

代码板子：第一弹

陈彦桥

July 17, 2025

0.1 前言

本册主要包含一些比较基础的代码板子，例如基础数据结构、基础优化操作、基础图论算法和搜索、基础数学算法等。一些进阶算法的内容请见后续。

Contents

0.1	前言	2
第一章	基础优化板子	7
1.1	排序	7
1.1.1	快排	7
1.1.2	归并	7
1.2	二分	8
1.2.1	整数二分	8
1.2.2	浮点数二分	9
1.3	高精度	9
1.3.1	加	9
1.3.2	减	10
1.3.3	高精度乘低精度	10
1.3.4	高精度除低精度	11
1.4	前缀和与差分	12
1.4.1	前缀和	12
1.4.2	差分	12
1.5	位运算	12
1.5.1	求 n 的第 k 位	12
1.5.2	返回 n 的最后一位 1	12
1.5.3	n 位格雷码	13
1.6	双指针	13
1.7	离散化	13
1.8	区间合并	14
第二章	数据结构	15
2.1	链表	15
2.1.1	单链表	15
2.1.2	双链表	15
2.2	栈	16

2.2.1	一般栈	16
2.2.2	单调栈	17
2.3	队列	17
2.3.1	一般队列	17
2.3.2	循环队列	18
2.3.3	单调队列	18
2.4	并查集	19
2.4.1	朴素并查集	19
2.4.2	维护 size 的并查集（按秩合并）	19
2.4.3	维护到祖宗节点距离的并查集	20
2.5	堆	21
2.6	哈希	22
2.6.1	一般哈希	22
2.6.2	字符串哈希	23
2.7	C++STL	24
第三章	搜索与图论	27
3.1	存图	27
3.1.1	邻接矩阵	27
3.1.2	链表向前星	27
3.2	朴素搜索	28
3.2.1	DFS	28
3.2.2	BFS	28
3.2.3	拓扑排序	29
3.3	单源最短路	29
3.3.1	Dijkstra	29
3.3.2	Bellman-Ford	31
3.3.3	SPFA（尽可能避免 SPFA!）	32
3.4	全源最短路	35
3.4.1	Floyd 算法	35
3.5	最小生成树	35
3.5.1	Prim 算法	35
3.5.2	Kruskal 算法	36
3.6	二分图	38
3.6.1	染色法判定二分图：不存在奇环	38
3.6.2	匈牙利算法：二分图最小匹配	39

第四章 数学	41
4.1 质数判定	41
4.1.1 试除法	41
4.2 分解因数	41
4.2.1 试除法	41
4.3 筛法	42
4.3.1 朴素筛法求素数	42
4.3.2 线性筛法求素数	42
4.4 约数	43
4.4.1 试除法求约数	43
4.4.2 约数个数和约数之和	43
4.5 欧几里得算法：求最大公约数	43
4.5.1 普通欧几里得算法	43
4.5.2 扩展欧几里得算法	44
4.6 欧拉函数	44
4.6.1 朴素求欧拉函数	44
4.6.2 筛法求欧拉函数	45
4.7 高斯消元法	45
4.8 求组合数	47
4.8.1 递推求组合数	47
4.8.2 预处理逆元求组合数	47
4.8.3 Lucas 定理求组合数	47
4.8.4 分解质因数求组合数	49
4.8.5 卡特兰数	51
4.9 公平组合博弈 (ICG)	51
4.9.1 NIM 博弈	51
4.9.2 公平组合博弈	51

第一章 基础优化板子

1.1 排序

1.1.1 快排

```
1 void quick_sort(int q[], int l, int r)
2 {
3     if (l >= r) return;
4
5     int i = l - 1, j = r + 1, x = q[l + r >> 1];
6     while (i < j)
7     {
8         do i ++ ; while (q[i] < x);
9         do j -- ; while (q[j] > x);
10        if (i < j) swap(q[i], q[j]);
11    }
12    quick_sort(q, l, j), quick_sort(q, j + 1, r);
13 }
```

1.1.2 归并

```
1 void merge_sort(int q[], int l, int r)
2 {
3     if (l >= r) return;
4
5     int mid = l + r >> 1;
6     merge_sort(q, l, mid);
7     merge_sort(q, mid + 1, r);
8 }
```

```
9      int k = 0, i = l, j = mid + 1;
10     while (i <= mid && j <= r)
11         if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
12         else tmp[k ++ ] = q[j ++ ];
13
14     while (i <= mid) tmp[k ++ ] = q[i ++ ];
15     while (j <= r) tmp[k ++ ] = q[j ++ ];
16
17     for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
18 }
```

1.2 二分

1.2.1 整数二分

```
1  bool check(int x) { /* ... */ } // 检查 $x$ 是否满足某种性质
2
3  // 区间  $[l, r]$  被划分成  $[l, mid]$  和  $[mid + 1, r]$  时使用:
4  int bsearch_1(int l, int r)
5  {
6      while (l < r)
7      {
8          int mid = l + r >> 1;
9          if (check(mid)) r = mid; // check() 判断  $mid$  是否满足性质
10         else l = mid + 1;
11     }
12     return l;
13 }
14 // 区间  $[l, r]$  被划分成  $[l, mid - 1]$  和  $[mid, r]$  时使用:
15 int bsearch_2(int l, int r)
16 {
17     while (l < r)
18     {
19         int mid = l + r + 1 >> 1;
20         if (check(mid)) l = mid;
21         else r = mid - 1;
```



```

22     }
23     return l;
24 }

