## 一. 基础算法

### 1.1 排序

#### STL sort函数

|  |
| --- |
| **sort(arr, arr + n); // 对 arr[0] 到 arr[n-1] 排序，默认升序 sort(arr, arr + n, greater<int>()); // 降序排序  sort(nodes.begin(), nodes.end()); //容器用迭代器排序  //cmp函数 bool cmp(int a, int b) {** return **a > b; // 降序：a 在 b 前 } sort(arr, arr + n, cmp);  //lambda sort(arr, arr + n, [](int a, int b) {** return **a > b; // 降序 });** |

#### 快速排序算法模板

|  |
| --- |
| **void quick\_sort(int q[], int l, int r) {** if **(l >= r)** return**;   int i = l - 1, j = r + 1, x = q[l + r >> 1];** while **(i < j)  {** do **i ++ ;** while **(q[i] < x);** do **j -- ;** while **(q[j] > x);** if **(i < j) swap(q[i], q[j]);  }  quick\_sort(q, l, j), quick\_sort(q, j + 1, r); }** |

#### 归并排序算法模板

|  |
| --- |
| **void merge\_sort(int q[], int l, int r) {** if **(l >= r)** return**;   int mid = l + r >> 1;  merge\_sort(q, l, mid);  merge\_sort(q, mid + 1, r);   int k = 0, i = l, j = mid + 1;** while **(i <= mid && j <= r)** if **(q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];** else **tmp[k ++ ] = q[j ++ ];** while **(i <= mid) tmp[k ++ ] = q[i ++ ];** while **(j <= r) tmp[k ++ ] = q[j ++ ];** for **(i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j]; }** |

### 1.2 二分

#### 整数二分算法模板

|  |
| --- |
| **bool check(int x) {/\* ... \*/} // 检查x是否满足某种性质 // 右偏 int bsearch\_1(int l, int r) {** while **(l < r)  {  int mid = l + r >> 1;** if **(check(mid)) r = mid; // check()判断mid是否满足性质** else **l = mid + 1;  }** return **l; } // 左偏 int bsearch\_2(int l, int r) {** while **(l < r)  {  int mid = l + r + 1 >> 1;** if **(check(mid)) l = mid;** else **r = mid - 1;  }** return **l; }** |

#### 浮点数二分算法模板

|  |
| --- |
| **bool check(double x) {/\* ... \*/} // 检查x是否满足某种性质  double bsearch\_3(double l, double r) {  const double eps = 1e-6; // eps 表示精度，取决于题目对精度的要求** while **(r - l > eps)  {  double mid = (l + r) / 2;** if **(check(mid)) r = mid;** else **l = mid;  }** return **l; }** |

### 1.3 高精度

#### 高精度加法

|  |
| --- |
| // C = A + B, A >= 0, B >= 0 **vector<int> add(vector<int> &A, vector<int> &B) {** if **(A.size() < B.size())** return **add(B, A);   vector<int> C;  int t = 0;** for **(int i = 0; i < A.size(); i ++ )  {  t += A[i];** if **(i < B.size()) t += B[i];  C.push\_back(t % 10);  t /= 10;  }** if **(t) C.push\_back(t);** return **C; }** |

#### 高精度减法

|  |
| --- |
| // C = A - B, 满足A >= B, A >= 0, B >= 0 **vector<int> sub(vector<int> &A, vector<int> &B) {  vector<int> C;** for **(int i = 0, t = 0; i < A.size(); i ++ )  {  t = A[i] - t;** if **(i < B.size()) t -= B[i];  C.push\_back((t + 10) % 10);** if **(t < 0) t = 1;** else **t = 0;  }** while **(C.size() > 1 && C.back() == 0) C.pop\_back();** return **C; }** |

#### **高精度乘低精度**

|  |
| --- |
| **// C = A \* b, A >= 0, b >= 0 vector<int> mul(vector<int> &A, int b) {  vector<int> C;   int t = 0;** for **(int i = 0; i < A.size() || t; i ++ )  {** if **(i < A.size()) t += A[i] \* b;  C.push\_back(t % 10);  t /= 10;  }** while **(C.size() > 1 && C.back() == 0) C.pop\_back();** return **C; }** |

#### 高精度除以低精度

|  |
| --- |
| **// A / b = C ... r, A >= 0, b > 0 vector<int> div(vector<int> &A, int b, int &r) {  vector<int> C;  r = 0;** for **(int i = A.size() - 1; i >= 0; i -- )  {  r = r \* 10 + A[i];  C.push\_back(r / b);  r %= b;  }  reverse(C.begin(), C.end());** while **(C.size() > 1 && C.back() == 0) C.pop\_back();** return **C; }** |

### 1.4 前缀和与差分

* **前缀和**：用于**查询**为主，适合**固定子区间的快速统计**。
* **差分**：用于**修改**为主，适合**频繁的区间修改操作**。
* 二维场景扩展了思路，可以解决棋盘、地图、图像等多维数据的问题，是动态规划和模拟算法中的重要工具。

#### **一维前缀和**

**核心思想**：快速求任意子区间的元素和。

* 应用场景
  1. **求区间和**：如数组中某段区间的累积和，快速查询多个子区间。
  2. **特定条件下的子数组统计**：如统计满足某和的子数组个数、等差数列的前缀统计等。
  3. **优化暴力循环**：在滑动窗口、双指针结合场景下减少重复计算。
  4. **动态和的判断**：如 LeetCode 560 的“和为 K 的子数组”。

S[i] = a[1] + a[2] + ... a[i]  
a[l] + ... + a[r] = S[r] - S[l - 1]

#### **二维前缀和**

**核心思想**：快速求任意矩形区域的元素和。

* 应用场景
  1. **矩形区域查询**：如地图/棋盘中矩形区域的累积值，快速实现范围统计。
  2. **统计二维频率矩阵**：如统计某字符矩阵内某个字符出现次数。
  3. **处理图像/像素值矩阵**：如积分图的计算，用于快速处理图像区域的统计。
  4. **最大子矩阵和**：如求二维数组的子矩阵的最大和，或固定形状的区域统计。

S[i, j] = 第i行j列格子左上部分所有元素的和  
以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵的和为：  
S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]

#### **一维差分**

**核心思想**：高效处理多次区间修改操作，最终还原原数组。

* 应用场景
  1. **区间增减问题**：如“给区间加上固定值”、“统计区间操作后某值出现的频次”。
  2. **区间频次统计**：如单点操作转换为区间统计，模拟更新效果。
  3. **物理量累积模拟**：如力的分布计算，能量在区间上的增减。
  4. **效率优化**：从 O(n⋅q)O(n \cdot q) 提升到 O(n+q)O(n + q)，如对一个数组进行大量区间修改的场景。

给区间[l, r]中的每个数加上c：B[l] += c, B[r + 1] -= c

#### **二维差分**

**核心思想**：高效处理多次矩形区域修改操作，最终还原 原矩阵。

* 应用场景
  1. **矩形区域增减问题**：如在二维数组的某矩形区域内统一加减一个数。
  2. **累计影响模拟**：如模拟一个范围的热量扩散、光照叠加。
  3. **频次矩阵构建**：如对二维频次表进行增量操作，快速得到最终统计值。
  4. **动态二维修改问题**：如棋盘状态更新，积木或区域重叠分析。

给以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵中的所有元素加上c：  
S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c

### 1.5 双指针算法

|  |
| --- |
| for (int i = 0, j = 0; i < n; i ++ ) {  while (j < i && check(i, j)) j ++ ;   // 具体问题的逻辑 } |

常见问题分类：  
 (1) 对于一个序列，用两个指针维护一段区间  
 (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作

### 1.6 位运算

位运算是直接在二进制位上进行操作的运算。位运算通常用于优化算法，尤其是当涉及到数字处理、快速计算和空间优化

1. 按位与 & （AND）

* **操作规则**：当两个二进制数的对应位都为1时，结果才为1，否则为0。
* 1010  
  & 1101  
  ------  
  1000
  + **检查某一位是否为1**： 例如，n & (1 << k) 判断第k位是否为1。如果为1，结果不为0；否则为0。
  + **清除最低位的1**： n & (n - 1) 会将n的最低位1变成0。

