dp[u][1] = 1;
    for(int i=0;i<E[u].size();i++) {
        int v = E[u][i];
        dfs(v);
        dp[u][0] += dp[v][1];
        dp[u][1] += min(dp[v][0], dp[v][1]); //几乎是相同的模板
    }
}

int main() {
    int n;
    while(~scanf("%d",&n)) {
        memset(fa, -1, N*4);
        for(int i=0;i<n;i++) {
            int node, nums;
            scanf("%d:(%d)", &node, &nums);
            E[node].clear();
            for(int j=0;j<nums;j++) {
                int son;
                scanf("%d", &son);
                E[node].push_back(son);
                fa[son] = node;
            }
        }
        int rt = 0;
        while(fa[rt] != -1) rt = fa[rt];
        dfs(rt);
        int s = min(dp[rt][0], dp[rt][1]);
        printf("%d\n", s);
    }
    return 0;
}
```

苹果树——树形背包问题

好难，三维DP

最小回文——区间DP

<http://poj.org/problem?id=3280>

Description

Keeping track of all the cows can be a tricky task so Farmer John has installed a system to automate it. He has installed on each cow an electronic ID tag that the system will read as the cows pass by a scanner. Each ID tag's contents are currently a single string with length M ($1 \leq M \leq 2,000$) characters drawn from an alphabet of N ($1 \leq N \leq 26$) different symbols (namely, the lower-case roman alphabet).

Cows, being the mischievous creatures they are, sometimes try to spoof the system by walking backwards. While a cow whose ID is "abcba" would read the same no matter which direction she walks, a cow with the ID "abcb" can potentially register as two different IDs ("abcb" and "bcba").

FJ would like to change the cow's ID tags so they read the same no matter which direction the cow walks by. For example, "abcb" can be changed by adding "a" at the end to form "abcba" so that the ID is palindromic (reads the same forwards and backwards). Some other ways to change the ID to be palindromic are to include adding the three letters "bcb" to the beginning to yield the ID "bcbabcb" or removing the letter "a" to yield the ID "bcb". One can add or remove characters at any location in the string yielding a string longer or shorter than the original string.

Unfortunately as the ID tags are electronic, each character insertion or deletion has a cost ($0 \leq \text{cost} \leq 10,000$) which varies depending on exactly which character value to be added or deleted. Given the content of a cow's ID tag and the cost of inserting or deleting each of the alphabet's characters, find the minimum cost to change the ID tag so it satisfies FJ's requirements. An empty ID tag is considered to satisfy the requirements of reading the same forward and backward. Only letters with associated costs can be added to a string.

Input

Line 1: Two space-separated integers: N and M

Line 2: This line contains exactly M characters which constitute the initial ID string

Lines 3.. $N+2$: Each line contains three space-separated entities: a character of the input alphabet and two integers which are respectively the cost of adding and deleting that character.

Output

Line 1: A single line with a single integer that is the minimum cost to change the given name tag.

Sample Input

```
3 4
abcb
a 1000 1100
b 350 700
c 200 800
```

Sample Output

```
900
```

Code:

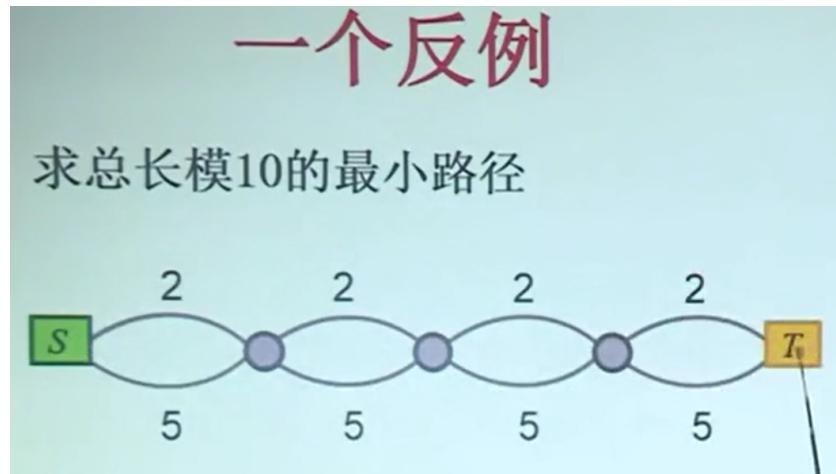
```
#define N 2010
char s[N], t;
int abt[26], a, b, dp[N][N];
int main() {
    int m, n;
    scanf("%d%d", &m, &n);
    memset(s, 0, N);
    memset(abt, 0, 26*4);
    scanf("%s", s);
    for(int i=0; i<m; i++) {
        scanf("\n%c", &t);
        scanf("%d%d", &a, &b);
        abt[t-'a']=min(a, b);
    }
    for (int j = n - 1; j >= 0; --j) {
        for (int i = j+1; i < n; ++i) {
            if(s[j]==s[i]) dp[j][i]=dp[j+1][i-1];
            else dp[j][i] = min(dp[j+1][i]+abt[s[j]-'a'], dp[j][i-1]+abt[s[i]-'a']);
        }
    }
}
```

```

    }
    printf("%d\n", dp[0][n-1]);
    return 0;
}

```

什么情况不能使用动态规划



原因：不能满足优化原则，不能使用动态规划；破坏了优化原则，它的问题和其的子问题之间，没有依赖关系，即全局的解放到子问题中，不能满足子问题最优（通过预先查看来确定？）。

课后习题

3-1 独立任务最优调度 难度：☆☆☆☆

- 计算 $m = \max\{\max\{a_i\}, \max\{b_i\}\}$
- 设置布尔量 $p(i, j, k)$ 为前 k 个作业再使用处理机 A 时候是否可以在不超过 i 时间且在处理机不超过 j 时间内的完成。

$$p(i, j, k) = p(i - a_k, j, k - 1) | p(i, j - b_k, k - 1)$$
- 最后计算：

$$\min_{0 \leq i, j \leq mn, p(i, j, k) = \text{TRUE}} \{\max(i, j)\}$$

```

//优点类似双重背包问题
#include<iostream>
using namespace std;
const int N = 205;
int n, a[N], b[N], m=0, ***dp;
void work() {
    for(int i=0; i<m; i++) { //设置边界条件
        for(int j=0; j<m; j++) {
            dp[i][j][0] = true;
            for(int k=1; k<n; k++) dp[i][j][k] = false;
        }
    }

    for(int k=1; k<n; k++) {
        for(int i=0; i<m; i++) {
            for(int j=0; j<m; j++) {
                if(i>=a[k-1]) p[i][j][k] = p[i-a[k-1]][j][k-1];
                if(j>=b[k-1]) p[i][j][k] = p[i][j-b[k-1]][k-1];
            }
        }
    }
    int opt = m;

    for (int i = 0; i <= m; ++i) {
        for (int j = 0; j <= m; ++j) {
            if(dp[i][j][n]) opt = min (opt, max(i, j));
        }
    }
}

```

```

        }
    }

    cout<<opt;
}

int main() {
    cin>>n;
    for(int i=0;i<n;i++) {
        cin>>a[i];
        m=max(m,a[i]);
    }
    for(int i=0;i<n;i++) {
        cin>>b[i];
        m=max(m,b[i]);
    }
    m = m*n;
    dp = new3DArray(m+1,m+1,n);
    work();
}

```

3-3 环形石子合并——区间DP 难度: ☆☆☆

<http://acm.hdu.edu.cn/showproblem.php?pid=3506>

<https://acm.sdu.edu.cn/onlinejudge3/solutions/8200445>

可以通过转换将环形转换为直线型: 如 $a_1, a_2, a_3 \dots a_n$ 可以变为直线型 $a_1, a_2, a_3, \dots, a_n, a_1, a_2 \dots a_{n-1}$, 将原来长度为n的环形, 转换为长度为 $2n - 1$ 的直线型。

- 环形转直线
- 四边不等式优化

四边形优化?