```

1.2.2 浮点数二分

```

1  bool check(double x) { /* ... */ } // 检查  $x$  是否满足某种性质
2
3  double bsearch_3(double l, double r)
4  {
5      const double eps = 1e-6; //  $eps$  表示精度, 取决于题目对精度的要求
6      while (r - l > eps)
7      {
8          double mid = (l + r) / 2;
9          if (check(mid)) r = mid;
10         else l = mid;
11     }
12     return l;
13 }

```

1.3 高精度

1.3.1 加

```

1  //  $C = A + B$ ,  $A \geq 0$ ,  $B \geq 0$ 
2  vector<int> add(vector<int> &A, vector<int> &B)
3  {
4      if (A.size() < B.size()) return add(B, A);
5
6      vector<int> C;
7      int t = 0;
8      for (int i = 0; i < A.size(); i++)
9      {
10         t += A[i];
11         if (i < B.size()) t += B[i];

```

```
12         C.push_back(t % 10);
13         t /= 10;
14     }
15
16     if (t) C.push_back(t);
17     return C;
18 }
```

1.3.2 减

```
1 // C = A - B, 满足 A >= B, A >= 0, B >= 0
2 vector<int> sub(vector<int> &A, vector<int> &B)
3 {
4     vector<int> C;
5     for (int i = 0, t = 0; i < A.size(); i ++ )
6     {
7         t = A[i] - t;
8         if (i < B.size()) t -= B[i];
9         C.push_back((t + 10) % 10);
10        if (t < 0) t = 1;
11        else t = 0;
12    }
13
14    while (C.size() > 1 && C.back() == 0) C.pop_back();
15    return C;
16 }
```

1.3.3 高精度乘低精度

```
1 // C = A * b, A >= 0, b >= 0
2 vector<int> mul(vector<int> &A, int b)
3 {
4     vector<int> C;
5
6     int t = 0;
```

```
7     for (int i = 0; i < A.size() || t; i ++ )
8     {
9         if (i < A.size()) t += A[i] * b;
10        C.push_back(t % 10);
11        t /= 10;
12    }
13
14    while (C.size() > 1 && C.back() == 0) C.pop_back();
15
16    return C;
17 }
```

1.3.4 高精度除低精度

```
1  //  $A / b = C \dots r, A \geq 0, b > 0$ 
2  vector<int> div(vector<int> &A, int b, int &r)
3  {
4      vector<int> C;
5      r = 0;
6      for (int i = A.size() - 1; i >= 0; i -- )
7      {
8          r = r * 10 + A[i];
9          C.push_back(r / b);
10         r %= b;
11     }
12     reverse(C.begin(), C.end());
13     while (C.size() > 1 && C.back() == 0) C.pop_back();
14     return C;
15 }
```

1.4 前缀和与差分

1.4.1 前缀和

```

1  \\一维前缀和
2  S[i] = a[1] + a[2] + ... a[i]
3  a[1] + ... + a[r] = S[r] - S[1 - 1]
4  \\二维前缀和
5  S[i][j] = 第i行j列格子左上部分所有元素的和
6  以(x1, y1)为左上角, (x2, y2)为右下角的子矩阵的和为:
7  S[x2][y2] - S[x1 - 1][y2] - S[x2][y1 - 1] + S[x1 - 1][y1 - 1]

```

1.4.2 差分

```

1  \\给区间内每个数加上c
2  B[1] += c, B[r + 1] -= c
3  \\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
4  S[x1][y1] += c, S[x2 + 1][y1] -= c, S[x1][y2 + 1] -= c, S[x2 + 1][y2 + 1]
    += c

```

1.5 位运算

1.5.1 求 n 的第 k 位

```

1  n >> k & 1

```

1.5.2 返回 n 的最后一位 1

```

1  lowbit(n) = n & -n

```

1.5.3 n 位格雷码

```
1 //第i个n位格雷码为
2  $i \oplus (i / 2)$ 
```

1.6 双指针

```
1 for (int i = 0, j = 0; i < n; i ++ )
2 {
3     while (j < i && check(i, j)) j ++ ;
4
5     // 具体问题的逻辑
6 }
```

1.7 离散化

```
1 vector<int> alls; // 存储所有待离散化的值
2 sort(alls.begin(), alls.end()); // 将所有值排序
3 alls.erase(unique(alls.begin(), alls.end()), alls.end()); // 去掉重复元素
4
5 // 二分求出x对应的离散化的值
6 int find(int x) // 找到第一个大于等于x的位置
7 {
8     int l = 0, r = alls.size() - 1;
9     while (l < r)
10     {
11         int mid = l + r >> 1;
12         if (alls[mid] >= x) r = mid;
13         else l = mid + 1;
14     }
15     return r + 1; // 映射到1, 2, ...n
16 }
```

1.8 区间合并

```
1 // 将所有存在交集的区间合并
2 void merge(vector<PII> &segs)
3 {
4     vector<PII> res;
5
6     sort(segs.begin(), segs.end());
7
8     int st = -2e9, ed = -2e9;
9     for (auto seg : segs)
10         if (ed < seg.first)
11         {
12             if (st != -2e9) res.push_back({st, ed});
13             st = seg.first, ed = seg.second;
14         }
15         else ed = max(ed, seg.second);
16
17     if (st != -2e9) res.push_back({st, ed});
18
19     segs = res;
20 }
```

第二章 数据结构

2.1 链表

2.1.1 单链表

```
1 // 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
2 int head, e[N], ne[N], idx;
3
4 // 初始化
5 void init()
6 {
7     head = -1;
8     idx = 0;
9 }
10
11 // 在链表头插入一个数a
12 void insert(int a)
13 {
14     e[idx] = a, ne[idx] = head, head = idx ++ ;
15 }
16
17 // 将头结点删除，需要保证头结点存在
18 void remove()
19 {
20     head = ne[head];
21 }
```