1. 按位或 | （OR）

* **操作规则**：当两个二进制数的对应位中至少有一个为1时，结果为1。
* 1010  
  | 1101  
  ------  
  1111
  + **设置某一位为1**： n | (1 << k) 将第k位置为1。

1. 按位异或 ^ （XOR）

* **操作规则**：当两个二进制数的对应位相异时，结果为1，否则为0。
* 1010  
  ^ 1101  
  ------  
  0111
  + **翻转某一位**： n ^ (1 << k) 将第k位的值翻转（1变成0，0变成1）。
  + **检查两个数是否相等**： a ^ b == 0 表示a与b相等。

1. 按位非 ~ （NOT）

* **操作规则**：对每一位取反，1变成0，0变成1。
* ~ 1010  
  ------  
  0101

1. 左移 << （Left Shift）

* **操作规则**：将二进制数的所有位向左移动指定的位数，左移时低位补0，高位丢弃。
* 1010 << 1 = 10100
  + **乘以2**：n << 1 相当于将n乘以2。
  + **高速计算乘法**：左移可以用来快速计算n的2的幂次方倍数。

1. 右移 >> （Right Shift）

* **操作规则**：将二进制数的所有位向右移动指定的位数，右移时高位补符号位（有符号数时补1，没符号数时补0）。
* 1010 >> 1 = 0101
  + **除以2**：n >> 1 相当于将n除以2。
  + **高速计算除法**：右移可以用来快速计算n的2的幂次方除法。

1. **获取某一位的值**

n >> k & 1

* **说明**：n >> k 将n右移k位，然后与1做按位与运算，这样可以获取n的第k位数字（0或1）。

1. **获取n的最后一位1**

lowbit(n) = n & -n

* **说明**：n & -n 用来得到n的最后一位1。例如，n = 12 (1100) 时，lowbit(n) 的结果为 4 (0100)。
  + -n 是n的二进制补码表示，n & -n 会保留n的最后一位1，并将其他所有位清零。

### **位运算的常见应用**

1. **判断奇偶性**：
   * 通过 n & 1 判断n是否为奇数（如果结果是1则n为奇数）。
2. **快速计算2的幂**：
   * 使用 1 << k 来计算 2^k。
3. **检查二进制表示中是否只有一个1**：
   * n & (n - 1) == 0 可以检查n是否为2的幂（如果n只包含一个1）。
4. **统计二进制中1的个数**：
   * \_\_builtin\_popcount(n) 或 while (n) {n &= (n - 1);}，每次去掉最右边的1。
5. **交换两个数**：
   * a ^= b; b ^= a; a ^= b; 可以用异或交换两个数。
6. **清除低位的1**：
   * n & (n - 1) 会把n的最低位的1清除。
7. **模拟集合**：
   * 通过位运算可以模拟集合的操作，如 set、union、intersection 等。例如，n 可以表示一个集合，其中第k位表示是否包含元素k。

### 1.7 离散化

**核心作用**：将数值范围较大的数据映射到一个较小的连续整数范围，便于在数组、前缀和、差分、树状数组、线段树等数据结构中操作。

主要解决值域大、数据稀疏的问题，结合其他算法实现高效求解，如扫描线、差分数组、树状数组和线段树等。

|  |
| --- |
| vector<int> alls; // 存储所有待离散化的值 **sort(alls.begin(), alls.end()); // 将所有值排序 alls.erase(unique(alls.begin(), alls.end()), alls.end()); // 去掉重复元素  // 二分求出x对应的离散化的值 int find(int x) // 找到第一个大于等于x的位置 {  int l = 0, r = alls.size() - 1;** while **(l < r)  {  int mid = l + r >> 1;** if **(alls[mid] >= x) r = mid;** else **l = mid + 1;  }** return **r + 1; // 映射到1, 2, ...n }** |

#### **经典离散化题目例子**

##### **1. 区间合并和覆盖长度问题**

**题目**：给定 n 个区间 li,ri，求所有区间的覆盖总长度（去重后的长度）。

* **分析**：原区间可能范围很大（如 1,109），无法直接操作。通过离散化，将所有区间端点压缩到较小范围。
* 思路
  1. 离散化所有区间的端点。

1. 用差分数组或扫描线记录区间变化，统计覆盖长度。

##### **2. 区间查询与修改**

**题目**：有 n 个点，每次给区间 l,r 增加一个值 cc，并查询某个点的最终值。

* **分析**：区间范围过大（如 1∼1091 \sim 10^9），通过离散化压缩到有限范围。
* 思路
  1. 离散化所有可能的坐标。

1. 使用差分数组、树状数组或线段树进行区间操作。

##### **3. 动态求第 K 小值**

**题目**：有 n 个操作，每次插入一个值或查询当前数据的第 k 小值。

* **分析**：可能有很大的值范围（如 1,109），通过离散化解决大范围权值问题。
* 思路
  1. 离散化所有插入的值。

1. 用树状数组或线段树统计权值频次。

##### **4. 区间最大重叠次数**

**题目**：给定 n 个区间，求区间重叠最多的时间点及重叠次数。

* **分析**：时间点范围可能很大，需离散化区间端点。
* 思路
  1. 离散化所有区间端点。

1. 使用扫描线或差分数组统计每个离散点的重叠次数。

##### **5. 动态逆序对计数**

**题目**：给定一个初始数组，每次插入一个值，实时输出数组的逆序对个数。

* **分析**：插入值可能范围过大（如 1,109），需离散化值域。
* 思路
  1. 离散化所有可能的值。

1. 用树状数组或线段树动态维护逆序对。

##### **6. 二维区间问题**

**题目**：给定 n 个矩形，求哪些矩形有重叠区域或总覆盖面积。

* **分析**：矩形坐标范围可能很大，需对 x 和 y 坐标分别离散化。
* 思路
  1. 对 x 和 y 轴分别离散化，压缩到二维数组。

1. 使用二维差分或扫描线处理。

##### **7. 模拟车流/时间段事件**

**题目**：给定 n 个车辆的进入和离开时间，求某时刻停车场中车辆的数量。

* **分析**：时间范围可能很大（如 1,109），需离散化时间点。
* 思路
  1. 离散化所有进入和离开的时间点。

1. 使用差分数组统计每个时间点的车辆变化。

##### **8. 区间众数统计**

**题目**：给定一个数组，支持动态查询任意区间内的众数。

* **分析**：值域较大（如 1,109），需离散化元素值。
* 思路
  1. 离散化数组中的值。

1. 用莫队算法或树状数组分块统计。

### **1.8 区间合并**

|  |
| --- |
| using **PII = pair<int, int>;  // 合并区间的函数 void merge(vector<PII> &segs) {  vector<PII> res;  // 1. 按起点排序，如果起点相同，按终点排序  sort(segs.begin(), segs.end());  // 2. 初始化st和ed为不可能出现的值  int st = -2e9, ed = -2e9;  // 3. 遍历所有区间** for **(**auto **seg : segs) {** if **(ed < seg.first) { // 当前区间与前一个区间不重叠** if **(st != -2e9) res.push\_back({st, ed}); // 把上一个区间加入结果  st = seg.first; // 更新st  ed = seg.second; // 更新ed  }** else **{ // 当前区间与前一个区间重叠  ed = max(ed, seg.second); // 合并区间，更新ed  }  }  // 4. 最后一个区间加入结果** if **(st != -2e9) res.push\_back({st, ed});   egs = res; // 更新输入数组 }** |

## **二. 数据结构**

### **2.1 链表与邻接表：树与图的存储**

#### **单链表**

|  |
| --- |
| **// head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点 int head, e[N], ne[N], idx;  // 初始化 void init() {  head = -1;  idx = 0; }  // 在链表头插入一个数a void insert(int a) {  e[idx] = a, ne[idx] = head, head = idx ++ ; }  // 将头结点删除，需要保证头结点存在 void remove() {  head = ne[head]; }** |

#### **双链表**

|  |
| --- |
| **// e[]表示节点的值，l[]表示节点的左指针，r[]表示节点的右指针，idx表示当前用到了哪个节点 int e[N], l[N], r[N], idx;  // 初始化 void init() {  //0是左端点，1是右端点  r[0] = 1, l[1] = 0;  idx = 2; }  // 在节点a的右边插入一个数x void insert(int a, int x) {  e[idx] = x;  l[idx] = a, r[idx] = r[a];  l[r[a]] = idx, r[a] = idx ++ ; }  // 删除节点a void remove(int a) {  l[r[a]] = l[a];  r[l[a]] = r[a]; }** |