```

//环形石子合并——典型区间DP
#include <bits/stdc++.h>

using namespace std;
const int N = 220;
int data[N];
int dp[N][N];
int dp2[N][N];
int s[N];
int merge(int *a, int n){ //O(n^4)——需要通过平行四边形法则优化为成O(n^3)
    for(int low = 1; low <= n; low++) {
        for(int gap=1;gap<n;gap++) {
            for(int i=low;i<n-low-gap;i++) {
                int k = i + gap;
                for(int j = i + 1; j<=k; j++) {
                    dp[i][k] = max(dp[i][k], dp[i][j-1]+dp[j][k]+s[k]-s[i-1]);
                    if(dp2[i][k]==0) dp2[i][k] = dp2[i][j-1]+dp2[j][k]+s[k]-s[i-1];
                    else dp2[i][k] = min(dp2[i][k], dp2[i][j-1]+dp2[j][k]+s[k]-s[i-1]);
                }
            }
        }
    }
    int _m = 0, _n = 0x3fff3f3f;
    for(int i=1;i<n;i++) {
        _m = max(dp[i][i+n-1], _m);
        _n = min(dp2[i][i+n-1], _n);
    }
    cout<<_n<<endl;
    return _m;
}

```

```

int main() {
    int n;
    cin>>n;
    for (int i = 1; i <= n; ++i) {
        cin>>data[i];
        data[n+i] = data[i];
    }
    for(int i = 1;i <= 2*n; ++i){
        s[i]=s[i-1]+data[i];
    }
    cout<<merge(data,n);
}

```

3-4数字三角形:

优化内存版: $a[i] = \max(a[i], a[i - 1]) + b[i]$

```

#include <bits/stdc++.h>

using namespace std;
const int N = 110;
int a[N], b[N];
int main() {
    int n;
    cin>>n;
    for(int i=1;i<=n;i++) {
        if(i==1)cin>>a[0];
        else{
            for(int j=0;j<i;j++)cin>>b[j];
            for(int j=i-1;j>0;j--)a[j]=max(a[j],a[j-1])+b[j];
            a[0]+=b[0];
        }
    }
    int m=0;
    for(int i=0;i<n;i++)m=max(m,a[i]);
    cout<<m<<endl;
}

```

3-6 租用游艇问题: 难度: ☆

$$\begin{aligned} 0 < N &\leq 1000 \\ 0 < V &\leq 20000 \\ 0 < v_i, w_i, s_i &\leq 20000 \end{aligned}$$

$\text{cost}(i) = \min\{\text{cost}(i), \text{cost}(j) + a[j][i]\}, j \in [2, i]$

```

#include <bits/stdc++.h>

using namespace std;
const int N = 210;
int a[N][N];
int rentBoat(int n) {
    int *cost = new int[n+1];
    memset(cost, 0, (n+1)*4);
    cost[2] = a[1][2];
    for(int i=3;i<=n;i++) {
        int t = a[1][i];
        for (int j = 2; j < i; ++j)
            t = min(t, cost[j] + a[j][i]);
        cost[i]=t;
    }
    return cost[n];
}

int main() {

```

```

int n;
cin>>n;
for(int i=1;i<n;i++)
    for(int j=i+1;j<=n;j++) cin>>a[i][j];
cout<<rentBoat(n);
}

```

贪心算法

贪心算法是指，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。贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备的特点是：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。

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，也是贪心算法与动态规划算法的主要区别。动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

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。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

局部最优->整体最优

最优装载问题

要使用贪心算法解决问题，我们必须先证明：（1）该问题具备贪心选择性质；（2）具备最优子结构性质。

首先先证明贪心选择性质：设集装箱已依其重量从大到小排序， (x_1, x_2, \dots, x_n) 是最优装载问题的一个最优解。又设 $k = \min\{i | x_i = 1\} \{1 \leq i \leq n\}$. 易知，如果给定的最优装载问题有解，则 $1 \leq k \leq n$;

证明：

当 $k=1$ 时， (x_1, x_2, \dots, x_n) 是满足贪心选择性质的最优解。

当 $k > 1$ 时，取 $y_1 = 1, y_k = 0, y_i = x_i, 1 < i \leq n, i \neq k$ ，则 $\sum_1^n w_i y_i = w_1 - w_k + \sum_1^n w_i x_i \leq \sum_1^n w_i x_i \leq c$ ，因此 (y_1, y_2, \dots, y_n) 是所给最优装载问题的可行解，又因为 $\sum_1^n y_i = \sum_1^n x_i$ ， (y_1, y_2, \dots, y_n) 是满足贪心性质的最优解，所以最优装载问题具有贪心选择性质。

其次，证明该问题具备最优子结构性质：设 (x_1, x_2, \dots, x_n) 是最优装载的满足贪心选择性质的最优解，易知， $x_1=1$ ， (x_2, x_3, \dots, x_n) 是轮船载重量为 $c - w_1$ ，待装船集装箱为 $\{2, 3, \dots, n\}$ 时相应的最优装载问题的最优解。

活动安排问题

难度：☆

目的就是尽可能多安排活动。开始时间即为 S_i ，结束时间记为 F_i 。

将各个结束时间进行排序，一个活动完成紧接着下一个相邻的活动即 $S_j \geq F_i$ ($i < j$)。

```

int cmp(int *a, int *b) {return a[1]<b[1];}

int greedSelect(int n, int **t) {
    sort(t, t+n, cmp); //按照结束时间进行排序
    int now = 0, sum = 0;
    for (int i = 1; i < n; ++i)
        if (t[i][0] > t[now][1]) { //开始时间是否在结束时间之后
            sum += t[i][1] - t[i][0];
            now = i;
        }
    return sum;
}

```

哈夫曼编码

难度：☆☆

注意：小的值放在左子树，大的值放在右子树

问题描述：长江游艇俱乐部在长江上设置了 n 个游艇出租站 $1, 2, \dots, n$ 。游客可在这些游艇出租站租用游艇，并在下游的任何一个游艇出租站归还游艇。游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i, j)$ ($1 \leq i < j \leq n$)。试设计一个算法，计算出从游艇出租站 1 到游艇出租站 n 所需的最少租金。

算法设计：对于给定的游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i, j)$ ($1 \leq i < j \leq n$)，计算从游艇出租站 1 到游艇出租站 n 所需的最少租金。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 n ($n \leq 200$)，表示有 n 个游艇出租站。接下来的 $n-1$ 行是 $r(i, j)$ ($1 \leq i < j \leq n$)。

结果输出：将算出的从游艇出租站 1 到游艇出租站 n 所需的最少租金输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	12
5 15	
7	