2.1.2 双链表

```
1 // 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
2 int e[N], l[N], r[N], idx;
3
4 // 初始化
5 void init()
6 {
7     //0是左端点, 1是右端点
8     r[0] = 1, l[1] = 0;
9     idx = 2;
10 }
11
12 // 在节点a的右边插入一个数x
13 void insert(int a, int x)
14 {
15     e[idx] = x;
16     l[idx] = a, r[idx] = r[a];
17     l[r[a]] = idx, r[a] = idx ++ ;
18 }
19
20 // 删除节点a
21 void remove(int a)
22 {
23     l[r[a]] = l[a];
24     r[l[a]] = r[a];
25 }
```

2.2 栈

2.2.1 一般栈

```
1 // tt表示栈顶
2 int stk[N], tt = 0;
3
4 // 向栈顶插入一个数
5 stk[ ++ tt] = x;
```



```
6
7 // 从栈顶弹出一个数
8 tt -- ;
9
10 // 栈顶的值
11 return stk[tt];
12
13 // 判断栈是否为空，如果  $tt > 0$ ，则表示不为空
14 if (tt > 0){
15     return true;
16 }
17 else{
18     return false;
19 }
```

2.2.2 单调栈

```
1 //实现代码与上述代码类似，但是维护其单调性质；用于求最近比当前数大/小的数
2 int tt = 0;
3 for (int i = 1; i <= n; i ++ )
4 {
5     while (tt && check(stk[tt], i)) tt -- ;
6     stk[ ++ tt] = i;
7 }
```

2.3 队列

2.3.1 一般队列

```
1 // hh 表示队头，tt表示队尾
2 int q[N], hh = 0, tt = -1;
3
4 // 向队尾插入一个数
5 q[ ++ tt] = x;
6
```

```
7 // 从队头弹出一个数
8 hh ++ ;
9
10 // 队头的值
11 q[hh];
12
13 // 判断队列是否为空, 如果 hh <= tt, 则表示不为空
14 if (hh <= tt)
15 {
16
17 }
```

2.3.2 循环队列

```
1 // hh 表示队头, tt表示队尾的后一个位置
2 int q[N], hh = 0, tt = 0;
3
4 // 向队尾插入一个数
5 q[tt ++ ] = x;
6 if (tt == N) tt = 0;
7
8 // 从队头弹出一个数
9 hh ++ ;
10 if (hh == N) hh = 0;
11
12 // 队头的值
13 q[hh];
14
15 // 判断队列是否为空, 如果 hh != tt, 则表示不为空
16 if (hh != tt)
17 {
18
19 }
```

2.3.3 单调队列

```
1 //维护方法；滑动窗口最小值
2 int hh = 0, tt = -1;
3 for (int i = 0; i < n; i ++ )
4 {
5     while (hh <= tt && check_out(q[hh])) hh ++ ; // 判断队头是否滑出窗口
6     while (hh <= tt && check(q[tt], i)) tt -- ;
7     q[ ++ tt] = i;
8 }
```

2.4 并查集

2.4.1 朴素并查集

```
1 int p[N]; //存储每个点的祖宗节点
2
3 // 返回x的祖宗节点
4 int find(int x)
5 {
6     if (p[x] != x) p[x] = find(p[x]);
7     return p[x];
8 }
9
10 // 初始化，假定节点编号是1~n
11 for (int i = 1; i <= n; i ++ ) p[i] = i;
12
13 // 合并a和b所在的两个集合：
14 p[find(a)] = find(b);
```

2.4.2 维护 size 的并查集（按秩合并）

```
1 int p[N], size[N];
2 //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
3
4 // 返回x的祖宗节点
```

```
5  int find(int x)
6  {
7      if (p[x] != x) p[x] = find(p[x]);
8      return p[x];
9  }
10
11 // 初始化, 假定节点编号是1~n
12 for (int i = 1; i <= n; i ++ )
13 {
14     p[i] = i;
15     size[i] = 1;
16 }
17
18 // 合并a和b所在的两个集合:
19 size[find(b)] += size[find(a)];
20 p[find(a)] = find(b);
```

2.4.3 维护到祖宗节点距离的并查集

```
1  int p[N], d[N];
2  //p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离
3
4  // 返回x的祖宗节点
5  int find(int x)
6  {
7      if (p[x] != x)
8      {
9          int u = find(p[x]);
10         d[x] += d[p[x]];
11         p[x] = u;
12     }
13     return p[x];
14 }
15
16 // 初始化, 假定节点编号是1~n
17 for (int i = 1; i <= n; i ++ )
```

```
18 {
19     p[i] = i;
20     d[i] = 0;
21 }
22
23 // 合并a和b所在的两个集合:
24 p[find(a)] = find(b);
25 d[find(a)] = distance; // 根据具体问题, 初始化find(a)的偏移量
```

2.5 堆

```
1 // 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
2 // ph[k]存储第k个插入的点在堆中的位置
3 // hp[k]存储堆中下标是k的点是第几个插入的
4 int h[N], ph[N], hp[N], size;
5
6 // 交换两个点, 及其映射关系
7 void heap_swap(int a, int b)
8 {
9     swap(ph[hp[a]], ph[hp[b]]);
10    swap(hp[a], hp[b]);
11    swap(h[a], h[b]);
12 }
13
14 void down(int u)
15 {
16     int t = u;
17     if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
18     if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
19     if (u != t)
20     {
21         heap_swap(u, t);
22         down(t);
23     }
24 }
25
```

```
26 void up(int u)
27 {
28     while (u / 2 && h[u] < h[u / 2])
29     {
30         heap_swap(u, u / 2);
31         u >>= 1;
32     }
33 }
34
35 //  $O(n)$ 建堆
36 for (int i = n / 2; i; i -- ) down(i);
```