### **2.2 栈与队列：单调队列、单调栈**

#### **栈**

|  |
| --- |
| **// tt表示栈顶  int stk[N], tt = 0;    // 向栈顶插入一个数  stk[ ++ tt] = x;    // 从栈顶弹出一个数  tt -- ;    // 栈顶的值  stk[tt];    // 判断栈是否为空** if **(tt > 0)  {  }** |

#### 队列

1. 普通队列：

|  |
| --- |
| // hh 表示队头，tt表示队尾 **int q[N], hh = 0, tt = -1;  // 向队尾插入一个数 q[ ++ tt] = x;  // 从队头弹出一个数 hh ++ ;  // 队头的值 q[hh];  // 判断队列是否为空** if **(hh <= tt) {  }** |

1. **循环队列**

|  |
| --- |
| **// hh 表示队头，tt表示队尾的后一个位置 int q[N], hh = 0, tt = 0;  // 向队尾插入一个数 q[tt ++ ] = x;** if **(tt == N) tt = 0;  // 从队头弹出一个数 hh ++ ;** if **(hh == N) hh = 0;  // 队头的值 q[hh];  // 判断队列是否为空** if **(hh != tt) {  }** |

#### **单调栈**

**常见模型：找出每个数左边离它最近的比它大/小的数**

|  |
| --- |
| **int tt = 0;** for **(int i = 1; i <= n; i ++ ) {** while **(tt && check(stk[tt], i)) tt -- ;  stk[ ++ tt] = i; }** |

#### **单调队列**

**常见模型：找出滑动窗口中的最大值/最小值**

|  |
| --- |
| **int hh = 0, tt = -1;** for **(int i = 0; i < n; i ++ ) {** while **(hh <= tt && check\_out(q[hh])) hh ++ ; // 判断队头是否滑出窗口** while **(hh <= tt && check(q[tt], i)) tt -- ;  q[ ++ tt] = i; }** |

### **2.5 并查集**

**查看是否属于集合**

**(1)朴素并查集：**

|  |
| --- |
| **int p[N]; //存储每个点的祖宗节点   // 返回x的祖宗节点  int find(int x)  {** if **(p[x] != x) p[x] = find(p[x]);** return **p[x];  }   // 初始化，假定节点编号是1~n** for **(int i = 1; i <= n; i ++ ) p[i] = i;   // 合并a和b所在的两个集合：  p[find(a)] = find(b);** |

**(2)维护size的并查集：**

|  |
| --- |
| **int p[N], size[N];  //p[]存储每个点的祖宗节点, size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量   // 返回x的祖宗节点  int find(int x)  {** if **(p[x] != x) p[x] = find(p[x]);** return **p[x];  }   // 初始化，假定节点编号是1~n** for **(int i = 1; i <= n; i ++ )  {  p[i] = i;  size[i] = 1;  }**  **// 合并a和b所在的两个集合：  size[find(b)] += size[find(a)];  p[find(a)] = find(b);** |

**(3)维护到祖宗节点距离的并查集：**

|  |
| --- |
| **int p[N], d[N];  //p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离   // 返回x的祖宗节点  int find(int x)  {** if **(p[x] != x)  {  int u = find(p[x]);  d[x] += d[p[x]];  p[x] = u;  }** return **p[x];  }   // 初始化，假定节点编号是1~n** for **(int i = 1; i <= n; i ++ )  {  p[i] = i;  d[i] = 0;  }   // 合并a和b所在的两个集合：  p[find(a)] = find(b);  d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量** |

### 2.6 堆

|  |
| --- |
| // h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1 // ph[k]存储第k个插入的点在堆中的位置 // hp[k]存储堆中下标是k的点是第几个插入的 **int h[N], ph[N], hp[N], size;  // 交换两个点，及其映射关系 void heap\_swap(int a, int b) {  swap(ph[hp[a]],ph[hp[b]]);  swap(hp[a], hp[b]);  swap(h[a], h[b]); }  void down(int u) {  int t = u;** if **(u \* 2 <= size && h[u \* 2] < h[t]) t = u \* 2;** if **(u \* 2 + 1 <= size && h[u \* 2 + 1] < h[t]) t = u \* 2 + 1;** if **(u != t)  {  heap\_swap(u, t);  down(t);  } }  void up(int u) {** while **(u / 2 && h[u] < h[u / 2])  {  heap\_swap(u, u / 2);  u >>= 1;  } }  // O(n)建堆** for **(int i = n / 2; i; i -- ) down(i);** |

### **2.7 哈希**

#### **一般哈希**

**(1) 拉链法**

|  |
| --- |
| **int h[N], e[N], ne[N], idx;   // 向哈希表中插入一个数  void insert(int x)  {  int k = (x % N + N) % N;  e[idx] = x;  ne[idx] = h[k];  h[k] = idx ++ ;  }   // 在哈希表中查询某个数是否存在  bool find(int x)  {  int k = (x % N + N) % N;** for **(int i = h[k]; i != -1; i = ne[i])** if **(e[i] == x)** returntrue**;** returnfalse**;  }** |

**(2) 开放寻址法**

|  |
| --- |
| **int h[N];   // 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置  int find(int x)  {  int t = (x % N + N) % N;** while **(h[t] != null && h[t] != x)  {  t ++ ;**  if (t == N) t = 0;  }  return t;  } |

#### 字符串哈希

核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低  
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

|  |
| --- |
| typedef **unsigned long long ULL; ULL h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64  // 初始化 p[0] = 1;** for **(int i = 1; i <= n; i ++ ) {  h[i] = h[i - 1] \* P + str[i];  p[i] = p[i - 1] \* P; }  // 计算子串 str[l ~ r] 的哈希值 ULL get(int l, int r) {** return **h[r] - h[l - 1] \* p[r - l + 1]; }** |

### **2.8 字符串**

#### **kmp**

|  |
| --- |
| **// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度  求模式串的Next数组：** for **(int i = 2, j = 0; i <= m; i ++ )  {** while **(j && p[i] != p[j + 1]) j = ne[j];** if **(p[i] == p[j + 1]) j ++ ;   ne[i] = j;  }    // 匹配** for **(int i = 1, j = 0; i <= n; i ++ )  {** while **(j && s[i] != p[j + 1]) j = ne[j];** if **(s[i] == p[j + 1]) j ++ ;** if **(j == m)   {   j = ne[j];   // 匹配成功后的逻辑   }  }** |

#### Trie 树 (**字典树**、**前缀树**)

**Trie树**（前缀树/字典树）是一种树形数据结构，用于高效地存储和检索字符串集合，特别是具有**公共前缀**的字符串。

1. 每个节点代表一个字符串的前缀。
2. 根节点为空，子节点表示从根到该节点的路径组成的前缀。
3. 插入和查询的时间复杂度为 **O(L)**，其中 L 是字符串长度。

##### **常见应用**

1. **字符串查询**：
   * 判断某字符串是否存在。
   * 判断某前缀是否存在。
2. **单词统计**：
   * 统计某字符串出现的次数。
   * 统计以某前缀开头的字符串数量。
3. **最长公共前缀**问题。
4. **字典序排序**：从Trie树中提取所有字符串，可按字典序输出。
5. **自动补全**：根据前缀匹配可能的候选词。
6. **拼词游戏**：判断单词拼接的可能性。

|  |
| --- |
| **int son[N][26], cnt[N], idx; // 0号点既是根节点，又是空节点 // son[][]存储树中每个节点的子节点 // cnt[]存储以每个节点结尾的单词数量  // 插入一个字符串 void insert(char \*str) {  int p = 0;** for **(int i = 0; str[i]; i ++ )  {  int u = str[i] - 'a';** if **(!son[p][u]) son[p][u] = ++ idx;  p = son[p][u];  }  cnt[p] ++ ; }  // 查询字符串出现的次数 int query(char \*str) {  int p = 0;** for **(int i = 0; str[i]; i ++ )  {  int u = str[i] - 'a';** if **(!son[p][u])** return **0;  p = son[p][u];  }** return **cnt[p]; }** |

