DS期末机考模板20230218

0. 代码模板

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
#define CaseT int CaseT; cin >> CaseT; while(CaseT--)
#define endl '\n'
#define all(x) (x).begin(), (x).end()
#define rall(x) (x).rbegin(), (x).rend()
const int INF = 0x3f3f3f3f;

void solve() {

}

int main() {
    cin.tie(0)->sync_with_stdio(false);
#ifndef ONLINE_JUDGE
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
#endif
    // init();
    // CaseT
    solve();
    return 0;
}
```

1. 二叉搜索树

```cpp
const int MAXN = 1e3 + 5;
struct BinarySearchTree {
  struct Node {
    int l, r;
    int val;
    int cnt;
  }BST[MAXN];
  int root;
  int idx;

  BinarySearchTree() {
    build();
  }

  int newNode(int x) {
    BST[++idx].val = x;
    BST[idx].cnt = 1;
    BST[idx].l = BST[idx].r = 0;
    return idx;
```

```
  }

  void build() {
    root = 1, idx = 0;
    newNode(-INF), newNode(INF);
    BST[root].r = 2;
  }

  void inorderTraversal(int u, vector<int>& res) {
    if (BST[u].l) inorderTraversal(BST[u].l, res);
    if (abs(BST[u].val) != INF)
      for (int i = 0; i < BST[u].cnt; i++) res.push_back(BST[u].val);
    if (BST[u].r) inorderTraversal(BST[u].r, res);
  }

  void print() {
    vector<int> res;
    inorderTraversal(root, res);
    for (auto u : res) cout << u << ' ';
    cout << endl;
  }

  void insert(int& u, int x) {
    if (!u) u = newNode(x);
    else if (BST[u].val == x) BST[u].cnt++;
    else if (BST[u].val > x) insert(BST[u].l, x);
    else insert(BST[u].r, x);
  }

  int getPrev(int u, int x) {
    if (!u) return -INF;
    else if (BST[u].val < x) return u;
    else if (BST[u].val >= x) return getPrev(BST[u].l, x);
    else return getPrev(BST[u].r, x);
  }

  void remove(int& u, int x) {
    if (!u) return;

    if (BST[u].val == x) {
      if (BST[u].cnt > 1) BST[u].cnt--;
      else if (!BST[u].l && !BST[u].r) u = 0;
      else if (BST[u].l && !BST[u].r) u = BST[u].l;
      else if (!BST[u].l && BST[u].r) u = BST[u].r;
      else {
        int pre = getPrev(u, x);
        int r = BST[u].r;
        BST[u = pre].r = r;
      }
    }
    else if (BST[u].val > x) remove(BST[u].l, x);
    else remove(BST[u].r, x);
  }

  int find(int u, int x, int step) {
    if (!u) return INF;
    else if (BST[u].val == x) return step;
    else if (BST[u].val > x) return find(BST[u].l, x, step + 1);
```

```
    else return find(BST[u].r, x, step + 1);
  }
};
```

2. 根据树的遍历建二叉树

2.1 根据先序遍历建二叉树

给定二叉树的先序遍历,求其高度.

```
const int MAXN = 1e3 + 5;
struct BinaryTree {
  int n;  // s的长度
  string s;  // 先序遍历
  int begin;

  struct Node {
    char data;
    int l, r;
  }Tree[MAXN];
  int idx;

  BinaryTree(string _s) :s(_s) {
    n = s.length();
    begin = idx = 0;
    build();
  }

  int build() {
    if (begin == n) return 0;

    if (s[begin] == '0') {
      begin++;
      return 0;
    }

    int cur = ++idx;
    Tree[cur].data = s[begin++];
    Tree[cur].l = build(), Tree[cur].r = build();
    return cur;
  }

  void print() {
    for (int i = 1; i <= n; i++) {
      cout << i << ' ' << Tree[i].data << ' '
        << Tree[i].l << ' ' << Tree[i].r << endl;
    }
  }

  int getHeight(int u) {
    if (!u) return 0;
    else return max(getHeight(Tree[u].l), getHeight(Tree[u].r)) + 1;
  }
};

```