2.6 哈希

2.6.1 一般哈希

```
1 //拉链法
2 int h[N], e[N], ne[N], idx;
3
4 // 向哈希表中插入一个数
5 void insert(int x)
6 {
7     int k = (x % N + N) % N;
8     e[idx] = x;
9     ne[idx] = h[k];
10    h[k] = idx ++ ;
11 }
12
13 // 在哈希表中查询某个数是否存在
14 bool find(int x)
15 {
16     int k = (x % N + N) % N;
17     for (int i = h[k]; i != -1; i = ne[i])
18         if (e[i] == x)
19             return true;
20 }
```

```
21     return false;
22 }
```

```
1 //开放寻址法
2 int h[N];
3
4 // 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
5 int find(int x)
6 {
7     int t = (x % N + N) % N;
8     while (h[t] != null && h[t] != x)
9     {
10         t ++ ;
11         if (t == N) t = 0;
12     }
13     return t;
14 }
```

2.6.2 字符串哈希

```
1 //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
2 //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
3
4 typedef unsigned long long ULL;
5 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  $P^k \bmod 2^{64}$ 
6
7 // 初始化
8 p[0] = 1;
9 for (int i = 1; i <= n; i ++ )
10 {
11     h[i] = h[i - 1] * P + str[i];
12     p[i] = p[i - 1] * P;
13 }
14
```

```
15 // 计算子串 str[l ~ r] 的哈希值
16 ULL get(int l, int r)
17 {
18     return h[r] - h[l - 1] * p[r - l + 1];
19 }
```

2.7 C++STL

```
1  vector, 变长数组, 倍增的思想
2      size() 返回元素个数
3      empty() 返回是否为空
4      clear() 清空
5      front()/back()
6      push_back()/pop_back()
7      begin()/end()
8      []
9      支持比较运算, 按字典序
10
11  pair<int, int>
12      first, 第一个元素
13      second, 第二个元素
14      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)
15
16  string, 字符串
17      size()/length() 返回字符串长度
18      empty()
19      clear()
20      substr(起始下标, (子串长度)) 返回子串
21      c_str() 返回字符串所在字符数组的起始地址
22
23  queue, 队列
24      size()
25      empty()
26      push() 向队尾插入一个元素
27      front() 返回队头元素
28      back() 返回队尾元素
```



```
29     pop() 弹出队头元素
30
31 priority_queue, 优先队列, 默认是大根堆
32     size()
33     empty()
34     push() 插入一个元素
35     top() 返回堆顶元素
36     pop() 弹出堆顶元素
37     定义成小根堆的方式: priority_queue<int, vector<int>, greater<int>> q;
38
39 stack, 栈
40     size()
41     empty()
42     push() 向栈顶插入一个元素
43     top() 返回栈顶元素
44     pop() 弹出栈顶元素
45
46 deque, 双端队列
47     size()
48     empty()
49     clear()
50     front()/back()
51     push_back()/pop_back()
52     push_front()/pop_front()
53     begin()/end()
54     []
55
56 set, map, multiset, multimap, 基于平衡二叉树 (红黑树), 动态维护有序序列
57     size()
58     empty()
59     clear()
60     begin()/end()
61     ++, -- 返回前驱和后继, 时间复杂度  $O(\log n)$ 
62
63 set/multiset
64     insert() 插入一个数
65     find() 查找一个数
```

```

66     count() 返回某一个数的个数
67     erase()
68         (1) 输入是一个数x, 删除所有x  $O(k + \log n)$ 
69         (2) 输入一个迭代器, 删除这个迭代器
70     lower_bound()/upper_bound()
71     lower_bound(x) 返回大于等于x的最小的数的迭代器
72     upper_bound(x) 返回大于x的最小的数的迭代器
73 map/multimap
74     insert() 插入的数是一个pair
75     erase() 输入的参数是pair或者迭代器
76     find()
77     [] 注意multimap不支持此操作。 时间复杂度是  $O(\log n)$ 
78     lower_bound()/upper_bound()
79
80 unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
81 和上面类似, 增删改查的时间复杂度是  $O(1)$ 
82 不支持 lower_bound()/upper_bound(), 迭代器的++, --
83
84 bitset, 压位
85     bitset<10000> s;
86     ~, &, |, ^
87     >>, <<
88     ==, !=
89     []
90
91     count() 返回有多少个1
92
93     any() 判断是否至少有一个1
94     none() 判断是否全为0
95
96     set() 把所有位置成1
97     set(k, v) 将第k位变成v
98     reset() 把所有位变成0
99     flip() 等价于~
100    flip(k) 把第k位取反

```

第三章 搜索与图论

3.1 存图

3.1.1 邻接矩阵

```
1 //二维矩阵储存点之间的连接
2 g[a][b] = w;
```

3.1.2 链表向前星

```
1 // 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
2 int h[N], e[N], ne[N], idx;
3
4 // 添加一条边a->b
5 void add(int a, int b)
6 {
7     e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
8 }
9
10 // 初始化
11 idx = 0;
12 memset(h, -1, sizeof h);
```

3.2 朴素搜索

3.2.1 DFS

```
1  int dfs(int u)
2  {
3      st[u] = true; // st[u] 表示点u已经被遍历过
4
5      for (int i = h[u]; i != -1; i = ne[i])
6      {
7          int j = e[i];
8          if (!st[j]) dfs(j);
9      }
10 }
```