#### **线段树**

##### **线段树（点更新）**

|  |
| --- |
| **/\*  实现功能：  1. 建立线段树，支持查询区间最大值和区间和。  2. 支持点更新操作（修改单个元素的值）。   适用题目：  - 查询区间最大值：如动态维护区间中的最大元素。  - 查询区间和：如 LeetCode 307. 区域和检索 - 数组可修改。 \*/** struct **node {  int left, right; // 该节点管理的区间 [left, right]  int max, sum; // 该区间的最大值与和 };  node tree[maxn << 2]; // 线段树数组 int a[maxn]; // 输入数组 int n; // 数组长度  // 构建线段树 void build(int m, int l, int r) { // m：当前树节点编号，[l, r]：该节点管理的区间  tree[m].left = l;  tree[m].right = r;** if **(l == r) { // 叶子节点  tree[m].max = a[l];  tree[m].sum = a[l];** return**;  }  int mid = (l + r) >> 1; // 中点  build(m << 1, l, mid); // 递归构建左子树  build(m << 1 | 1, mid + 1, r); // 递归构建右子树  // 合并左右子区间  tree[m].max = max(tree[m << 1].max, tree[m << 1 | 1].max);  tree[m].sum = tree[m << 1].sum + tree[m << 1 | 1].sum; }  // 点更新操作：修改位置 a 的值为原值加 val void update(int m, int a, int val) {** if **(tree[m].left == a && tree[m].right == a) { // 定位到叶子节点  tree[m].max += val;  tree[m].sum += val;** return**;  }  int mid = (tree[m].left + tree[m].right) >> 1;** if **(a <= mid)  update(m << 1, a, val); // 更新左子树** else **update(m << 1 | 1, a, val); // 更新右子树  // 更新当前节点的信息  tree[m].max = max(tree[m << 1].max, tree[m << 1 | 1].max);  tree[m].sum = tree[m << 1].sum + tree[m << 1 | 1].sum; }  // 查询区间 [l, r] 的和 int querySum(int m, int l, int r) {** if **(l == tree[m].left && r == tree[m].right) // 完全覆盖** return **tree[m].sum;  int mid = (tree[m].left + tree[m].right) >> 1;** if **(r <= mid)** return **querySum(m << 1, l, r); // 查询左子树** elseif **(l > mid)** return **querySum(m << 1 | 1, l, r); // 查询右子树** return **querySum(m << 1, l, mid) + querySum(m << 1 | 1, mid + 1, r); // 合并结果 }  // 查询区间 [l, r] 的最大值 int queryMax(int m, int l, int r) {** if **(l == tree[m].left && r == tree[m].right) // 完全覆盖** return **tree[m].max;  int mid = (tree[m].left + tree[m].right) >> 1;** if **(r <= mid)** return **queryMax(m << 1, l, r); // 查询左子树** elseif **(l > mid)** return **queryMax(m << 1 | 1, l, r); // 查询右子树** return **max(queryMax(m << 1, l, mid), queryMax(m << 1 | 1, mid + 1, r)); // 合并结果 }** |

##### **线段树（区间更新）**

|  |
| --- |
| **/\*  实现功能：  1. 构建线段树，支持区间更新操作（区间加）。  2. 支持区间查询操作。   适用题目：  - 区间加操作 + 区间和查询：如 LeetCode 307、HDU 1698。  - 动态维护区间信息，解决区间修改类问题。 \*/** struct **node {  ll l, r; // 区间范围  ll addv, sum; // addv 表示延迟标记（区间加值），sum 表示区间和 };  node tree[maxn << 2]; // 线段树数组  // 更新当前节点信息 void maintain(int id) {  tree[id].sum = tree[id << 1].sum + tree[id << 1 | 1].sum; // 合并左右子树的和 }  // 将延迟标记下推到子节点 void pushdown(int id) {** if **(tree[id].addv) {  int tmp = tree[id].addv;  // 更新子节点的延迟标记和和  tree[id << 1].addv += tmp;  tree[id << 1 | 1].addv += tmp;  tree[id << 1].sum += (tree[id << 1].r - tree[id << 1].l + 1) \* tmp;  tree[id << 1 | 1].sum += (tree[id << 1 | 1].r - tree[id << 1 | 1].l + 1) \* tmp;  tree[id].addv = 0; // 清除当前节点的标记  } }  // 构建线段树 void build(int id, ll l, ll r) {  tree[id].l = l;  tree[id].r = r;  tree[id].addv = 0;  tree[id].sum = 0;** if **(l == r)** return**; // 叶子节点  ll mid = (l + r) >> 1;  build(id << 1, l, mid);  build(id << 1 | 1, mid + 1, r);  maintain(id); }  // 区间更新：将 [l, r] 区间内的每个元素加上 val void updateAdd(int id, ll l, ll r, ll val) {** if **(tree[id].l >= l && tree[id].r <= r) { // 完全覆盖  tree[id].addv += val;  tree[id].sum += (tree[id].r - tree[id].l + 1) \* val;** return**;  }  pushdown(id); // 将标记下推  ll mid = (tree[id].l + tree[id].r) >> 1;** if **(l <= mid) updateAdd(id << 1, l, r, val); // 更新左子树** if **(mid < r) updateAdd(id << 1 | 1, l, r, val); // 更新右子树  maintain(id); // 更新当前节点信息 }  // 区间查询：求 [l, r] 区间的和 void query(int id, ll l, ll r, ll &anssum) {** if **(tree[id].l >= l && tree[id].r <= r) { // 完全覆盖  anssum += tree[id].sum;** return**;  }  pushdown(id); // 将标记下推  ll mid = (tree[id].l + tree[id].r) >> 1;** if **(l <= mid) query(id << 1, l, r, anssum); // 查询左子树** if **(mid < r) query(id << 1 | 1, l, r, anssum); // 查询右子树 }** |

##### **树状数组**

|  |
| --- |
| **/\*  实现功能：  1. 支持前缀和查询。  2. 支持单点更新。   适用题目：  - 区间求和问题：如 LeetCode 307。  - 动态维护前缀和类问题。 \*/  int a[maxn]; // 树状数组存储的前缀和 int n; // 数组长度  // 计算 lowbit int lowbit(int x) {** return **x & (-x); }  // 单点更新：位置 t 加上 d void insert(int t, int d) {** while **(t <= n) {  a[t] += d;  t += lowbit(t);  } }  // 查询前缀和：1 到 t 的和 ll getSum(int t) {  ll sum = 0;** while **(t > 0) {  sum += a[t];  t -= lowbit(t);  }** return **sum; }** |

### 2.9C++ STL

|  |
| --- |
| **vector, 变长数组，倍增的思想  size() 返回元素个数  empty() 返回是否为空  clear() 清空  front()/back()  push\_back()/pop\_back()  begin()/end()  []  支持比较运算，按字典序  pair<int, int>  first, 第一个元素  second, 第二个元素  支持比较运算，以first为第一关键字，以second为第二关键字（字典序）  string，字符串  size()/length() 返回字符串长度  empty()  clear()  substr(起始下标，(子串长度)) 返回子串  c\_str() 返回字符串所在字符数组的起始地址  queue, 队列  size()  empty()  push() 向队尾插入一个元素  front() 返回队头元素  back() 返回队尾元素  pop() 弹出队头元素  priority\_queue, 优先队列，默认是大根堆  size()  empty()  push() 插入一个元素  top() 返回堆顶元素  pop() 弹出堆顶元素  定义成小根堆的方式：priority\_queue<int, vector<int>, greater<int>> q;  stack, 栈  size()  empty()  push() 向栈顶插入一个元素  top() 返回栈顶元素  pop() 弹出栈顶元素  deque, 双端队列  size()  empty()  clear()  front()/back()  push\_back()/pop\_back()  push\_front()/pop\_front()  begin()/end()  []  set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列  size()  empty()  clear()  begin()/end()  ++, -- 返回前驱和后继，时间复杂度 O(logn)   set/multiset  insert() 插入一个数  find() 查找一个数  count() 返回某一个数的个数  erase()  (1) 输入是一个数x，删除所有x O(k + logn)  (2) 输入一个迭代器，删除这个迭代器  lower\_bound()/upper\_bound()  lower\_bound(x) 返回大于等于x的最小的数的迭代器  upper\_bound(x) 返回大于x的最小的数的迭代器  map/multimap  insert() 插入的数是一个pair  erase() 输入的参数是pair或者迭代器  find()  [] 注意multimap不支持此操作。 时间复杂度是 O(logn)  lower\_bound()/upper\_bound()  unordered\_set, unordered\_map, unordered\_multiset, unordered\_multimap, 哈希表  和上面类似，增删改查的时间复杂度是 O(1)  不支持 lower\_bound()/upper\_bound()， 迭代器的++，--  bitset, 圧位  bitset<10000> s;  ~, &, |, ^  >>, <<  ==, !=  []   count() 返回有多少个1   any() 判断是否至少有一个1  none() 判断是否全为0   set() 把所有位置成1  set(k, v) 将第k位变成v  reset() 把所有位变成0  flip() 等价于~  flip(k) 把第k位取反** |