```
void solve() {
  string s; cin >> s;

  BinaryTree tr(s);
  cout << tr.getHeight(1) << endl;
}
```

2.2 先序+中序还原二叉树

```
const int MAXN = 1e3 + 5;
struct BinaryTree {
  struct Node {
    char data;
    int l, r;
  }Tree[MAXN];
  int idx;

  BinaryTree(int _n, string _pre, string _in) {
    idx = 0;
    build(_pre, _in, _n);
  }

  int build(string pre, string in, int n) {  // 要建立的节点数
    if (!n) return 0;

    int cur = ++idx;
    Tree[cur].data = pre[0];

    int pos = in.find(pre[0]);
    Tree[cur].l = build(pre.substr(1), in, pos);
    Tree[cur].r = build(pre.substr(pos + 1), in.substr(pos + 1), n - pos - 1);
    return cur;
  }

  int getHeight(int u) {
    if (!u) return 0;
    else return max(getHeight(Tree[u].l), getHeight(Tree[u].r)) + 1;
  }
};

void solve() {
  int n; string pre, in; cin >> n >> pre >> in;

  BinaryTree tr(n, pre, in);
  cout << tr.getHeight(1) << endl;
}
```

2.3 后序+中序还原二叉树

```
const int MAXN = 1e3 + 5;
struct BinaryTree {
  struct Node {
    char data;
    int l, r;
```

```
  }Tree[MAXN];
  int idx;

  BinaryTree(int _n, string _post, string _in) {
    idx = 0;
    build(_post, _in, _n);
  }

  int build(string post, string in, int n) {  // 要建立的节点数
    if (!n) return 0;

    int cur = ++idx;
    Tree[cur].data = post.back();

    int pos = in.find(post.back());
    Tree[cur].r = build(post.substr(pos, n - pos - 1),
                        in.substr(pos + 1), n - pos - 1);
    Tree[cur].l = build(post.substr(0, pos), in, pos);
    return cur;
  }

  void inorderTraversal(int u, vector<char>& res) {
    res.push_back(Tree[u].data);
    if (Tree[u].l) inorderTraversal(Tree[u].l, res);
    if (Tree[u].r) inorderTraversal(Tree[u].r, res);
  }
};

void solve() {
  int n; cin >> n;
  string post, in;
  for (int i = 0; i < n; i++) {
    int ch; cin >> ch;
    post.push_back(ch);
  }
  for (int i = 0; i < n; i++) {
    int ch; cin >> ch;
    in.push_back(ch);
  }

  BinaryTree tr(n, post, in);
  vector<char> res;
  tr.inorderTraversal(1, res);
  cout << "Preorder: ";
  for (auto u : res) cout << (int)u << " \n"[u == res.back()];
}
```

3. 二叉树的遍历

3.1 递归实现

```cpp
const int MAXN = 1e3 + 5;
struct BinaryTree {
  int n;  // s的长度
  string s;  // 先序遍历
  int begin;

  struct Node {
    char data;
    int l, r;
  }Tree[MAXN];
  int idx;

  BinaryTree(string _s) :s(_s) {
    n = s.length();
    begin = idx = 0;
    build();
  }

  int build() {
    if (begin == n) return 0;

    if (s[begin] == '#') {
      begin++;
      return 0;
    }

    int cur = ++idx;
    Tree[cur].data = s[begin++];
    Tree[cur].l = build(), Tree[cur].r = build();
    return cur;
  }

  void print() {
    for (int i = 1; i <= n; i++) {
      cout << i << ' ' << Tree[i].data << ' '
        << Tree[i].l << ' ' << Tree[i].r << endl;
    }
  }

  void bfs(vector<char>& res) {
    queue<int> que;
    que.push(1);

    while (que.size()) {
      auto u = que.front(); que.pop();
      res.push_back(Tree[u].data);

      if (Tree[u].l) que.push(Tree[u].l);
      if (Tree[u].r) que.push(Tree[u].r);
    }
  }

  void preorderTraversal(int u, vector<char>& res) {
    res.push_back(Tree[u].data);
    if (Tree[u].l) preorderTraversal(Tree[u].l, res);
```