3.2.2 BFS

```
1  queue<int> q;
2  st[1] = true; // 表示1号点已经被遍历过
3  q.push(1);
4
5  while (q.size())
6  {
7      int t = q.front();
8      q.pop();
9
10     for (int i = h[t]; i != -1; i = ne[i])
11     {
12         int j = e[i];
13         if (!st[j])
14         {
15             st[j] = true; // 表示点j已经被遍历过
16             q.push(j);
17         }
18     }
19 }
```

3.2.3 拓扑排序

```
1  bool topsort()
2  {
3      int hh = 0, tt = -1;
4
5      // d[i] 存储点 i 的入度
6      for (int i = 1; i <= n; i ++ )
7          if (!d[i])
8              q[ ++ tt] = i;
9
10     while (hh <= tt)
11     {
12         int t = q[hh ++ ];
13
14         for (int i = h[t]; i != -1; i = ne[i])
15         {
16             int j = e[i];
17             if (-- d[j] == 0)
18                 q[ ++ tt] = j;
19         }
20     }
21
22     // 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。
23     return tt == n - 1;
24 }
```

3.3 单源最短路

3.3.1 Dijkstra

```
1  //朴素Dijkstra
2  int g[N][N]; // 存储每条边
3  int dist[N]; // 存储1号点到每个点的最短距离
4  bool st[N]; // 存储每个点的最短路是否已经确定
5
```

```

6 // 求1号点到n号点的最短路, 如果不存在则返回-1
7 int dijkstra()
8 {
9     memset(dist, 0x3f, sizeof dist);
10    dist[1] = 0;
11
12    for (int i = 0; i < n - 1; i ++ )
13    {
14        int t = -1;    // 在还未确定最短路的点中, 寻找距离最小的点
15        for (int j = 1; j <= n; j ++ )
16            if (!st[j] && (t == -1 || dist[t] > dist[j]))
17                t = j;
18
19        // 用t更新其他点的距离
20        for (int j = 1; j <= n; j ++ )
21            dist[j] = min(dist[j], dist[t] + g[t][j]);
22
23        st[t] = true;
24    }
25
26    if (dist[n] == 0x3f3f3f3f) return -1;
27    return dist[n];
28 }

```

```

1 //堆优化版本Dijkstra
2 typedef pair<int, int> PII;
3
4 int n;    // 点的数量
5 int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边
6 int dist[N];    // 存储所有点到1号点的距离
7 bool st[N];    // 存储每个点的最短距离是否已确定
8
9 // 求1号点到n号点的最短距离, 如果不存在, 则返回-1
10 int dijkstra()
11 {
12     memset(dist, 0x3f, sizeof dist);
13     dist[1] = 0;

```

```
14     priority_queue<PII, vector<PII>, greater<PII>> heap;
15     heap.push({0, 1});    // first存储距离, second存储节点编号
16
17     while (heap.size())
18     {
19         auto t = heap.top();
20         heap.pop();
21
22         int ver = t.second, distance = t.first;
23
24         if (st[ver]) continue;
25         st[ver] = true;
26
27         for (int i = h[ver]; i != -1; i = ne[i])
28         {
29             int j = e[i];
30             if (dist[j] > distance + w[i])
31             {
32                 dist[j] = distance + w[i];
33                 heap.push({dist[j], j});
34             }
35         }
36     }
37
38     if (dist[n] == 0x3f3f3f3f) return -1;
39     return dist[n];
40 }
```

3.3.2 Bellman-Ford

```
1     int n, m;    // n表示点数, m表示边数
2     int dist[N]; // dist[x]存储1到x的最短路距离
3
4     struct Edge // 边, a表示出点, b表示入点, w表示边的权重
5     {
6         int a, b, w;
```

```

7  }edges[M];
8
9  // 求1到n的最短路距离, 如果无法从1走到n, 则返回-1。
10 int bellman_ford()
11 {
12     memset(dist, 0x3f, sizeof dist);
13     dist[1] = 0;
14
15     // 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。
16     for (int i = 0; i < n; i ++ )
17     {
18         for (int j = 0; j < m; j ++ )
19         {
20             int a = edges[j].a, b = edges[j].b, w = edges[j].w;
21             if (dist[b] > dist[a] + w)
22                 dist[b] = dist[a] + w;
23         }
24     }
25
26     if (dist[n] > 0x3f3f3f3f / 2) return -1;
27     return dist[n];
28 }

```

3.3.3 SPFA (尽可能避免 SPFA!)

```

1  //SPFA查找最短路
2  int n;    // 总点数
3  int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边
4  int dist[N]; // 存储每个点到1号点的最短距离
5  bool st[N]; // 存储每个点是否在队列中
6
7  // 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
8  int spfa()
9  {
10     memset(dist, 0x3f, sizeof dist);

```



```

11     dist[1] = 0;
12
13     queue<int> q;
14     q.push(1);
15     st[1] = true;
16
17     while (q.size())
18     {
19         auto t = q.front();
20         q.pop();
21
22         st[t] = false;
23
24         for (int i = h[t]; i != -1; i = ne[i])
25         {
26             int j = e[i];
27             if (dist[j] > dist[t] + w[i])
28             {
29                 dist[j] = dist[t] + w[i];
30                 if (!st[j])    // 如果队列中已存在j, 则不需要将j重复插入
31                 {
32                     q.push(j);
33                     st[j] = true;
34                 }
35             }
36         }
37     }
38
39     if (dist[n] == 0x3f3f3f3f) return -1;
40     return dist[n];
41 }