## 三. 搜索与图论

### 3.1 DFS 与 BFS

#### DFS 全排列

|  |
| --- |
| /\*  |求1到n的全排列, 有条件|  |16/11/05ztx, thanks to wangqiqi| \*/  void Pern(int list[], int k, int n) { // k表示前k个数不动仅移动后面n-k位数  if (k == n - 1) {  for (int i = 0; i < n; i++) {  printf("%d", list[i]);  }  printf("\n");  }else {  for (int i = k; i < n; i++) { // 输出的是满足移动条件所有全排列  swap(list[k], list[i]);  Pern(list, k + 1, n);  swap(list[k], list[i]);  }  } } |

#### BFS 最短路径问题（无权图）(迷宫)

|  |
| --- |
| void bfs(int start, vector<vector<int>>& graph) {  queue<int> q;  vector<bool> visited(graph.size(), false);  q.push(start);  visited[start] = true;   while (!q.empty()) {  int node = q.front();  q.pop();  // 处理节点 node 的逻辑   for (int neighbor : graph[node]) {  if (!visited[neighbor]) {  visited[neighbor] = true;  q.push(neighbor);  }  }  } } |

### 3.2 树与图的遍历：拓扑排序

#### 树与图的存储

树是一种特殊的图，与图的存储方式相同。  
对于[无向图](https://so.csdn.net/so/search?q=%E6%97%A0%E5%90%91%E5%9B%BE&spm=1001.2101.3001.7020)中的边 ab，存储两条有向边 a->b, b->a。  
因此我们可以只考虑有向图的存储。

(1) [邻接矩阵](https://so.csdn.net/so/search?q=%E9%82%BB%E6%8E%A5%E7%9F%A9%E9%98%B5&spm=1001.2101.3001.7020)：ga 存储边 a->b

(2) 邻接表：

|  |
| --- |
| // 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点 int h[N], e[N], ne[N], idx;  // 添加一条边a->b void add(int a, int b) {  e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ; }  // 初始化 idx = 0; memset(h, -1, sizeof h); |

#### 树与图的遍历

**时间复杂度 O(n+m), n 表示点数，m 表示边数**  
(1) 深度优先遍历

|  |
| --- |
| int dfs(int u) {  st[u] = true; // st[u] 表示点u已经被遍历过   for (int i = h[u]; i != -1; i = ne[i])  {  int j = e[i];  if (!st[j]) dfs(j);  } } |

(2) 宽度优先遍历

|  |
| --- |
| queue<int> q; st[1] = true; // 表示1号点已经被遍历过 q.push(1);  while (q.size()) {  int t = q.front();  q.pop();   for (int i = h[t]; i != -1; i = ne[i])  {  int j = e[i];  if (!st[j])  {  st[j] = true; // 表示点j已经被遍历过  q.push(j);  }  } } |

#### 拓扑排序

**时间复杂度 O(n+m), n 表示点数，m 表示边数**

|  |
| --- |
| bool topsort() {  int hh = 0, tt = -1;   // d[i] 存储点i的入度  for (int i = 1; i <= n; i ++ )  if (!d[i])  q[ ++ tt] = i;   while (hh <= tt)  {  int t = q[hh ++ ];   for (int i = h[t]; i != -1; i = ne[i])  {  int j = e[i];  if (-- d[j] == 0)  q[ ++ tt] = j;  }  }   // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。  return tt == n - 1; } |

### 3.3 最短路

#### 朴素 dijkstra 算法

**时间复杂是 O(n2+m), n 表示点数，m 表示边数**

|  |
| --- |
| int g[N][N]; // 存储每条边 int dist[N]; // 存储1号点到每个点的最短距离 bool st[N]; // 存储每个点的最短路是否已经确定  // 求1号点到n号点的最短路，如果不存在则返回-1 int dijkstra() {  memset(dist, 0x3f, sizeof dist);  dist[1] = 0;   for (int i = 0; i < n - 1; i ++ )  {  int t = -1; // 在还未确定最短路的点中，寻找距离最小的点  for (int j = 1; j <= n; j ++ )  if (!st[j] && (t == -1 || dist[t] > dist[j]))  t = j;   // 用t更新其他点的距离  for (int j = 1; j <= n; j ++ )  dist[j] = min(dist[j], dist[t] + g[t][j]);   st[t] = true;  }   if (dist[n] == 0x3f3f3f3f) return -1;  return dist[n]; } |

#### 堆优化版 dijkstra

**时间复杂度 O(mlogn), n 表示点数，m 表示边数**

|  |
| --- |
| typedef pair<int, int> PII;  int n; // 点的数量 int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边 int dist[N]; // 存储所有点到1号点的距离 bool st[N]; // 存储每个点的最短距离是否已确定  // 求1号点到n号点的最短距离，如果不存在，则返回-1 int dijkstra() {  memset(dist, 0x3f, sizeof dist);  dist[1] = 0;  priority\_queue<PII, vector<PII>, greater<PII>> heap;  heap.push({0, 1}); // first存储距离，second存储节点编号   while (heap.size())  {  auto t = heap.top();  heap.pop();   int ver = t.second, distance = t.first;   if (st[ver]) continue;  st[ver] = true;   for (int i = h[ver]; i != -1; i = ne[i])  {  int j = e[i];  if (dist[j] > distance + w[i])  {  dist[j] = distance + w[i];  heap.push({dist[j], j});  }  }  }   if (dist[n] == 0x3f3f3f3f) return -1;  return dist[n]; } |

#### Bellman-Ford 算法

**时间复杂度 O(nm), n 表示点数，m 表示边数**

|  |
| --- |
| int n, m; // n表示点数，m表示边数 int dist[N]; // dist[x]存储1到x的最短路距离  struct Edge // 边，a表示出点，b表示入点，w表示边的权重 {  int a, b, w; }edges[M];  // 求1到n的最短路距离，如果无法从1走到n，则返回-1。 int bellman\_ford() {  memset(dist, 0x3f, sizeof dist);  dist[1] = 0;   // 如果第n次迭代仍然会松弛三角不等式，就说明存在一条长度是n+1的最短路径，由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。  for (int i = 0; i < n; i ++ )  {  for (int j = 0; j < m; j ++ )  {  int a = edges[j].a, b = edges[j].b, w = edges[j].w;  if (dist[b] > dist[a] + w)  dist[b] = dist[a] + w;  }  }   if (dist[n] > 0x3f3f3f3f / 2) return -1;  return dist[n]; } |

#### spfa 算法（带有**负权边**的最短路径问题）

**时间复杂度 平均情况下 O(m)，最坏情况下 O(nm), n 表示点数，m 表示边数**

|  |
| --- |
| int n; // 总点数 int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边 int dist[N]; // 存储每个点到1号点的最短距离 bool st[N]; // 存储每个点是否在队列中  // 求1号点到n号点的最短路距离，如果从1号点无法走到n号点则返回-1 int spfa() {  memset(dist, 0x3f, sizeof dist);  dist[1] = 0;   queue<int> q;  q.push(1);  st[1] = true;   while (q.size())  {  auto t = q.front();  q.pop();   st[t] = false;   for (int i = h[t]; i != -1; i = ne[i])  {  int j = e[i];  if (dist[j] > dist[t] + w[i])  {  dist[j] = dist[t] + w[i];  if (!st[j]) // 如果队列中已存在j，则不需要将j重复插入  {  q.push(j);  st[j] = true;  }  }  }  }   if (dist[n] == 0x3f3f3f3f) return -1;  return dist[n]; } |