```
struct Node {  
    int num = 0;  
    Node *left = NULL, *right = NULL;  
};  
  
bool comp(Node*a, Node*b) {return a->num>b->num;} //堆和正常情况下是相反的，小顶堆用大于号  
Node* Huffman(vector<int> sequence) { //字符数量的序列  
    vector<Node*> heap;  
    for(int i=0;i<sequence.size();i++) {  
        Node *t = new Node;  
        t->num=sequence[i];  
        heap.push_back(t);  
    }  
    while(heap.size() != 1) {  
        make_heap(heap.begin(), heap.end(), comp);  
        Node* a = heap.front();  
        pop_heap(heap.begin(), heap.end(), comp);heap.pop_back();  
        Node* b = heap.front();  
        pop_heap(heap.begin(), heap.end(), comp);heap.pop_back();  
        Node *c = new Node;  
        c->left = a;c->right=b; //小的在左子树，大的在右子树  
        c->num = a->num + b->num;  
        heap.push_back(c);push_heap(heap.begin(), heap.end());  
    }  
    return heap[0];  
}  
  
void showHuffman(Node* head, string code = "") {  
    if(head==NULL) return;  
    if(head->left==NULL&&head->right==NULL)  
        cout<<head->num<<": "<< code<<endl;  
    showHuffman(head->left, code+"0");  
    showHuffman(head->right, code+"1");  
}
```

单源最短路径

见图论中的dijkstra算法。[链接](#)

最小生成树

见图论中的最小生成树算法 (Prim+Kruskal) 。[链接](#)

多机调度

排序+小顶堆 难度: ☆

¶ 注意, 这个是求解的近似解最优解, 真正的解还得看回溯

C++里面堆可以用priority_queue<>()

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

void Greedy(int a[], int n, int m){ //n表示作业的数量, m表示处理机的数量
    if(n<m){
        cout<<"为每个作业分配一个处理机即可";
        return;
    }
    sort(a, n); //排序
    MinHeap<CPU> S(m); //m个处理机, 初始工作时间为0
    for(int i=n-1;i>=0;i--){
        H.delete(x); //取出时间分配最少的处理机
        x.time += a[i];
        H.insert(x); //重新加入小顶堆
    }
}
```

课后习题

4-1 会场安排问题 难度: ☆

想让会场最少, 那就让每个会场安排数量最多。数据结构可以使用小顶堆或者排序即可。O(nlogn) 取决于排序算法的效率。

```
struct activity{
    int start=0, end=0;
};

bool cmp(activity a, activity b){return a.start<b.start;}
int arrangement(vector<activity> grp){
    vector<int> nums(1, 0);
    sort(grp.begin(), grp.end(), cmp);
    nums[0]=grp[0].end;
    for(int i=1;i<grp.size();i++){
        bool ok = false;
        for(int j=0;j<nums.size();j++){
            if(grp[i].start>nums[j]){
                ok=true;
                nums[j]=grp[i].end;
                break;
            }
        }
        if(!ok) nums.push_back(grp[i].end);
    }
    return nums.size();
}
```

4-2 最优合并问题 难度: ☆

```
void merge(vector<int> seq, int &minimum, int & maximum){
    minimum=0, maximum=0;
    sort(seq.begin(), seq.end());
    int n = seq.size();
    vector<int> tmp =seq;
```

```

        for(int i=0;i<n-1;i++) {
            minimum+=tmp[i]+tmp[i+1]-1;
            tmp[i+1]+=tmp[i];
        }
        tmp = seq;
        for(int i=n-1;i>0;i--) {
            maximum+=tmp[i]+tmp[i-1]-1;
            tmp[i-1]+=tmp[i];
        }
    }
}

```

4-3 磁带最优存储 难度: ☆

4-7 多处最优服务次序问题 难度: ☆

O(mn)

```

double multiService(vector<int> line, int n) {
    sort(line.begin(),line.end());
    vector<vector<int>> time(n,vector<int>(2,0));
    for (int i = 0; i < line.size(); ++i) {
        int min_index = 0,min_value = time[0][time[0].size()-1];
        for (int j = 1; j < n; ++j) {
            if(time[j][time[j].size()-1]<min_value) {
                min_value = time[j][time[j].size()-1];
                min_index = j;
            }
        }
        int cost = time[min_index][time[min_index].size()-1]+line[i];
        time[min_index].push_back(cost);
        time[min_index][0]+=cost;
    }
    int all = 0;
    for (int k = 0; k < n; ++k) {
        all+=time[k][0];
    }
    return all*1.0/line.size();
}

```

4-9 汽车加油问题 难度: ☆

```

int addOil(vector<int> list, int n) {
    int gas = n,cnt=0;
    for (int i = 0; i < list.size(); ++i) {
        if(gas-list[i]<0) {
            cnt++;
            gas=n-list[i];
            if(gas<0) {
                cout<<"No Solution"<<endl;
                return -1;
            }
        } else gas-=list[i];
    }
    return cnt;
}

```

4-10 区间覆盖问题 难度: ☆

```

int gapCover(vector<int> l, int n) {
    int i=l[0], j=l[0];
    for (int k = 1; k < l.size(); ++k) {
        if(l[k]<i) i=l[k];
        if(l[k]>j) j=l[k];
    }
    return int(ceil(1.0*(j-i)/n));
}

```

书上属于睿智解法O(nlogn)。我的正常解法：O(n)

4-11 删数问题 难度：☆☆

我的：删除 k 位就是保留 $n - k$ 位，先找第 i 位合适的数字（还需要找 $n - k - i$ 位的数字）就要在第 $k + i$ 之前找最小值，因为在 k 位之后剩余的数量就 $< n - k - i$ 位，不满足要求。

```

int deletion(vector<int> a, int k) {
    int start = 0, n = a.size(), last = k;
    vector<int> nums;
    while(start<=last&&last<n) {
        int min_value = a[start], min_index = start;
        for (int i = start; i <= last; ++i)
            if(a[i] < min_value) {
                min_value=a[i];
                min_index=i;
            }
        start = min_index;
        nums.push_back(a[start++]);
        last++;
    }
    int s = 0;
    for(;s<nums.size()&&nums.size() != 1;s++) if(nums[s] != 0) break;
    for (int j = s; j < nums.size(); ++j) cout<<nums[j];
    cout<<endl;
    return 0;
}

```

例题做法：

逆序删除法：例如178256，第一个逆序 82，删除8，得到17256；第二个逆序72，删除7，得到256；如果没有逆序并且没有删除够相应数量的元素，则删除最后一个元素。

4-15 最优分解问题

使用数组解决重复子问题。

```

int f(int n, int *a) {
    if(a[n]==0) {
        int m = -1;
        for(int i=n-1;i>=n/2;i--) {
            int res = i*f(n-i,a);
            if(res>m)m=res;
        }
        a[n]=m;
    }
    return a[n];
}

int bestDivision(int n) {
    if(n<=4) return n;
    int * a = new int[n+1];
    memset(a, 0, n*4+4);
    for(int i=0;i<=4;i++) a[i]=i;
    return f(n, a);
}

```

回溯法

找到解空间满足约束条件的所有解

子集树和排列树

Perm

模板代码

```
void permutation(int *a, int low, int high) {
    if(low==high) {
        for (int i = 0; i <= high; ++i) printf("%d ", a[i]);
        printf("\n");
    } else{
        for (int i = low; i <= high; ++i) {
            swap(&a[low], &a[i]);
            permutation(a, low +1, high);
            swap(&a[low], &a[i]); //再换回来
        }
    }
}
```

最优装载(变形)

一种特殊的0-1背包问题，可以把轮船1当成一个背包，0就是选轮船1，1就是选轮船2。dp[v-w]+w 难度：☆☆

【问题描述】 有 n 个集装箱要装进2个重量分别为 C_1 和 C_2 的轮船，其中集装箱 i 的重量是 w_i ，且 $\sum_{i=1}^n w_i \leq C_1 + C_2$ 。装载问题要求确定是否有一个方案能够将 n 个集装箱都装上两艘轮船。如果有，找出这种装载方案。

【问题分析】 将第一艘船尽可能装满，再将剩余的装入到第二个船。

代码：

最优装载有两处剪枝：1. 累积货物+当前货物超出当前轮船最大装载量时 2. 剩余货物不足以大于最优解时 (A*的思想)