```

```

1  //SPFA判定负权环
2  int n;    // 总点数
3  int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边
4  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

```



```

40         }
41     }
42 }
43
44 return false;
45 }

```

3.4 全源最短路

3.4.1 Floyd 算法

```

1 //初始化:
2 for (int i = 1; i <= n; i ++ )
3     for (int j = 1; j <= n; j ++ )
4         if (i == j) d[i][j] = 0;
5         else d[i][j] = INF;
6
7 // 算法结束后, d[a][b]表示a到b的最短距离
8 void floyd()
9 {
10     for (int k = 1; k <= n; k ++ )
11         for (int i = 1; i <= n; i ++ )
12             for (int j = 1; j <= n; j ++ )
13                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
14 }

```

3.5 最小生成树

3.5.1 Prim 算法

```

1 int n;    // n表示点数
2 int g[N][N];    // 邻接矩阵, 存储所有边
3 int dist[N];    // 存储其他点到当前最小生成树的距离
4 bool st[N];    // 存储每个点是否已经在生成树中
5

```

```

6
7 // 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
8 int prim()
9 {
10     memset(dist, 0x3f, sizeof dist);
11
12     int res = 0;
13     for (int i = 0; i < n; i ++ )
14     {
15         int t = -1;
16         for (int j = 1; j <= n; j ++ )
17             if (!st[j] && (t == -1 || dist[t] > dist[j]))
18                 t = j;
19
20         if (i && dist[t] == INF) return INF;
21
22         if (i) res += dist[t];
23         st[t] = true;
24
25         for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
26     }
27
28     return res;
29 }

```

3.5.2 Kruskal 算法

```

1 int n, m;    // n是点数, m是边数
2 int p[N];    // 并查集的父节点数组
3
4 struct Edge  // 存储边
5 {
6     int a, b, w;
7
8     bool operator< (const Edge &W) const

```

```
9      {
10          return w < W.w;
11      }
12  }edges[M];
13
14  int find(int x)    // 并查集核心操作
15  {
16      if (p[x] != x) p[x] = find(p[x]);
17      return p[x];
18  }
19
20  int kruskal()
21  {
22      sort(edges, edges + m);
23
24      for (int i = 1; i <= n; i ++ ) p[i] = i; // 初始化并查集
25
26      int res = 0, cnt = 0;
27      for (int i = 0; i < m; i ++ )
28      {
29          int a = edges[i].a, b = edges[i].b, w = edges[i].w;
30
31          a = find(a), b = find(b);
32          if (a != b)    // 如果两个连通块不连通，则将这两个连通块合并
33          {
34              p[a] = b;
35              res += w;
36              cnt ++ ;
37          }
38      }
39
40      if (cnt < n - 1) return INF;
41      return res;
42  }
```

3.6 二分图

3.6.1 染色法判定二分图：不存在奇环

```
1  int n;    // n表示点数
2  int h[N], e[M], ne[M], idx; // 邻接表存储图
3  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
4
5  // 参数: u表示当前节点, c表示当前点的颜色
6  bool dfs(int u, int c)
7  {
8      color[u] = c;
9      for (int i = h[u]; i != -1; i = ne[i])
10     {
11         int j = e[i];
12         if (color[j] == -1)
13         {
14             if (!dfs(j, !c)) return false;
15         }
16         else if (color[j] == c) return false;
17     }
18
19     return true;
20 }
21
22 bool check()
23 {
24     memset(color, -1, sizeof color);
25     bool flag = true;
26     for (int i = 1; i <= n; i ++ )
27         if (color[i] == -1)
28             if (!dfs(i, 0))
29             {
30                 flag = false;
31                 break;
32             }
33     return flag;
```

```
34 }
```

3.6.2 匈牙利算法：二分图最小匹配

```
1  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
2  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
3  int match[N];   // 存储第二个集合中的每个点当前匹配的的第一个集合中的点是哪
    个
4  bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过
5
6  bool find(int x)
7  {
8      for(int i = h[x]; i != -1; i = ne[i])
9      {
10         int j = e[i];
11         if (!st[j])
12         {
13             st[j] = true;
14             if (match[j] == 0 || find(match[j]))
15             {
16                 match[j] = x;
17                 return true;
18             }
19         }
20     }
21
22     return false;
23 }
24
25 // 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
26 int res = 0;
27 for (int i = 1; i <= n1; i ++ )
28 {
29     memset(st, false, sizeof st);
30     if (find(i)) res ++ ;
```

31

}

第四章 数学

4.1 质数判定

4.1.1 试除法

```
1  bool is_prime(int x)
2  {
3      if (x < 2) return false;
4      for (int i = 2; i <= x / i; i ++ )
5          if (x % i == 0)
6              return false;
7      return true;
8  }
```

4.2 分解因数

4.2.1 试除法

```
1  void divide(int x)
2  {
3      for (int i = 2; i <= x / i; i ++ )
4          if (x % i == 0)
5              {
6                  int s = 0;
7                  while (x % i == 0) x /= i, s ++ ;
8                  cout << i << ' ' << s << endl;
9              }
10     if (x > 1) cout << x << ' ' << 1 << endl;
11     cout << endl;
```

```
12 }
```

4.3 筛法

4.3.1 朴素筛法求素数

```
1  int primes[N], cnt; // primes[]存储所有素数
2  bool st[N];         // st[x]存储x是否被筛掉
3
4  void get_primes(int n)
5  {
6      for (int i = 2; i <= n; i ++ )
7      {
8          if (st[i]) continue;
9          primes[cnt ++ ] = i;
10         for (int j = i + i; j <= n; j += i)
11             st[j] = true;
12     }
13 }
```