#### spfa 判断图中是否存在负环

**时间复杂度是 O(nm), n 表示点数，m 表示边数**

|  |
| --- |
| int n; // 总点数 int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边 int dist[N], cnt[N]; // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路中经过的点数 bool st[N]; // 存储每个点是否在队列中  // 如果存在负环，则返回true，否则返回false。 bool spfa() {  // 不需要初始化dist数组  // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉原理一定有两个点相同，所以存在环。   queue<int> q;  for (int i = 1; i <= n; i ++ )  {  q.push(i);  st[i] = true;  }   while (q.size())  {  auto t = q.front();  q.pop();   st[t] = false;   for (int i = h[t]; i != -1; i = ne[i])  {  int j = e[i];  if (dist[j] > dist[t] + w[i])  {  dist[j] = dist[t] + w[i];  cnt[j] = cnt[t] + 1;  if (cnt[j] >= n) return true; // 如果从1号点到x的最短路中包含至少n个点（不包括自己），则说明存在环  if (!st[j])  {  q.push(j);  st[j] = true;  }  }  }  }   return false; } |

#### floyd 算法

**时间复杂度是 O(n3), nn 表示点数**

初始化：

|  |
| --- |
| for (int i = 1; i <= n; i ++ )  for (int j = 1; j <= n; j ++ )  if (i == j) d[i][j] = 0;  else d[i][j] = INF;  // 算法结束后，d[a][b]表示a到b的最短距离 void floyd() {  for (int k = 1; k <= n; k ++ )  for (int i = 1; i <= n; i ++ )  for (int j = 1; j <= n; j ++ )  d[i][j] = min(d[i][j], d[i][k] + d[k][j]); } |

### 3.4 最小生成树

#### 朴素版 prim 算法

**时间复杂度是 O(n2+m), n 表示点数，m 表示边数**

|  |
| --- |
| int n; // n表示点数 int g[N][N]; // 邻接矩阵，存储所有边 int dist[N]; // 存储其他点到当前最小生成树的距离 bool st[N]; // 存储每个点是否已经在生成树中   // 如果图不连通，则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和 int prim() {  memset(dist, 0x3f, sizeof dist);   int res = 0;  for (int i = 0; i < n; i ++ )  {  int t = -1;  for (int j = 1; j <= n; j ++ )  if (!st[j] && (t == -1 || dist[t] > dist[j]))  t = j;   if (i && dist[t] == INF) return INF;   if (i) res += dist[t];  st[t] = true;   for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);  }   return res; } |

#### Kruskal 算法

**时间复杂度是 O(mlogm), n 表示点数，m 表示边数**

|  |
| --- |
| int n, m; // n是点数，m是边数 int p[N]; // 并查集的父节点数组  struct Edge // 存储边 {  int a, b, w;   bool operator< (const Edge &W)const  {  return w < W.w;  } }edges[M];  int find(int x) // 并查集核心操作 {  if (p[x] != x) p[x] = find(p[x]);  return p[x]; }  int kruskal() {  sort(edges, edges + m);   for (int i = 1; i <= n; i ++ ) p[i] = i; // 初始化并查集   int res = 0, cnt = 0;  for (int i = 0; i < m; i ++ )  {  int a = edges[i].a, b = edges[i].b, w = edges[i].w;   a = find(a), b = find(b);  if (a != b) // 如果两个连通块不连通，则将这两个连通块合并  {  p[a] = b;  res += w;  cnt ++ ;  }  }   if (cnt < n - 1) return INF;  return res;  } |

### 3.5 二分图：染色法、匈牙利算法

**时间复杂度是 O(n+m), n 表示点数，m 表示边数**

#### 染色法判别二分图

判断一个图是不是二分图

|  |
| --- |
| int n; // n表示点数 int h[N], e[M], ne[M], idx; // 邻接表存储图 int color[N]; // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色  // 参数：u表示当前节点，c表示当前点的颜色 bool dfs(int u, int c) {  color[u] = c;  for (int i = h[u]; i != -1; i = ne[i])  {  int j = e[i];  if (color[j] == -1)  {  if (!dfs(j, !c)) return false;  }  else if (color[j] == c) return false;  }   return true; }  bool check() {  memset(color, -1, sizeof color);  bool flag = true;  for (int i = 1; i <= n; i ++ )  if (color[i] == -1)  if (!dfs(i, 0))  {  flag = false;  break;  }  return flag; } |

#### 匈牙利算法（二分图最大匹配问题）

**在一个二分图中找到最大匹配数**，即尽可能多的匹配两侧的点对。

**时间复杂度是 O(nm), n 表示点数，m 表示边数**

|  |
| --- |
| int n1, n2; // n1表示第一个集合中的点数，n2表示第二个集合中的点数 int h[N], e[M], ne[M], idx; // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边 int match[N]; // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个 bool st[N]; // 表示第二个集合中的每个点是否已经被遍历过  bool find(int x) {  for (int i = h[x]; i != -1; i = ne[i])  {  int j = e[i];  if (!st[j])  {  st[j] = true;  if (match[j] == 0 || find(match[j]))  {  match[j] = x;  return true;  }  }  }   return false; }  // 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点 int res = 0; for (int i = 1; i <= n1; i ++ ) {  memset(st, false, sizeof st);  if (find(i)) res ++ ; } |

## 四. 数学知识

### 4.1 质数

#### 试除法判定质数

|  |
| --- |
| bool is\_prime(int x) {  if (x < 2) return false;  for (int i = 2; i <= x / i; i ++ )  if (x % i == 0)  return false;  return true; } |

#### 试除法分解质因数

|  |
| --- |
| int primes[N], cnt; // primes[]存储所有素数 bool st[N]; // st[x]存储x是否被筛掉  void get\_primes(int n) {  for (int i = 2; i <= n; i ++ )  {  if (st[i]) continue;  primes[cnt ++ ] = i;  for (int j = i + i; j <= n; j += i)  st[j] = true;  } } |

#### 线性筛法求素数

|  |
| --- |
| int primes[N], cnt; // primes[]存储所有素数 bool st[N]; // st[x]存储x是否被筛掉  void get\_primes(int n) {  for (int i = 2; i <= n; i ++ )  {  if (!st[i]) primes[cnt ++ ] = i;  for (int j = 0; primes[j] <= n / i; j ++ )  {  st[primes[j] \* i] = true;  if (i % primes[j] == 0) break;  }  } } |

### 4.2 约数

#### 试除法求所有约数

|  |
| --- |
| vector<int> get\_divisors(int x) {  vector<int> res;  for (int i = 1; i <= x / i; i ++ )  if (x % i == 0)  {  res.push\_back(i);  if (i != x / i) res.push\_back(x / i);  }  sort(res.begin(), res.end());  return res; } |

#### 约数个数和约数之和

如果 N = p1^c1 \* p2^c2 \* ... \*pk^ck  
约数个数： (c1 + 1) \* (c2 + 1) \* ... \* (ck + 1)  
约数之和： (p1^0 + p1^1 + ... + p1^c1) \* ... \* (pk^0 + pk^1 + ... + pk^ck)

### 4.3 快速幂

求 m^k mod p，时间复杂度 O(logk)。

|  |
| --- |
| int qmi(int m, int k, int p) {  int res = 1 % p, t = m;  while (k)  {  if (k&1) res = res \* t % p;  t = t \* t % p;  k >>= 1;  }  return res; } |

### 4.4 辗转相除法

#### GCD 求最大公约数

|  |
| --- |
| int gcd(int big, int small) {  if (small > big) swap(big, small);  int temp;  while (small != 0){ // 辗转相除法  if (small > big) swap(big, small);  temp = big % small;  big = small;  small = temp;  }  return(big); } |

#### LCMc 求最小公倍数

|  |
| --- |
| int gcd(int big, int small) {  if (small > big) swap(big, small);  int temp;  while (small != 0){ // 辗转相除法  if (small > big) swap(big, small);  temp = big % small;  big = small;  small = temp;  }  return(big); } |