递归法：

```
class loading{
public:
    loading(int n, int c, int *w);
    int n, c, best, cw, r
    // 集装箱数目 容量 最优解 当前重量 剩余的质量
    , *x, *bestx, *w;
    // 解空间 最优解空间 集装箱重量数组
    void backtrack(int i);
};

loading::loading(int n, int c, int *w) { //初始化操作
    this->n = n;
    this->c = c;
    this->w = w;
    best = cw = 0;
    x = new int [n];
    bestx = new int[n];
    for(int i=0;i<n;i++) r+=w[i];
    backtrack(0);
    cout<<best<<endl;
    for(int i=0;i<n;i++) cout<<bestx[i]<<" ";
}
```

```

}

void loading::backtrack(int i) {
    if(i>=n) {
        if(cw>best) {
            best = cw;
            for (int j = 0; j < n; ++j)bestx[j] = x[j];
        }
    } else{
        r -= w[i];
        if(cw+w[i]<=c) {
            cw += w[i];           // 选择第i个集装箱
            x[i] = 1;
            backtrack(i+1);
            cw -= w[i];
        }
        if(cw + r > best){// 还有可能存在最大值 不选择集装箱i
            x[i] = 0;
            backtrack(i+1);
        }
        r += w[i];
    }
}
}

```

迭代方式回溯：

相当于使用二叉树的迭代方式，这个以及不是回溯法迭代了，属于分支限界了，相当于二叉树的层序遍历，到达最后一层的时候，将最后一层的最大值取出即可。

```

#include<iostream>
#include<queue>
using namespace std;
#define N 1010
int n, c1, c2, a[N], best; //相当于BFS

int main() {
    cin>>n>>c1>>c2;
    queue<int> q;
    for(int i=0;i<n;i++) {
        #include <bits/stdc++.h>

        using namespace std;
        int n, c;

        int main() {
            cin>>n>>c;
            queue<int> q;
            int w;
            q.push(-1); //使用-1来标志每一层之间的分界点
            for(int i=0;i<n;i++) {
                cin>>w;
                while(q.front() != -1) {
                    if(q.front() + w <= c)q.push(q.front() + w);
                    else q.push(q.front());
                    q.pop();
                }
                q.pop();
                q.push(w);
                q.push(-1);
            }
            int cnt = 0;
            while(!q.empty()) {
                cnt = max(cnt, q.front());
                q.pop();
            }
            cout<<cnt;
        }
    }
}

```

```
    return 0;
}
```

dp动态规划 (0-1背包问题) :

```
#include <bits/stdc++.h>

using namespace std;
int n, c;

int main() {
    cin >> n >> c;
    int *dp = new int[c+10];
    memset(dp, 0, 4*(c+10));
    int w;
    for(int i=0; i<n; i++) {
        cin >> w;
        for(int j=c; j>=w; j--) dp[j] = max(dp[j-w]+w, dp[j]);
    }
    cout << dp[c];
}
```

批处理作业调度

排列数 难度: ☆

```
class FlowShop{
public:
    int best = -1, *best_serial, *cur, *finish, **cost, n;//最优解 最优序列 当前值
    // finish 每一个作业完后的完成时间 cost 花费数组
    FlowShop(int **cost_arr, int num) {
        n = num;           // 初始化
        cur = new int[n];
        finish = new int[n+1];
        best_serial = new int[n+1];
        memset(finish, 0, (n+1)*4);
        for(int i=0; i<n; i++) cur[i] = i;
        cost = cost_arr;
        backtrack(0);
        cout << best << endl;
        for (int j = 0; j < n; ++j) cout << best_serial[j] << " ";
    }

    void backtrack(int i) {
        if(i==n) {
            if(best == -1 || finish[n] < best) {
                best = finish[n]; //记录最优值
                for (int j = 0; j < n; ++j) best_serial[j] = cur[j];
            }
        } else {
            int tmp = finish[n];
            for (int j = i; j < n; ++j) {
                swap(cur[i], cur[j]); // 排列树
                if(i==0) finish[i] = cost[cur[i]][0]+cost[cur[i]][1];
                else finish[i] = finish[i-1]+cost[cur[i]][0]+cost[cur[i]][1]-cost[cur[i-1]][1];
                finish[n] += finish[i]; // 记和
                if(finish[n] < best || best == -1) backtrack(i+1);
                finish[n] = tmp;
                swap(cur[i], cur[j]);
            }
        }
    }
};
```

符号三角形

难度：☆

```
class Triangle
{
    friend int Compute(int);
private:
    void Backtrack(int t);
    int half,
        count,
        **p;
    long sum;
};

void Backtrack(int t)
{
    if((count > half) || (t*(t+1)/2-count>half))
        return;
    if(t>n)
        sum++;
    else
    {
        for(int i=0;i<2;j++)
        {
            p[1][t] = i;
            count+=i;
            for(int j=2;j<=t;j++)
            {
                p[j][t-j+1] = p[j-1][t-j+1]^p[j-1][t-j+2];
                count+=p[j][t-j+1];
            }
            Backtrack(t+1);
            for(int j=2;j<=t;j++)
                count -= p[j][t-j+1];
            count -= i;
        }
    }
}

int Compute(int n)
{
    Triangle X;
    X.n = n;
    X.count = 0;
    X.sum = 0;
    X.half = n*(n+1)/2;
    if(X.half%2 == 1)
        return 0;
    X.half = X.half/2;
    int **p = new int *[n+1];
    for(int i=0;i<=n;i++)
        p[i] = new int [n+1];
    for(int i=0;i<=n;i++)
        for(int j=0;j<=n;j++)
            p[i][j] = 0;
    X.p = p;
    X.Backtrack(1);
    return X.sum;
}
```

N皇后问题

难度：☆

```
int backtrack(int i, int n, int *x) {
    if(i>=n) return 1;
    int nums = 0;
    for (int j = 0; j < n; ++j) {
        bool hit = false;
        for (int k = 0; k < i; ++k) {
            if(j==x[k] || abs(i-k)==abs(j-x[k])) {//比较前面的皇后
                hit = true;
                break;
            }
        }
        if(!hit) {
            x[i] = j;
            nums += backtrack(i+1, n, x);
        }
    }
    return nums;
}

int nQueen(int n) {
    int *x = new int[n];
    memset(x, 0, 4*n);
    return backtrack(0, n, x);
}
```

最大团问题

最大团：离散数学概念，对于图中是最大所有节点的连通图

给定无向图 $G = (V, E)$ ，其中 V 是非空集合，称为顶点集； E 是 V 中元素构成的无序二元组的集合，称为边集，无向图中的边均是顶点的无序对，无序对常用圆括号 “()” 表示。如果 $U \in V$ ，且对任意两个顶点 $u, v \in U$ 有 $(u, v) \in E$ ，则称 U 是 G 的完全子图。 G 的完全子图 U 是 G 的团。 G 的最大团是指 G 的最大完全子图。

如果 $U \subseteq V$ 且对任意 $u, v \in U$ 有 $(u, v) \notin E$ ，则称 U 是 G 的空子图。 G 的空子图 U 是 G 的独立集当且仅当 U 不包含在 G 的更大的空子图中。 G 的最大独立集是 G 中所含顶点数最多的独立集。

对于任一无向图 $G = (V, E)$ ，其补图 $G' = (V', E')$ 定义为： $V' = V$ ，且 $(u, v) \in E'$ 当且仅当 $(u, v) \notin E$ 。

如果 U 是 G 的完全子图，则它也是 G' 的空子图，反之亦然。因此， G 的团与 G' 的独立集之间存在一一对应的关系。特别地， U 是 G 的最大团当且仅当 U 是 G' 的最大独立集。