4.3.2 线性筛法求素数

```
1  int primes[N], cnt; // primes[]存储所有素数
2  bool st[N];         // st[x]存储x是否被筛掉
3
4  void get_primes(int n)
5  {
6      for (int i = 2; i <= n; i ++ )
7      {
8          if (!st[i]) primes[cnt ++ ] = i;
9          for (int j = 0; primes[j] <= n / i; j ++ )
10             {
11                 st[primes[j] * i] = true;
12                 if (i % primes[j] == 0) break;
13             }
14     }
```

```
14     }  
15 }
```

4.4 约数

4.4.1 试除法求约数

```
1  vector<int> get_divisors(int x)  
2  {  
3      vector<int> res;  
4      for (int i = 1; i <= x / i; i ++ )  
5          if (x % i == 0)  
6              {  
7                  res.push_back(i);  
8                  if (i != x / i) res.push_back(x / i);  
9              }  
10     sort(res.begin(), res.end());  
11     return res;  
12 }
```

4.4.2 约数个数和约数之和

```
1  //如果  $N = p_1^{c_1} * p_2^{c_2} * \dots * p_k^{c_k}$   
2  //约数个数  
3   $(c_1 + 1) * (c_2 + 1) * \dots * (c_k + 1)$   
4  //约数之和  
5   $(p_1^0 + p_1^1 + \dots + p_1^{c_1}) * \dots * (p_k^0 + p_k^1 + \dots + p_k^{c_k})$ 
```

4.5 欧几里得算法：求最大公约数

4.5.1 普通欧几里得算法

```
1  int gcd(int a, int b)  
2  {
```

```
3     return b ? gcd(b, a % b) : a;
4 }
```

4.5.2 扩展欧几里得算法

```
1 // 求  $x, y$ , 使得  $ax + by = \gcd(a, b)$ 
2 int exgcd(int a, int b, int &x, int &y)
3 {
4     if (!b)
5     {
6         x = 1; y = 0;
7         return a;
8     }
9     int d = exgcd(b, a % b, y, x);
10    y -= (a/b) * x;
11    return d;
12 }
```

4.6 欧拉函数

4.6.1 朴素求欧拉函数

```
1 int phi(int x)
2 {
3     int res = x;
4     for (int i = 2; i <= x / i; i++)
5         if (x % i == 0)
6         {
7             res = res / i * (i - 1);
8             while (x % i == 0) x /= i;
9         }
10    if (x > 1) res = res / x * (x - 1);
11
12    return res;
13 }
```

4.6.2 筛法求欧拉函数

```
1  int primes[N], cnt; // primes[]存储所有素数
2  int euler[N];       // 存储每个数的欧拉函数
3  bool st[N];         // st[x]存储x是否被筛掉
4
5
6  void get_eulers(int n)
7  {
8      euler[1] = 1;
9      for (int i = 2; i <= n; i ++ )
10     {
11         if (!st[i])
12         {
13             primes[cnt ++ ] = i;
14             euler[i] = i - 1;
15         }
16         for (int j = 0; primes[j] <= n / i; j ++ )
17         {
18             int t = primes[j] * i;
19             st[t] = true;
20             if (i % primes[j] == 0)
21             {
22                 euler[t] = euler[i] * primes[j];
23                 break;
24             }
25             euler[t] = euler[i] * (primes[j] - 1);
26         }
27     }
28 }
```

4.7 高斯消元法

```
1  // a[N][N]是增广矩阵
2  int gauss()
3  {
```

```
4      int c, r;
5      for (c = 0, r = 0; c < n; c ++ )
6      {
7          int t = r;
8          for (int i = r; i < n; i ++ ) // 找到绝对值最大的行
9              if (fabs(a[i][c]) > fabs(a[t][c]))
10                 t = i;
11
12         if (fabs(a[t][c]) < eps) continue;
13
14         for (int i = c; i <= n; i ++ ) swap(a[t][i], a[r][i]); // 将绝对值
15             最大的行换到最顶端
16         for (int i = n; i >= c; i -- ) a[r][i] /= a[r][c]; // 将当前行的首
17             位变成1
18         for (int i = r + 1; i < n; i ++ ) // 用当前行将下面所有的列消成0
19             if (fabs(a[i][c]) > eps)
20                 for (int j = n; j >= c; j -- )
21                     a[i][j] -= a[r][j] * a[i][c];
22
23         r ++ ;
24     }
25
26     if (r < n)
27     {
28         for (int i = r; i < n; i ++ )
29             if (fabs(a[i][n]) > eps)
30                 return 2; // 无解
31         return 1; // 有无穷多组解
32     }
33
34     for (int i = n - 1; i >= 0; i -- )
35         for (int j = i + 1; j < n; j ++ )
36             a[i][n] -= a[i][j] * a[j][n];
37
38     return 0; // 有唯一解
39 }
```

4.8 求组合数

4.8.1 递推求组合数

```
1 // c[a][b] 表示从a个苹果中选b个的方案数
2 for (int i = 0; i < N; i ++ )
3     for (int j = 0; j <= i; j ++ )
4         if (!j) c[i][j] = 1;
5         else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
```

4.8.2 预处理逆元求组合数

```
1 //首先预处理出所有阶乘取模的余数fact[N], 以及所有阶乘取模的逆元infact[N]
2 //如果取模的数是质数, 可以用费马小定理求逆元
3 int qmi(int a, int k, int p) // 快速幂模板
4 {
5     int res = 1;
6     while (k)
7     {
8         if (k & 1) res = (LL)res * a % p;
9         a = (LL)a * a % p;
10        k >>= 1;
11    }
12    return res;
13 }
14
15 // 预处理阶乘的余数和阶乘逆元的余数
16 fact[0] = infact[0] = 1;
17 for (int i = 1; i < N; i ++ )
18 {
19     fact[i] = (LL)fact[i - 1] * i % mod;
20     infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
21 }
```