### 4.5几何计算

#### 向量基本用法

|  |
| --- |
| struct node {   double x; // 横坐标   double y; // 纵坐标  };   typedef node Vector;  Vector operator + (Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }  Vector operator - (Point A, Point B) { return Vector(A.x - B.y, A.y - B.y); }  Vector operator \* (Vector A, double p) { return Vector(A.x\*p, A.y\*p); }  Vector operator / (Vector A, double p) { return Vector(A.x / p, A.y\*p); }   double Dot(Vector A, Vector B) { return A.x\*B.x + A.y\*B.y; } // 向量点乘  double Length(Vector A) { return sqrt(Dot(A, A)); } // 向量模长  double Angle(Vector A, Vector B) { return acos(Dot(A, B) / Length(A) / Length(B)); } // 向量之间夹角   double Cross(Vector A, Vector B) { // 叉积计算 公式   return A.x\*B.y - A.y\*B.x;  }   Vector Rotate(Vector A, double rad) // 向量旋转 公式 {   return Vector(A.x\*cos(rad) - A.y\*sin(rad), A.x\*sin(rad) + A.y\*cos(rad));  }   Point getLineIntersection(Point P, Vector v, Point Q, Vector w) { // 两直线交点t1 t2计算公式   Vector u = P - Q;   double t = Cross(w, u) / Cross(v, w); // 求得是横坐标   return P + v\*t; // 返回一个点  } |

#### 求多边形面积

|  |
| --- |
| node G[maxn];  int n;   double Cross(node a, node b) { // 叉积计算   return a.x\*b.y - a.y\*b.x;  }    int main()  {   while (scanf("%d", &n) != EOF && n) {   for (int i = 0; i < n; i++)   scanf("%lf %lf", &G[i].x, &G[i].y);   double sum = 0;   G[n].x = G[0].x;   G[n].y = G[0].y;   for (int i = 0; i < n; i++) {   sum += Cross(G[i], G[i + 1]);   }   // 或者   //for (int i = 0; i < n; i++) {   //sum += fun(G[i], G[（i + 1）% n]);   //}   sum = sum / 2.0;   printf("%.1f\n", sum);   }   system("pause");   return 0;  } |

#### 判断线段相交

|  |
| --- |
| node P[35][105];   double Cross\_Prouct(node A,node B,node C) { // 计算BA叉乘CA   return (B.x-A.x)\*(C.y-A.y)-(B.y-A.y)\*(C.x-A.x);  }  bool Intersect(node A,node B,node C,node D) { // 通过叉乘判断线段是否相交；   if(min(A.x,B.x)<=max(C.x,D.x)&& // 快速排斥实验；   min(C.x,D.x)<=max(A.x,B.x)&&   min(A.y,B.y)<=max(C.y,D.y)&&   min(C.y,D.y)<=max(A.y,B.y)&&   Cross\_Prouct(A,B,C)\*Cross\_Prouct(A,B,D)<0&& // 跨立实验；   Cross\_Prouct(C,D,A)\*Cross\_Prouct(C,D,B)<0) // 叉乘异号表示在两侧；   return true;   else return false;  } |

#### 求三角形外心

|  |
| --- |
| Point circumcenter(const Point &a, const Point &b, const Point &c) { //返回三角形的外心   Point ret;   double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1\*a1 + b1\*b1) / 2;   double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2\*a2 + b2\*b2) / 2;   double d = a1\*b2 - a2\*b1;   ret.x = a.x + (c1\*b2 - c2\*b1) / d;   ret.y = a.y + (a1\*c2 - a2\*c1) / d;   return ret;  } |

#### 极角排序

|  |
| --- |
| double cross(point p1, point p2, point q1, point q2) { // 叉积计算   return (q2.y - q1.y)\*(p2.x - p1.x) - (q2.x - q1.x)\*(p2.y - p1.y);  }  bool cmp(point a, point b) {   point o;   o.x = o.y = 0;   return cross(o, b, o, a) < 0; // 叉积判断  }  sort(convex + 1, convex + cnt, cmp); // 按角排序, 从小到大 |

## 五. 动态规划

### **5.1 背包问题**

dp[j] 表示在容量为 j 的背包中，能够获得的最大价值。

#### **01背包模板**

每个物品只能选一次。

|  |
| --- |
| for (int i = 1; i <= n; i++) { // 遍历物品  for (int j = m; j >= w[i]; j--) { // 遍历容量，从大到小  dp[j] = max(dp[j], dp[j - w[i]] + v[i]);  } } |

#### **完全背包模板**

每个物品可以选无限次。

|  |
| --- |
| for (int i = 1; i <= n; i++) { // 遍历物品  for (int j = w[i]; j <= m; j++) { // 遍历容量，从小到大  dp[j] = max(dp[j], dp[j - w[i]] + v[i]);  } } |

#### **多重背包模板**

每个物品有固定数量。

|  |
| --- |
| for (int i = 1; i <= n; i++) {  for (int k = 1; k <= s[i]; k++) { // 枚举每个物品的数量  for (int j = m; j >= w[i]; j--) {  dp[j] = max(dp[j], dp[j - w[i]] + v[i]);  }  } } |

### **5.2 线性 DP**

#### **最大子数组和**

在一个整数数组中找到一个具有最大和的连续子数组。

dp[i] 表示以第 i 个元素结尾的最大子数组和。

|  |
| --- |
| int dp[n];  dp[0] = a[0]; int res = dp[0]; for (int i = 1; i < n; i++) {  dp[i] = max(a[i], dp[i - 1] + a[i]);  res = max(res, dp[i]); } |

#### **最长递增子序列: LIS模板**

dp[i] 表示以 nums[i] 结尾的最长递增子序列的长度。

|  |
| --- |
| /\*  状态转移dp[i] = max{ 1.dp[j] + 1 }; j<i; a[j]<a[i];  d[i]是以i结尾的最长上升子序列  与i之前的 每个a[j]<a[i]的 j的位置的最长上升子序列+1后的值比较 \*/ void solve(){ // 参考挑战程序设计入门经典;  for(int i = 0; i < n; ++i){   dp[i] = 1;   for(int j = 0; j < i; ++j){   if(a[j] < a[i]){   dp[i] = max(dp[i], dp[j] + 1);   } } } }  /\*   优化方法：  dp[i]表示长度为i+1的上升子序列的最末尾元素   找到第一个比dp末尾大的来代替  \*/   void solve() {   for (int i = 0; i < n; ++i){  dp[i] = INF;  }  for (int i = 0; i < n; ++i) {   \*lower\_bound(dp, dp + n, a[i]) = a[i]; // 返回一个指针   }   printf("%d\n", \*lower\_bound(dp, dp + n, INF) - dp;   } /\*   函数lower\_bound()返回一个 iterator 它指向在[first,last)标记的有序序列中可以插入value，而不会破坏容器顺序的第一个位置，而这个位置标记了一个不小于value的值。 \*/ |

#### 最长公共子序列 LCS

* dp[i][j] 表示字符串 s1 的前 i 个字符和字符串 s2 的前 j 个字符的最长公共子序列的长度。

|  |
| --- |
| void solve() {   for (int i = 0; i < n; ++i) {  for (int j = 0; j < m; ++j) {   if (s1[i] == s2[j]) {   dp[i + 1][j + 1] = dp[i][j] + 1;   }else {   dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j]);   } } } } |

### **5.3 区间 DP**

* 求解需要分割或合并区间的最优值。

1. **戳气球**：LeetCode 312。
2. **石子合并问题**：AcWing 282。

|  |
| --- |
| for (int len = 2; len <= n; len++) { // 枚举区间长度  for (int l = 1; l + len - 1 <= n; l++) { // 枚举左端点  int r = l + len - 1; // 计算右端点  for (int k = l; k < r; k++) { // 枚举分割点  dp[l][r] = max(dp[l][r], dp[l][k] + dp[k + 1][r] + cost);  } } } |