通俗点讲就是在一个无向图中找出一个点数最多的完全图

```
class maxClique{
    int n, **G, bestn= 0, cnt;
    bool *cur, *best; //节点数目，邻接矩阵，最优结果 节点数 当前节点数
    maxClique(int num, int **graph) {
        n=num, G=graph;
        best = new bool[n];
        cur = new bool[n];
        memset(best, 0, n);
        memset(cur, 0, n);
        backtrack(0);
        cout<<bestn<<endl;
    }
    void backtrack(int i) {
        if(i>=n) {
            if(cnt>bestn) {
                bestn= cnt;
                best=cur;
            }
        }
    }
}
```

```

} else{
    if(i>0){
        bool ok = true;
        for (int j = 0; j < i; ++j) {
            if(cur[j]&&G[i][j]==-1){ //存在一个节点无边
                ok = false;
                break;
            }
        }
        if(ok){
            cur[i] = 1;
            cnt++;
            backtrack(i+1); //添加本结点
            cnt--;
        }
        cur[i] = 0; //不添加此节点
        if(cnt+n-i>bestn)backtrack(i+1);
    }
}
};


```

尝试回溯题

旅行售货商问题

改装Perm就行

```

class TSP{
public:
    int **G, n, best=-1, *best_path,*cur_path;
    TSP(int **Graph, int num){
        G=Graph;
        n=num;
        best_path = new int[n];
        cur_path = new int[n];
        for(int i=0;i<n;i++)cur_path[i]=i;
        backtrack(0,0);
    }
    void backtrack(int i, int cur){
        // i 是递归层数 cur是当前的路程
        if(i==n){
            //链接最后一个节点和初始结点
            if(G[cur_path[n-1]][cur_path[0]]!= -1&&cur+G[cur_path[n-1]][cur_path[0]]<best){
                best = cur+G[cur_path[n-1]][cur_path[0]];
                for (int j = 0; j < n; ++j) best_path[j]=cur_path[i];
            }
            return;
        }
        if(i!=0)cur+=G[cur_path[i-1]][cur_path[i]];
        for (int j = i; j < n; ++j) {
            if(i!=0&&G[cur_path[i-1]][cur_path[j]]==-1)continue;//i-1 和 j 无边
            swap(cur_path[i],cur_path[j]);
            if(i==0)backtrack(i+1,cur);
            else {
                int cost = cur+G[cur_path[i-1]][cur_path[i]];
                if(best== -1||cost<best) backtrack(i+1,cost); //剪枝操作
            }
            swap(cur_path[i],cur_path[j]);
        }
    }
};


```

电路板排列问题

看不懂题目

连续邮资问题

重点题目

假设国家发行了n种不同面值的邮票，并且规定每张信封上最多只允许m张邮票。连续邮资问题要求对于给定的n和m的值，给出邮票面值的最佳设计，即在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。例如，当n=5和m=4时，面值为(1,3,11,15,32)的5种邮票可以贴出邮资的最大连续邮资区间是1 - 70。

```
class stamp {
public:
    int m, n, best=0, maxint = 32767;
    vector<int> x, bestx, y;
    // m邮票数量 n邮票种类 best最大邮资 x邮票选取
    // y:邮资i的最少邮票数目 maxint用于比较
    stamp(int m, int n) { //初始化
        this->m = m;
        this->n = n;
        x = vector<int>(n + 1, 1);
        bestx = vector<int>(n + 1, 0);
        y = vector<int>(1500, maxint); // 邮资的最大值决定
        y[0] = 0;
        backtrack(1, 1);
        cout << best << endl;
        for(int i=1;i<=n;i++) cout << bestx[i] << " ";
    }

    void backtrack(int i, int r) {
        if (i > n) {
            if (r > best) {
                best = r; //有更优解，则替换更优解
                bestx = x;
            }
        } else {
            for (int j = 0; j <= x[i - 1] * (m - 1); ++j) {
                if (y[j] < m) { //使用动态规划的思想
                    for (int k = 1; k <= m - y[j]; ++k)
                        if (y[j] + k < y[j + x[i] * k])
                            y[j + x[i] * k] = y[j] + k;
                }
            }
            while (y[r] < maxint) r++; //依次扫描，得到最小的连续最佳值
            vector<int> tmp = y; //由于后面会破坏y的值，所以需要保存临时的值
            for (int l = x[i-1]+1; l <= r; ++l) {
                x[i+1] = 1; // x[i+1]取值范围从{x[i]+1, r}
                backtrack(i+1, r-1);
                y=tmp; //还原y
            }
        }
    }
};
```

课后习题

5-15 最佳调度问题：

<https://acm.sdu.edu.cn/onlinejudge3/problems/1780>

[Description]

假设有n个任务由k个可并行工作的机器完成。完成任务i需要的时间为 t_i 。试设计一个算法找出完成这n个任务的最佳调度，使得完成全部任务的时间最早。

对任意给定的整数n和k，以及完成任务i需要的时间为 t_i , $i=1 \sim n$ 。计算完成这n个任务的最佳调度。

[Input]

输入数据的第一行有2个正整数n和k, $n \leq 20$, $k \leq 5$ 。第2行的n个正整数是完成n个任务需要的时间。

[Output]

将计算出的完成全部任务的最早时间输出。

```
#include <bits/stdc++.h>

using namespace std;
int n, k;
void dfs(int *t, int *q, int i, int &best) {
    if(i==n) {
        int cnt = 0;
        for(int j=0;j<k;j++) cnt=max(cnt,q[j]); //得到最后完成任务的时间
        best = min(best, cnt);
    } else {
        for(int j=0;j<k;j++) {
            q[j]+=t[i];
            if(q[j]<best)dfs(t,q,i+1,best); //剪枝
            q[j]-=t[i];
        }
    }
}

int main() {
    cin>>n>>k;
    int best = 0x3f3f3f3f; //INF
    int *t = new int[n+5], *q = new int[k];
    memset(q, 0, 4*k);
    for(int i=0;i<n;i++) cin>>t[i];
    dfs(t, q, 0, best);
    cout<<best;
}
```

分支限界法

队列和优先队列

找到满足条件的一个解（极大或者极小值）即可。使用广度优先搜索解空间，找到一个最有利的节点当作扩展节点。

分支限界的基本思想：分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

单源最短路径

Dijkstra

最优装载

普通队列法：

```
struct node{
    int w = 0;
    vector<bool> record;
};

int maxLoading(vector<int> w, int c) {
    int n = w.size(), r=0, index = 0;      // r 剩余集装箱的重量
    node * cw,*best;      //cw 当前节点的信息 best最优解的信息
    for (int i:w)r += i;
    queue<node*> q;      //使用队列
    best = new node;      //初始化
    best->w = 0;
    best->record = vector<bool>(n, false);
    q.push(best);
    q.push(NULL);        //每一层之间使用NULL值隔开
    while(true) {
        cw = q.front();q.pop();
        if(cw == NULL){
            if(index==n-1){ //退出条件
                for (int i = 0; i < n; ++i) if(best->record[i]) cout<<w[i]<<" ";
                cout<<endl;
                return best->w;
            }
            index++;        //进入下一层
            q.push(NULL);
            r-=w[index];    //更新
            continue;
        }
        if(cw->w>best->w)best=cw;
        if(cw->w+w[index]<=c){
            node*tmp = new node;
            tmp->w = cw->w+w[index];
            tmp->record = cw->record;
            tmp->record[index] = true;
            q.push(tmp);
        }
        if(cw->w+r> best->w)q.push(cw);
    }
}

int main() {
    vector<int> w = {10, 38, 4, 6, 12, 18};
    cout<<maxLoading(w,50);
    return 0;
}
```

