**此章收录算法模板**

**感谢acwing，感谢y总**

**高精度加法**

vector<int> add(vector<int> &A, vector<int> &B) *// C = A + B, A >= 0, B >= 0*

{

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**

vector<int> sub(vector<int> &A, vector<int> &B) *// C = A - B, 满足A >= B, A >= 0, B >= 0*

{

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**

vector<int> mul(vector<int> &A, int b) // C = A \* b, A >= 0, b >= 0

{

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**

vector<int> div(vector<int> &A, int b, int &r) // A / b = C ... r, A >= 0, b > 0

{

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;

}

**lowbit运算**

int lowbit(int x) *// 返回末尾的1*

{

return x & -x;

}

**马拉车算法**

void init() *// a[]为原串，b[]为插入'#'后的新串*

{

int k = 0;

b[k ++ ] = '$', b[k ++ ] = '#';

for (int i = 0; i < n; i ++ ) b[k ++ ] = a[i], b[k ++ ] = '#';

b[k ++ ] = '^';

n = k;

}

void manacher() *// 马拉车算法，b[]为插入'#'后的新串*

{

int mr = 0, mid;

for (int i = 1; i < n; i ++ )

{

if (i < mr) p[i] = min(p[mid \* 2 - i], mr - i);

else p[i] = 1;

while (b[i - p[i]] == b[i + p[i]]) p[i] ++ ;

if (i + p[i] > mr)

{

mr = i + p[i];

mid = i;

}

}

}

**归并排序**

void merge\_sort(int q[], int l, int r) *// 归并排序*

{

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];

}

**DLX重复覆盖**

int l[N], r[N], u[N], d[N], col[N], row[N], s[N], idx;

int ans[N], top; *// 记录选择了哪些行*

bool st[M]; *// N为节点数，M为列数*

void init() *// 初始化十字链表*

{

for (int i = 0; i <= m; i ++ )

{

l[i] = i - 1, r[i] = i + 1;

u[i] = d[i] = i;

s[i] = 0, col[i] = i;

}

l[0] = m, r[m] = 0;

idx = m + 1;

}

void add(int& hh, int& tt, int x, int y) *// 在十字链表中插入节点*

{

row[idx] = x, col[idx] = y, s[y] ++ ;

u[idx] = y, d[idx] = d[y], u[d[y]] = idx, d[y] = idx;

r[hh] = l[tt] = idx, r[idx] = tt, l[idx] = hh;

tt = idx ++ ;

}

int h() *// IDA\*的启发函数*

{

int res = 0;

memset(st, 0, sizeof st);

for (int i = r[0]; i; i = r[i])

{

if (st[col[i]]) continue;

res ++ ;

st[col[i]] = true;

for (int j = d[i]; j != i; j = d[j])

for (int k = r[j]; k != j; k = r[k])

st[col[k]] = true;

}

return res;

}

void remove(int p)

{

for (int i = d[p]; i != p; i = d[i])

{

r[l[i]] = r[i];

l[r[i]] = l[i];

}

}

void resume(int p)

{

for (int i = u[p]; i != p; i = u[i])

{

r[l[i]] = i;

l[r[i]] = i;

}

}

bool dfs(int k)

{

if (k + h() > top) return false;

if (!r[0])

{

top = k;

return true;

}

int p = r[0];

for (int i = r[0]; i; i = r[i])

if (s[i] < s[p])

p = i;

for (int i = d[p]; i != p; i = d[i])

{

ans[k] = row[i];

remove(i);

for (int j = r[i]; j != i; j = r[j]) remove(j);

if (dfs(k + 1)) return true;

for (int j = l[i]; j != i; j = l[j]) resume(j);

resume(i);

}

return false;

}

**DLX精确覆盖**

int l[N], r[N], u[N], d[N], col[N], row[N], s[N], idx;

int ans[N], top; *// 记录选择了哪些行*

void init() *// 初始化十字链表*

{

for (int i = 0; i <= m; i ++ )

{

l[i] = i - 1, r[i] = i + 1;

u[i] = d[i] = i;

}

l[0] = m, r[m] = 0;

idx = m + 1;

}

void add(int& hh, int& tt, int x, int y) *// 在十字链表中添加节点*

{

row[idx] = x, col[idx] = y, s[y] ++ ;

u[idx] = y, d[idx] = d[y], u[d[y]] = idx, d[y] = idx;

r[hh] = l[tt] = idx, r[idx] = tt, l[idx] = hh;

tt = idx ++ ;

}

void remove(int p)

{

r[l[p]] = r[p], l[r[p]] = l[p];

for (int i = d[p]; i != p; i = d[i])

for (int j = r[i]; j != i; j = r[j])

{

s[col[j]] -- ;

d[u[j]] = d[j], u[d[j]] = u[j];

}

}

void resume(int p)

{

for (int i = d[p]; i != p; i = d[i])

for (int j = r[i]; j != i; j = r[j])

{

s[col[j]] ++ ;

d[u[j]] = j, u[d[j]] = j;

}

r[l[p]] = p, l[r[p]] = p;

}

bool dfs()

{

if (!r[0]) return true;

int p = r[0];

for (int i = r[0]; i; i = r[i])

if (s[i] < s[p])

p = i;

if (!s[p]) return false;

remove(p);

for (int i = d[p]; i != p; i = d[i])