### **5.4 状态压缩 DP**

* 适用于子集问题，利用位运算压缩状态。
* 状态表示为某个集合的子集。

1. **旅行商问题（TSP）**：AcWing 91。
2. **划分子集和问题**：LeetCode 698。

|  |
| --- |
| for (int mask = 0; mask < (1 << n); mask++) { // 枚举所有状态  for (int i = 0; i < n; i++) { // 枚举当前状态下可选元素  if (mask & (1 << i)) {  dp[mask] = min(dp[mask], dp[mask ^ (1 << i)] + cost);  }  } } |

### **5.5 树形 DP**

* 用于树形结构的最优值问题。

1. **树的直径**：LeetCode 124。
2. **树的最大路径和**：AcWing 285。

|  |
| --- |
| void dfs(int u, int parent) {  for (int v : graph[u]) {  if (v == parent) continue;  dfs(v, u);  dp[u] = max(dp[u], dp[v] + value[v]);  } } |

### **5.6 记忆化搜索**

* 使用递归 + 记忆化数组避免重复计算。
* 自底向上递归。

1. **爬楼梯问题**：LeetCode 70。
2. **三角形最小路径和**：LeetCode 120。

|  |
| --- |
| int dfs(int u) {  if (vis[u]) return dp[u];  vis[u] = 1;  dp[u] = ...; // 状态转移  return dp[u]; } |

## 六. 贪心

### 6.1 区间选点

给定 N 个闭区间 [ai,bi]，请你在数轴上选择尽量少的点，使得每个区间内至少包含一个选出的点。  
输出选择的点的最小数量。  
位于区间端点上的点也算作区间内。

思路：

1. 将所有区间按照右端点排序
2. 遍历所有区间，ed 初始化为无穷小  
   如果本次区间不能覆盖上次区间的右端点，ed<e[i].l，那么需要选择一个新的点 res++;ed=e[i].r  
   如果本次区间可以覆盖上次区间的右端点，则进行下一轮循环

|  |
| --- |
| #include<bits/stdc++.h> using namespace std; int n; struct SS{  int l,r; }e[100010]; bool cmp(SS x,SS y) {  return x.r<y.r; } int main() {  cin>>n;  for(int i=1;i<=n;i++)  {  int x,y;  cin>>x>>y;  e[i]={x,y};  }  sort(e+1,e+1+n,cmp);  int ed=-2e8,res=0;  for(int i=1;i<=n;i++)  {  if(e[i].l>ed)  {  res++;  ed=e[i].r;  }  }  cout<<res;  return 0; } |

### 6.2 区间分组

给定 N 个闭区间 [ai,bi]，请你将这些区间分成若干组，使得每组内部的区间两两之间（包括端点）没有交集，并使得组数尽可能小。  
输出最小组数。

应用场景：  
有若干个活动，第 i 个活动开始时间和结束时间是 [SiSi,fifi]，同一个教室安排的活动之间不能交叠，求要安排所有活动，少需要几个教室？

思路：

1. 将所有区间按照左端点从小到大排序
2. 用小根堆维护每一个不相交区间的右端点的最大值
3. 若区间之间有交集，那么增加一个新的教室

|  |
| --- |
| #include<bits/stdc++.h> using namespace std; int n; struct SS{  int l,r; }e[100010]; bool cmp(SS x,SS y) {  return x.l<y.l; } int main() {  cin>>n;  for(int i=1;i<=n;i++)  {  int x,y;  cin>>x>>y;  e[i]={x,y};  }  sort(e+1,e+1+n,cmp);  //用小根堆来维护所有组右端点的最大值  //堆中每一个值存的是每个组的右端点的最大值   priority\_queue<int,vector<int>,greater<int>>heap;  for(int i=1;i<=n;i++)  {  auto t=e[i];  //若堆为空或者堆顶元素 >= 现在区间左端点，说明有交集，不能合并   if(heap.empty()||heap.top()>=t.l) heap.push(t.r);  else  {  //更新当前区间的右端点   heap.pop();   heap.push(t.r);  }  }  //输出组数   cout<<heap.size();  return 0;  } |

### 6.3 区间覆盖

给定 N 个闭区间 [ai,bi] 以及一个线段区间 [s,t]，请你选择尽量少的区间，将指定线段区间完全覆盖。  
输出最少区间数，如果无法完全覆盖则输出 -1。

思路：

1. 根据所有区间的左端点从小到大排序
2. 从前往后枚举每个区间，在所有能覆盖 st 的区间里，选择右端点最大的区间，然后将 st 更新为右端点的最大值

|  |
| --- |
| #include<bits/stdc++.h> using namespace std; int st,ed; int n; struct SS{  int l,r; }e[100010]; bool cmp(SS x,SS y) {  return x.l<y.l; } int main() {  cin>>st>>ed;  cin>>n;  for(int i=1;i<=n;i++)  {  int x,y;  cin>>x>>y;  e[i]={x,y};  }  sort(e+1,e+1+n,cmp);  int res=0;  for(int i=1;i<=n;i++)  {  int j=i,r=-2e8;   while(j<n&&e[j].l<=st)//寻找右端点最大的左端点能覆盖st的区间  {  r=max(r,e[j].r);  j++;  }  if(r<st)//不能覆盖区间   {  cout<<"-1";  break;  }  res++;//寻找一次，次数+1  st=r;//更新st   if(r>=ed)//已经覆盖区间   {  cout<<res;  break;  }  i=j-1;  }   return 0; } |

### 6.4 最大不相交区间数量

给定 N 个闭区间 [ai,bi]，请你在数轴上选择若干区间，使得选中的区间之间互不相交（包括端点）。  
输出可选取区间的最大数量。

思路：

1. 根据所有区间的右端点从小到大排序
2. 从前往后枚举每个区间，如果当前区间已经包含点，则 pass，否则，选择当前区间的右端点  
   3.ps: 该题实质上是区间选点的本质

|  |
| --- |
| #include<bits/stdc++.h> using namespace std; int n; struct SS{  int l,r; }e[100010]; bool cmp(SS x,SS y) {  return x.r<y.r; } int main() {  cin>>n;  for(int i=1;i<=n;i++)  {  int x,y;  cin>>x>>y;  e[i]={x,y};  }  sort(e+1,e+1+n,cmp);  int ed=-2e8,res=0;  for(int i=1;i<=n;i++)  {  if(e[i].l>ed)  {  res++;  ed=e[i].r;  }  }  cout<<res;  return 0; } |

### 6.5 摆动序列

如果相邻数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。例如 [1,7,4,9,2,5] 是一个摆动序列，差值 (6,-3,5,-7,3) 正负交替出现。  
[1,4,7,2,5] 和 [1,7,8,6,4,2,3] 不是摆动序列。  
 给定一个整数序列，返回作为摆动序列的最长子序列的长度 。通过从原始序列中删除一些（也可以不删除）元素来获得 子序列，剩下的元素保持其原始顺序。  
例： 输入：[1,7,4,9,2,5] ，输出 6  
 输入：[1,7,8,6,4,2,3] ，输出4

思路：

因为摆动序列要求正负交替出现，且数量匹配，所以维护两个变量 up 和 down，分别表示当前元素作为上升或下降趋势时的最长摆动子序列长度。  
 遍历数组 nums 的每个元素：  
 如果当前元素 nums[i] 大于前一个元素 nums[i-1]，则说明存在上升趋势，更新 up = down + 1。  
 如果 nums[i] 小于前一个元素 nums[i-1]，则存在下降趋势，更新 down = up + 1。  
最终 max(up, down) 即为最长的摆动子序列长度。

|  |
| --- |
| #include <bits/stdc++.h>  usingnamespace **std;    int maxLength(vector<int> nums) {** if **(nums.size() < 2)** return **nums.size();     int up = 1, down = 1; // 初始值设置为1，因为单个元素或相同元素序列也可以看作摆动序列** for **(int i = 1; i < nums.size(); ++i) {** if **(nums[i] > nums[i - 1]) { // 上升趋势   up = down + 1;   }** elseif **(nums[i] < nums[i - 1]) { // 下降趋势   down = up + 1;   }   }** return **max(up, down);  }    int main() {   int n;   cin>>n;   vector<int>nums(n);** for**(int i=0;i<n;i++){   cin>>nums[i];   }   cout << "最长摆动子序列长度: " << maxLength(nums) << endl;** return **0;  }** |