优先队列法：

```
struct node{
    int prior = 0,wt=0,level,s; //优先级 重量 层数
    bool* res; //记录解
    node(int p,int w,int l,int n) {
        prior=p,wt=w,level=l,s=n;
        res = new bool[n];
        memset(res,0,n);
    }
};

struct cmp{
    bool operator()(node *a, node*b) {return a->prior<b->prior;} //大顶堆结构
};
```

```

int maxLoading(vector<int> w, int c) {
    int n = w.size();
    vector<int> r(n, 0);
    for(int i=n-2; i>=0; i--) r[i] = r[i+1] + w[i+1]; //记录剩余装载量
    priority_queue<node*, vector<node*>, cmp> q; //建立大顶堆
    node * tmp = new node(r[0], 0, 0, n);
    node *best = tmp;
    int i = 0;
    while(i!=n+1) {
        if(tmp->wt>best->wt) best=tmp;
        node *t = new node(tmp->wt+r[i], tmp->wt, i+1, n);
        for (int j = 0; j < n; ++j) t->res[j] = tmp->res[j];
        q.push(t); //子节点a
        if(tmp->wt+w[i]<=c) {
            node *t2 = new node(tmp->wt+r[i]+w[i], tmp->wt+w[i], i+1, n);
            for (int j = 0; j < n; ++j) t2->res[j] = tmp->res[j]; //for循环检查循环变量
            t2->res[i] = true;
            q.push(t2); //子节点b
        }
        tmp = q.top();
        q.pop(); //得到最优节点
        i = tmp->level;
    }
    for(int i=0; i<n; i++)
        if(best->res[i]) cout << w[i] << " ";
    cout << endl;
    return best->wt;
}

```

❖ for循环检查循环变量

旅行售货商问题

优先队列法：

优先级判别标准：当前

```

struct node{
    int lcost = 0, cost = 0, rcost = 0, i = 0;
    //权值 当前花费 x[i+1:n]的最小出边和 第i层
    vector<int> seq;
    node(int index, int lc, int c, int rc, vector<int> sequence) {
        i = index, lcost = lc, cost = c, rcost = rc, seq = sequence;
    }
};

struct cmp{
    bool operator()(node *a, node*b) {return a->lcost>b->lcost;} //小顶堆用
};

class TSP{
public:
    int best=-1, n, min_sum, **G;
    // 最优值 节点数目
    priority_queue<node*, vector<node*>, cmp> minHeap;
    vector<int> op_seq, minOut;
    // op_seq 最优序列 minOut每个节点的最小出度
    node * E; //当前节点
    TSP(int **G, int num) {
        n = num;
        op_seq = vector<int>(n, 0);
        minOut = vector<int>(n, 0);
        for(int i=0; i<n; i++) { //初始化minOut出度
            op_seq[i] = i;
            int min_ = -1;
            for(int j=0; j<n; j++)

```

```

        if(G[i][j] < min_ || min_ == -1)min_ = G[i][j];
        if(min_ == -1){           //不存在边就退出
            cout<<"No Path\n";
            return;
        }
        minOut[i] = min_;
        min_sum += min_;
    }
    E = new node(0, 0, 0, min_sum, op_seq);
    ac();
}
int ac(){
    int idx = E->i;
    while(1){
        if(E->i==n-2){
            // 在n-2的时候就直接可以得出结果了 不必等到n-1
            if(G[E->seq[idx]][E->seq[idx+1]]!=1&&G[E->seq[idx+1]][E->seq[1]]!=1){
                int c = E->cost + G[E->seq[idx]][E->seq[idx+1]] + G[E->seq[idx+1]][E->seq[1]];
                if(best==-1||c<best){
                    best=E->cost=c;
                    op_seq=E->seq;
                    E->i++;
                    minHeap.push(E);
                }
            }
        }else{
            for (int i = idx + 1; i < n; ++i) {
                int s = G[E->seq[idx]][E->seq[i]];
                if(s==-1)continue;//无边则跳过
                if(best==-1||E->cost+s<best){
                    int r = E->cost-minOut[E->i];
                    node *tmp = new node(E->i+1,E->cost+s+r,E->cost+s,r,E->seq);
                    swap(tmp->seq[E->i+1], tmp->seq[i]);
                    minHeap.push(tmp);
                }
            }
        }
        if(minHeap.empty())return best;
        E = minHeap.top();minHeap.pop();
    }
}
};

}

```

布线问题

就是路径搜索问题

```

struct point{
    int x,y;
    bool operator==(point a){return x==a.x&&y==a.y;}
};

bool findPath(point start,point finish, vector<vector<int>> map){
    int dir[][2] = {{1,0}, {-1,0}, {0,-1}, {0,1}}; // 上下左右4个方位
    if(start==finish) return true;
    point cur, tmp;                      //当前节点和临时节点
    queue<point> queue;                  //队列
    queue.push(cur);
    vector<point> path;                 //建立路径数组
    while(!queue.empty()){
        cur = queue.front();queue.pop();      //出队
        for (int i = 0; i < 4; ++i) {          //四个方位
            tmp.x = cur.x+dir[i][0];
            tmp.y = cur.y+dir[i][1];
            if(tmp.x<0||tmp.x>=map.size()||tmp.y<0||tmp.y>=map[0].size())
                continue;
            if(map[tmp.y][tmp.x]==1)
                continue;
            if(tmp==finish)
                return true;
            map[tmp.y][tmp.x]=1;
            queue.push(tmp);
        }
    }
    return false;
}

```

```

        if(map[tmp.x][tmp.y] != -1) { //不是围墙
            if(map[tmp.x][tmp.y] == 0 || map[tmp.x][tmp.y] > map[cur.x][cur.y] + 1) { //更新最新值
                map[tmp.x][tmp.y] = map[cur.x][cur.y] + 1;
                if(tmp == finish) break; //找到节点之后退出
                queue.push(tmp);
            }
        }
    }
    if(tmp == finish) break; //退出
}

cur = finish;
if(cur == finish) {
    path = vector<point> (map[cur.x][cur.y]); //寻找回家的路
    path[0] = start, path[path.size() - 1] = finish;
    for(int i = path.size() - 2; i > 0; i++) {
        for (int j = 0; j < 4; ++j) {
            tmp.x = cur.x + dir[i][0];
            tmp.y = cur.y + dir[i][1];
            if(map[tmp.x][tmp.y] == i) {
                path[i] = tmp;
                cur = tmp;
            }
        }
    }
    cout << "Path length:" << map[finish.x][finish.y] << endl; //输出信息
    cout << "Path way:\n";
    for(auto i : path) cout << "(" << i.x << ", " << i.y << ")" << endl;
    return true;
} while(true);
return false;
}

```

这个就是单纯的广度优先搜索，没有设计到权值什么的。

使用 A^* 算法来优化搜索算法。 $f(x) = g(x) + m(x)$ 权值=已经走过的路程+当前与终点欧式距离。

最大团问题

```

struct node{
    int wt = 0, i, num = 0;
    bool *path;
    node(int weight, int n, int index) {
        wt=weight;
        path=new bool[n];
        i = index;
        memset(path, 0, n);
    }
};

struct cmp{
    bool operator()(node *a, node*b) {return a->wt < b->wt;}
};

class maxClique{
public:
    int **G, n, bestn = 0;
    bool *best;
    maxClique(int **graph, int num) {
        G = graph, n = num;
        clique();
        for (int i = 0; i < n; ++i)
            if(best[i]) cout << i + 1 << " ";
        cout << endl << bestn << endl;
    }

    void clique() {
        priority_queue<node*, vector<node*>, cmp> maxHeap;
        node *cur = new node(n, n, 0);