{

ans[ ++ top] = row[i];

for (int j = r[i]; j != i; j = r[j]) remove(col[j]);

if (dfs()) return true;

for (int j = r[i]; j != i; j = r[j]) resume(col[j]);

top -- ;

}

resume(p);

return false;

}

**并查集 + 路径压缩**

int find(int x) *// 并查集*

{

if (p[x] != x) p[x] = find(p[x]);

return p[x];

}

**字符串哈希**

ULL get(int l, int r) *// 计算子串 str[l ~ r] 的哈希值*

{

return h[r] - h[l - 1] \* p[r - l + 1];

}

**Trie插入**

int son[N][26], cnt[N], idx;

void insert(char \*str) *// 插入字符串*

{

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] ++ ;

}

int query(char \*str) *// 查询字符串出现次数*

{

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];

}

**邻接链表（无权）**

void add(int a,int b)

{

e[idx] = b, next[idx] = h[a], h[a] = idx++;

}

**邻接链表（带权）**

void add(int a,int b,int c)

{

e[idx] = b, next[idx] = h[a], w[idx] = c, h[a] = idx++;

}

**dijkstra算法**

int dijkstra() *// 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*

{

memset(dist, 0x3f, sizeof dist);

dist[1] = 0;

priority\_queue<PII, vector<PII>, greater<PII>> heap;

heap.push({0, 1});

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] > dist[ver] + w[i])

{

dist[j] = dist[ver] + w[i];

heap.push({dist[j], j});

}

}

}

if (dist[n] == 0x3f3f3f3f) return -1;

return dist[n];

}

void dijkstra() *// 求1号点到n号点的最短路距离*

{

memset(dist, 0x3f, sizeof dist);

dist[1] = 0;

priority\_queue<PII, vector<PII>, greater<PII>> heap;

heap.push({0, 1});

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] > dist[ver] + w[i])

{

dist[j] = dist[ver] + w[i];

heap.push({dist[j], j});

}

}

}

}

**匈牙利算法（NTR算法）**

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;

}

**spfa算法（最短路）**

int spfa() *// 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 hh = 0, tt = 0;

memset(dist, 0x3f, sizeof dist);

dist[1] = 0;

q[tt ++ ] = 1;

st[1] = true;

while (hh != tt)

{

int t = q[hh ++ ];

if (hh == N) hh = 0;

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[tt ++ ] = j;

if (tt == N) tt = 0;

st[j] = true;

}

}

}

}

if (dist[n] == 0x3f3f3f3f) return -1;

return dist[n];

}

**spfa算法（判断负环）**

bool spfa() *// 如果存在负环，则返回true，否则返回false。*

{

*// 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。*

int hh = 0, tt = 0;

for (int i = 1; i <= n; i ++ ) q[tt ++ ] = i, st[i] = true;

while (hh != tt)

{

int t = q[hh ++ ];

if (hh == N) hh = 0;

st[t] = false;

for (int i = h[t]; ~i; 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;

if (!st[j])

{

st[j] = true;

q[tt ++ ] = j;

if (tt == N) tt = 0;

}

}

}

}

return false;

}

**括扑排序**

void 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;

}

}

}

**欧拉函数**

int phi(int x) *// 欧拉函数*

{

int res = x;

for (int i = 2; i <= x / i; i ++ )

if (x % i == 0)

{

res = res / i \* (i - 1);

while (x % i == 0) x /= i;

}

if (x > 1) res = res / x \* (x - 1);

return res;

}

**线性筛 + 欧拉函数**

void get\_eulers(int n) *// 线性筛法求1~n的欧拉函数*

{

euler[1] = 1;

for (int i = 2; i <= n; i ++ )

{

if (!st[i])

{

primes[cnt ++ ] = i;

euler[i] = i - 1;

}

for (int j = 0; primes[j] <= n / i; j ++ )

{

int t = primes[j] \* i;

st[t] = true;

if (i % primes[j] == 0)

{

euler[t] = euler[i] \* primes[j];

break;

}

euler[t] = euler[i] \* (primes[j] - 1);

}

}

}

**欧几里得算法**

int gcd(int a, int b) *// 欧几里得算法*

{

return b ? gcd(b, a % b) : a;

}

**扩展欧几里得算法**

int exgcd(int a, int b, int &x, int &y) *// 扩展欧几里得算法, 求x, y，使得ax + by = gcd(a, b)*

{

if (!b)

{

x = 1; y = 0;

return a;

}

int d = exgcd(b, a % b, y, x);

y -= (a / b) \* x;

return d;

}

**判定质数**

bool is\_prime(int x) *// 判定质数*

{

if (x < 2) return false;

for (int i = 2; i <= x / i; i ++ )

if (x % i == 0)

return false;

return true;

}

**线性质数筛**

void get\_primes(int n) *// 线性筛质数*

{

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;

}

}

}

**快速幂算法**

int quick\_power(int a, int k, int p) *// 求a^k mod p*

{

int res = 1 % p;

while (k)

{

if (k & 1) res = (LL)res \* a % p;

a = (LL)a \* a % p;

k >>= 1;

}

return res;

}