4.8.3 Lucas 定理求组合数

```
1 //若p是质数, 则对于任意整数  $1 \leq m \leq n$ , 有:
2 //  $C(n, m) = C(n \% p, m \% p) * C(n / p, m / p) \pmod p$ 
3
4 int qmi(int a, int k, int p) // 快速幂模板
5 {
6     int res = 1 % p;
7     while (k)
8     {
9         if (k & 1) res = (LL)res * a % p;
10        a = (LL)a * a % p;
11        k >>= 1;
12    }
13    return res;
14 }
15
16 int C(int a, int b, int p) // 通过定理求组合数  $C(a, b)$ 
17 {
18     if (a < b) return 0;
19
20     LL x = 1, y = 1; // x是分子, y是分母
21     for (int i = a, j = 1; j <= b; i --, j ++ )
22     {
23         x = (LL)x * i % p;
24         y = (LL) y * j % p;
25     }
26
27     return x * (LL)qmi(y, p - 2, p) % p;
28 }
29
30 int lucas(LL a, LL b, int p)
31 {
32     if (a < p && b < p) return C(a, b, p);
33     return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
34 }
```


4.8.4 分解质因数求组合数

```
1 //当我们要求出组合数的真实值，而非对某个数的余数时，分解质因数的方式比较
  好用：
2 //1. 筛法求出范围内的所有质数
3 //2. 通过  $C(a, b) = a! / b! / (a - b)!$  这个公式求出每个质因子的次数。  $n!$ 
  中  $p$  的次数是  $n / p + n / p^2 + n / p^3 + \dots$ 
4 //3. 用高精度乘法将所有质因子相乘
5
6 int primes[N], cnt; // 存储所有质数
7 int sum[N]; // 存储每个质数的次数
8 bool st[N]; // 存储每个数是否已被筛掉
9
10
11 void get_primes(int n) // 线性筛法求素数
12 {
13     for (int i = 2; i <= n; i++)
14     {
15         if (!st[i]) primes[cnt++] = i;
16         for (int j = 0; primes[j] <= n / i; j++)
17         {
18             st[primes[j] * i] = true;
19             if (i % primes[j] == 0) break;
20         }
21     }
22 }
23
24
25 int get(int n, int p) // 求  $n!$  中的次数
26 {
27     int res = 0;
28     while (n)
29     {
30         res += n / p;
31         n /= p;
32     }
33     return res;
```

```
34 }
35
36
37 vector<int> mul(vector<int> a, int b) // 高精度乘低精度模板
38 {
39     vector<int> c;
40     int t = 0;
41     for (int i = 0; i < a.size(); i++)
42     {
43         t += a[i] * b;
44         c.push_back(t % 10);
45         t /= 10;
46     }
47
48     while (t)
49     {
50         c.push_back(t % 10);
51         t /= 10;
52     }
53
54     return c;
55 }
56
57 get_primes(a); // 预处理范围内的所有质数
58
59 for (int i = 0; i < cnt; i++) // 求每个质因数的次数
60 {
61     int p = primes[i];
62     sum[i] = get(a, p) - get(b, p) - get(a - b, p);
63 }
64
65 vector<int> res;
66 res.push_back(1);
67
68 for (int i = 0; i < cnt; i++) // 用高精度乘法将所有质因子相乘
69     for (int j = 0; j < sum[i]; j++)
70         res = mul(res, primes[i]);
```

4.8.5 卡特兰数

- 1 //给定 n 个0和 n 个1, 它们按照某种顺序排成长度为 $2n$ 的序列, 满足任意前缀中0的个数
都不少于1的个数的序列的数量为
- 2 $\text{Cat}(n) = C(2n, n) / (n + 1);$

4.9 公平组合博弈 (ICG)

4.9.1 NIM 博弈

给定 N 堆物品, 第 i 堆物品有 A_i 个。两名玩家轮流行动, 每次可以任选一堆, 取走任意多个物品, 可把一堆取光, 但不能不取。取走最后一件物品者获胜。两人都采取最优策略, 问先手是否必胜。我们把这种游戏称为 NIM 博弈。把游戏过程中面临的状态称为局面。整局游戏第一个行动的称为先手, 第二个行动的称为后手。若在某一局面下无论采取何种行动, 都会输掉游戏, 则称该局面必败。所谓采取最优策略是指, 若在某一局面下存在某种行动, 使得行动后对面面临必败局面, 则优先采取该行动。同时, 这样的局面被称为必胜。我们讨论的博弈问题一般都只考虑理想情况, 即两人都无失误, 都采取最优策略行动时游戏的结果。NIM 博弈不存在平局, 只有先手必胜和先手必败两种情况。

Theorem 4.9.1. NIM 博弈先手必胜, 当且仅当 $A_1 \oplus A_2 \oplus \dots \oplus A_n \neq 0$

4.9.2 公平组合博弈

若一个游戏满足:

- 由两名玩家交替行动;
- 在游戏进程的任意时刻, 可以执行的合法行动与轮到哪名玩家无关;
- 不能行动的玩家判负;

则称该游戏为一个公平组合游戏。NIM 博弈属于公平组合游戏, 但城建的棋类游戏, 比如围棋, 就不是公平组合游戏。因为围棋交战双方分别只能落黑子和白子, 胜负判定也比较复杂, 不满足条件 2 和条件 3。