```

```

maxHeap.push(cur);
while(!maxHeap.empty()) {
    cur = maxHeap.top(); maxHeap.pop();
    if(cur->i==n) {
        if(cur->num>bestn) {
            bestn = cur->num;
            best = cur->path;
        }
    } else {
        bool ok = true;
        for (int i = 0; i < cur->i; ++i)
            if(cur->path[i]&&G[i][cur->i]==-1)
                ok=false;
        if(ok) {
            node * tmp = new node(cur->wt, n, cur->i+1);
            memcpy(tmp->path, cur->path, n);
            tmp->num=cur->num+1;
            tmp->path[cur->i]=true;
            maxHeap.push(tmp);
        }
        if(cur->num+n-cur->i>bestn) {
            node * tmp = new node(cur->wt-1, n, cur->i+1);
            memcpy(tmp->path, cur->path, n);
            tmp->num=cur->num;
            maxHeap.push(tmp);
        }
    }
}
};


```

尝试分支限界

批作业处理调度

较难

串

朴素模式匹配

就是暴力法

```

int stringMatch(string a, string b) {
    //find b in a
    int i=0, j=0, k=0;
    while(i<a.length()&&j<b.length()) {
        if(a[i]==b[j]) {
            i++, j++;
        } else {
            k++;
            i=k;
            j=0;
        }
    }
    if(j>=b.length()) return k;
    else return -1;
}

```

算法分析：

时间复杂度O(mn)，空间复杂度O(1).

KMP算法

关键词：next数组，前缀后缀

算法分析：时间复杂度O(m+n)，使用前缀和后缀的性质来对搜索的过程进一步优化。

```
void get_next(string T, int next[]) {
    int i = 0, j = -1;
    next[0] = -1;
    while (i < T.length()) {
        if (j == -1 || T[i] == T[j]) next[++i] = ++j; //根据前后缀更新next数组
        else j = next[j];
    }
}

int kmp(string s, string t) {
    int i = 0, j = 0;
    int *next = new int[t.length()];
    memset(next, 0, 4 * t.length());
    get_next(t, next);
    while (i < s.length() && j < int(t.length())) {
        if (j == -1 || s[i] == t[j]) {
            i++, j++;
        } else j = next[j]; //找到上一个前缀
    }
    if (j >= t.length()) return i - t.length();
    return -1;
}
```

背包问题专题

视频讲解：<https://www.bilibili.com/video/BV1qt411Z7nE>

01背包问题

每个物品只能选或者不选两种情况

使用 $f[i][j]$ 来表示轮到选择第 i 个物品的时候体积为 j 的价值，则可以得到dp方程：

$$f(i, j) = \begin{cases} f(i-1, j), & \text{discard} \\ f(i-1, j - v[i]) + w[i], & \text{selected} \end{cases}$$

初次得到：

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

const int N = 1010;
int w[N], v[N], dp[N][N]; //w价值数组, v体积数组, dp数组

int main() {
    int n, m; //物品数量, 最大容积
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> v[i] >> w[i];
    for (int i = 1; i < n; i++) {
        for (int j = 0; j <= m; j++) {
            dp[i][j] = dp[i-1][j];
            if (j >= v[i])
```

```

        dp[i][j] = max(dp[i][j], dp[i-1][j-v[i]] + w[i]);
    }
}

int res=0;
for(int i=0;i<=m;i++)res = max(res, dp[n][m]);
cout<<res<<endl;
return 0;
}

```

优化空间：

```

#include<iostream>
using namespace std;

const int N = 1010;
int w[N], v[N], dp[N]; //w价值数组, v体积数组, dp数组

int main() {
    int n, m; //物品数量, 最大容积
    cin>>n>>m;
    for(int i=1;i<=n;i++)cin>>v[i]>>w[i];
    for(int i=1;i<=n;i++) {
        for(int j=m;j>=v[i];j--) {
            dp[j] = max(dp[j], dp[j-v[i]] + w[i]);
            //采用倒序就不会导致dp[i-1][j-v[i]]的值改变
            //如果这里的倒序改成正序, 循环体不变, 那就导致二维时候dp[i-1][j-v[i]]发生改变
            //那就变成完全背包问题了
        }
    }
    cout<<dp[m]<<endl;
    return 0;
}

```

结果的值为01背包最后的最优结果为背包体积小于等于 m 的情况，如果想要找体积恰好为 m 的情况，需要将 $dp[0] = 0$ 之外，其他 $dp[i] = -\infty$ 即可。

完全背包问题

每件物品可以选无限次

思路：由于每个物品都可以选无数次，那就将在遍历每个物品的时候一直选，直到选满背包的体积为止。设 $dp[i], (i \leq V)$ 为体积，每次遍历一个物品，依次更新 $dp[i + k * v[i]] = dp[i] + k * w[i]$ ，其中 k 值满足 $0 \leq k \leq M, i + M * v[i] \leq V \leq i + (M + 1) * v[i]$

```

for(int i = 0; i < n; i++) { //依次选取物品
    for(int j = m; j >= v[i]; j--) { //依次从最大值开始减
        for(int k = 1; k * v[i] <= j; k++) { //依次选取k个物品i
            f[j] = max(f[j], f[j - k * v[i]] + k * w[i]);
        }
    }
}

```

优化时间复杂度：

```

#include<iostream>
using namespace std;
const int N = 1010;
int dp[N];

int main() {
    int n, m;
    cin>>n>>m;
    for(int i=0;i<n;i++) {
        int v, w;
        cin>>v>>w;
        for(int j=v;j<=m;j++) //从低开始计算

```

```

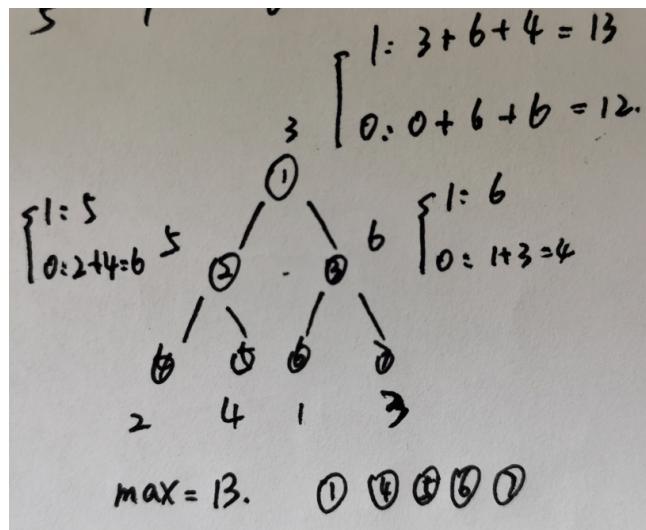
        dp[j] = max(dp[j], dp[j-v]+w);
        //此时就包含了取(k-1)个物品i的情况，数学归纳证明
    }
    cout<<dp[m]<<endl;
    return 0;
}

```

多重背包问题

每个物品选择的上限次数不同

有了前面两道的基础，这道题就简单了



未优化: $O(n^3)$ <https://www.acwing.com/problem/content/4/>