```
    if (Tree[u].r) preorderTraversal(Tree[u].r, res);
  }

  void inorderTraversal(int u, vector<char>& res) {
    if (Tree[u].l) inorderTraversal(Tree[u].l, res);
    res.push_back(Tree[u].data);
    if (Tree[u].r) inorderTraversal(Tree[u].r, res);
  }

  void postorderTraversal(int u, vector<char>& res) {
    if (Tree[u].l) postorderTraversal(Tree[u].l, res);
    if (Tree[u].r) postorderTraversal(Tree[u].r, res);
    res.push_back(Tree[u].data);
  }
};
```

3.2 非递归实现

```
const int MAXN = 1e3 + 5;
struct BinaryTree {
  int n;  // s的长度
  string s;  // 先序遍历
  int begin;

  struct Node {
    char data;
    int l, r;
  }Tree[MAXN];
  int idx;

  BinaryTree(string _s) :s(_s) {
    n = s.length();
    begin = idx = 0;
    build();
  }

  int build() {
    if (begin == n) return 0;

    if (s[begin] == '#') {
      begin++;
      return 0;
    }

    int cur = ++idx;
    Tree[cur].data = s[begin++];
    Tree[cur].l = build(), Tree[cur].r = build();
    return cur;
  }

  void print() {
    for (int i = 1; i <= n; i++) {
      cout << i << ' ' << Tree[i].data << ' '
        << Tree[i].l << ' ' << Tree[i].r << endl;
    }
  }
```

```
39
40    void preorderTraversal(int u, vector<char>& res) {
41      stack<int> stk;
42      stk.push(u);
43
44      while (stk.size()) {
45        auto v = stk.top(); stk.pop();
46        res.push_back(Tree[v].data);
47
48        if (Tree[v].r) stk.push(Tree[v].r);
49        if (Tree[v].l) stk.push(Tree[v].l);
50      }
51    }
52
53    void inorderTraversal(int u, vector<char>& res) {
54      stack<int> stk;
55      while (u || stk.size()) {
56        while (u) {
57          stk.push(u);
58          u = Tree[u].l;
59        }
60
61        u = stk.top(); stk.pop();
62        res.push_back(Tree[u].data);
63        u = Tree[u].r;
64      }
65    }
66
67    void postorderTraversal(int u, vector<char>& res) {
68      stack<int> stk;
69      int last = -1;
70
71      while (u || stk.size()) {
72        while (u) {
73          stk.push(u);
74          u = Tree[u].l;
75        }
76
77        u = stk.top();
78        if (!Tree[u].r || last == Tree[u].r) {
79          res.push_back(Tree[u].data);
80          stk.pop();
81          last = u;
82          u = 0;
83        }
84        else u = Tree[u].r;
85      }
86    }
87  };
88
89  void solve() {
90    string s; cin >> s;
91    BinaryTree tr(s);
92
93    cout << "Inorder: ";
94    vector<char> res;
95    tr.inorderTraversal(1, res);
96    for (auto u : res) cout << u << " \n"[u == res.back()];
```

```
97
98    cout << "Preorder: ";
99    res.clear();
100   tr.preorderTraversal(1, res);
101   for (auto u : res) cout << u << " \n"[u == res.back()];
102
103   cout << "Postorder: ";
104   res.clear();
105   tr.postorderTraversal(1, res);
106   for (auto u : res) cout << u << " \n"[u == res.back()];
107 }
```

4. KMP

4.1 KMP模板