```

#include<iostream>
using namespace std;
const int N = 1010;
int dp[N];

int main() {
    int n, m;
    cin>>n>>m;
    for(int i=0;i<n;i++) {
        int v, w, cnt;
        cin>>v>>w>>cnt;
        for(int j=m;j>=v;j--) {
            for(int k=1;k<=cnt&&k*v<=j;k++) //就是完全背包的O(n^3)样例，加个cnt限制即可
                dp[j] = max(dp[j], dp[j-k*v]+k*w);
        }
    }
    cout<<dp[m]<<endl;
    return 0;
}

```

二进制优化: <https://www.acwing.com/problem/content/5/>

思考如何将混合背包问题转化成01背包问题

由上图可知，该背包问题的解空间树的节点数为 $O(2^n)$ 。

由于多重背包问题限制了每个物品的最多拿取的次数，因此不如使用拆分的思想将m个相同的物品 i ，套用01背包 $O(n^2)$ 的解法实现时间复杂度的减少。

假如一个物品 i 价值为 w 有 10 个，那么根据二进制来讲 10 个用 4 个二进制数来进行表示，最高为 1010。每次进行分割的时候可以分为 1, 2, 4, 8 个。由于 $1+2+4+8=15$ 超过了 10，因此最后一个 8 可以用 3 来替换。那么 10 以内的数字就可以用 1, 2, 4, 3 表示为：

$$1 = 1$$

$$2 = 2 \quad 3 = 1 + 2$$

$$4 = 4 \quad 5 = 1 + 4 \quad 6 = 2 + 4$$

$$7 = 1 + 2 + 4 \quad 8 = 1 + 4 + 3 \quad 9 = 2 + 4 + 3 \quad 10 = 1 + 2 + 4 + 3$$

这样每个数字只出现了一遍，但是经过组合之后仍可以凑满从 0 ~ 10 所有的数字，因此在内层循环中的复杂度就减为了 $O(\log_2 n)$ 数量级，对于 $dp[i]$ 来说，如果 $dp[i]$ 值更新，那么就选取某个基数，如果未更新就不选取即可，总之最大值为 10。

```
#include <bits/stdc++.h>

using namespace std;
const int N = 2020;

int dp[N];

struct Good {
    int v, w;
};

int main() {
    int n, m;
    cin >> n >> m;
    vector<Good> goods;
    for (int i = 0; i < n; i++) {
        int v, w, s;
        cin >> v >> w >> s;
        for (int k = 1; k <= s; k *= 2) {
            s -= k;
            goods.push_back({k * v, k * w});
        }
        if (s > 0) goods.push_back({s * v, s * w});
    }
    for (auto good : goods)
        for (int i = m; i >= good.v; i--)
            dp[i] = max(dp[i], dp[i - good.v] + good.w);
    cout << dp[m] << endl;
}
```

究极版：

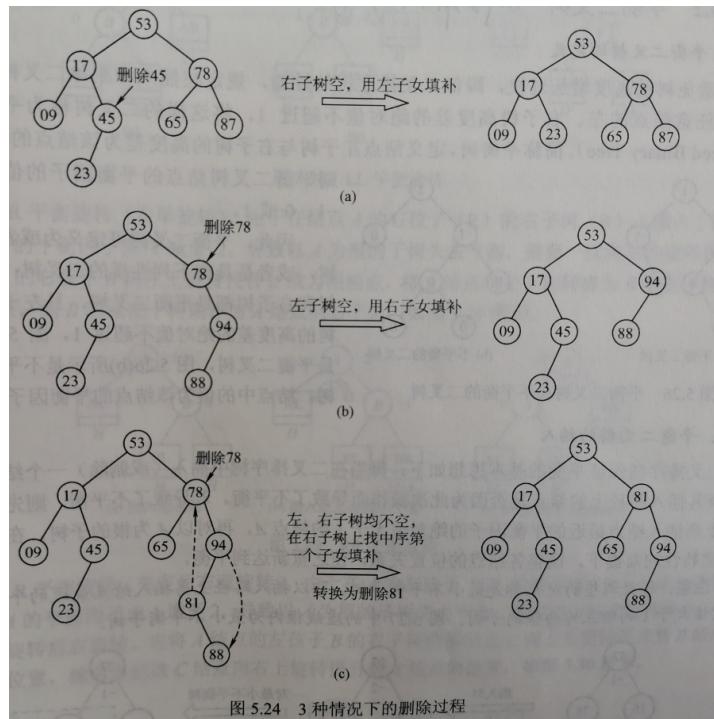


图 5.24 3 种情况下的删除过程

多重背包的单调队列优化

略 😅，太难了，具体提高可以搜索“男人八题”

混合背包问题

多种背包问题混合

将三种问题的for凑在一起就可以

二维费用背包问题

一维限制（只有价值）到二维限制（价值，重量）

有了原先的背包问题的基础，这些就简单多了，只需要加两层倒序即可。

```
#include<iostream>
#include<cstring>
using namespace std;

const int N = 1010;
int f[N][N];

int n, V, M;

int main() {
    cin >> n >> V >> M;
    for (int i=0; i<n; i++) {
        int v, m, w;
        cin >> v >> m >> w;
        for (int j=V; j>=v; j--) {
            for (int k=M; k>=m; k--) {
                f[j][k] = max(f[j-v][k-m] + w, f[j][k]);
            }
        }
        cout << f[V][M] << endl;
    }
    return 0;
}
```

分组背包问题

把物品分成若干组，每一组里面只能选一件，组内互斥

```
#include<iostream>
using namespace std;
const int N = 110;
int f[N], v[N], w[N];

int main() {
    int n, m;
    cin >> n >> m;
    int s;
    for (int i=0; i<n; i++) {
        cin >> s;
        for (int j = 0; j < s; j++) cin >> v[j] >> w[j]; // 对于同类型的物品进行保存
        for (int k=m; k>=0; k--) {
            for (int j=0; j<s; j++) // 对于相同类型的物品进行互斥选择
                if (k>=v[j]) f[k] = max(f[k], f[k-v[j]]+w[j]);
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

真题

2019-3 字典序

给出一篇英语文章，将单词进行字典序排序，只含小写字母

思想：使用基数排序，按照 $a - z$ 的顺序进行升序排序。设置一个 $\text{alpha}[27]$ 的数组用于承载基数排序的数组，其中 $\text{alpha}[0]$ 用于接收比较对应索引大于字符串长度的字符串， $\text{alpha}[s[i] - 'a' + 1]$ ，用接收索引为 i 的字符串，然后使用头结点 head ，依次将 alpha 数组索引按照 0-26 依次连接起来，以此类推，直到指向第 0 位之后， head 连接起来即为已经排好序列的字典序单词。

```
#include <bits/stdc++.h>
using namespace std;

struct lNode {
    string s;
    lNode * next = NULL;
    lNode() {};
    lNode(string str):s(str) {};
};

vector<lNode*> alpha(27);

void appendNode(lNode* head, lNode *t) {
    if (t==NULL) return;
    while (head->next!=NULL) head= head->next;
    head->next = t;
}

int main() {
    string a;
    lNode * head = new lNode("#"); // 构造空头结点
    int max_len = 0;
    while (cin >> a) {
        if (a=="#") break;
        appendNode(head, new lNode(a)); // 输入字符串
        max_len = max(max_len, (int)a.size()); // 记录最长字符串的长度
    }
}
```

```

}

for(int i=0;i<27;i++)alpha[i] = new lNode("#");

for(int j=max_len-1; j>=0; j--) {
    while(head->next!=NULL) {
        lNode * t = head->next;      //这里很重要
        head->next = head->next->next;
        t->next=NULL;           //将取下来的结点的next域进行清零，否则会导致死循环
        if(t->s.length()<=j)appendNode(alpha[0],t);      //长度不足
        else appendNode(alpha[t->s[j]-'a'+1],t);
    }
    for(int i=0;i<27;i++) {
        appendNode(head,alpha[i]->next);      //重新串联起来
        alpha[i]->next = NULL;
    }
}

while(head->next!=NULL) {
    cout<<head->next->s<<endl; //输出字典序
    head=head->next;
}

return 0;
}

```

设 $n = \max(\text{len}(S_i))$ 为最长字符串的长度，则时间复杂度为： $T(n) = O(n * 27) = O(n)$

Tips

- 在OJ中使用动态分配内存的耗费的内存要比静态分配内存大，干脆使用静态分配内存进行重复利用
- `free()` 操作放在循环体的最后