```
1  struct KMP {
2    int n, m;  // 模式串、文本串长度
3    string p, s;  // 模式串、文本串
4    vector<int> nxt;
5
6    KMP(string _p, string _s = "") :
7    p(" " + _p), n(_p.length()), s(" " + _s), m(_s.length()) {
8      nxt.resize(n + 1);
9
10     for (int i = 2, j = 0; i <= n; i++) {
11       while (j && p[i] != p[j + 1]) j = nxt[j];
12       if (p[i] == p[j + 1]) j++;
13       nxt[i] = j;
14     }
15   }
16
17   vector<int> kmp() {
18     vector<int> res;
19     for (int i = 1, j = 0; i <= m; i++) {
20       while (j && s[i] != p[j + 1]) j = nxt[j];
21       if (s[i] == p[j + 1]) j++;
22
23       if (j == n) {
24         res.push_back(i - n);
25         j = nxt[j];
26       }
27     }
28     return res;
29   }
30 };
31
32 void solve() {
33   string s, p; cin >> s >> p;
34
35   KMP solver(p, s);
36
37   int len = (int)p.length();
38   cout << -1 << ' ';
39   for (int i = 1; i < len; i++) cout << solver.nxt[i] << ' ';
```

```
  cout << endl;

  auto ans = solver.kmp();
  cout << (ans.size() ? ans[0] + 1 : 0) << endl;
}
```

4.2 求字符串的最长不重叠公共前后缀

```
void solve() {
  string p; cin >> p;

  KMP solver(p);
  int len = p.length();
  if (!solver.nxt[len]) cout << "empty" << endl;
  else {
    while (solver.nxt[len] * 2 > len) len = solver.nxt[len];
    cout << p.substr(0, solver.nxt[len]) << endl;
  }
}
```

4.3 求字符串周期

给定一个字符串,求至少需在末尾添加多少个字符可使得该串由某个不为自身的字串循环而成.

```
void solve() {
  string p; cin >> p;

  KMP solver(p);
  int len = p.length(), t = len - solver.nxt[len];
  cout << (t == len ? len : (len + t - 1) / t * t - len) << endl;
}
```

5. Huffman树与Huffman编码

```
const int MAXN = 1e3 + 5;
struct HuffmanTree {
  int n;  // 原序列长度

  struct Node {
    int id;
    char data;
    int w;
    int l, r;
    int fa;

    Node() : l(0), r(0), fa(0) {}
    Node(int _id, int _w, int _l, int _r) :id(_id), w(_w), l(_l), r(_r) {}

    bool operator>(const Node& p)const {
      return w != p.w ? w > p.w : id > p.id;
    }
```

```
}Tree[MAXN];
int idx;  // 当前用到的节点编号

map<char, string> mp;  // 编码结果

// 新建一个权值为w的节点,左右儿子节点分别为l、r
Node newNode(int w, int l = 0, int r = 0) {
  return Tree[idx] = { ++idx, w, l, r };
}

HuffmanTree(int _n, const vector<int>& _a, const vector<char>& _chs):
n(_n) {  // a[]下标从1开始
  idx = 0;

  priority_queue<Node, vector<Node>, greater<Node>> heap;
  for (int i = 1; i <= n; i++) {
    heap.push(newNode(_a[i]));
    Tree[i].data = _chs[i];
  }

  while (heap.size() >= 2) {
    auto left = heap.top(); heap.pop();
    auto right = heap.top(); heap.pop();
    auto root = newNode(left.w + right.w, left.id, right.id);
    heap.push(root);
    Tree[left.id].fa = Tree[right.id].fa = root.id;
  }
}

void print() {
  for (int i = 1; i <= idx; i++) {
    cout << Tree[i].id << ' ' << Tree[i].w << ' '
      << Tree[i].l << ' ' << Tree[i].r << endl;
  }
}

void encode() {
  for (int i = 1; i <= n; i++) {  // 枚举叶子节点
    string res;
    for (int u = i; u != idx; u = Tree[u].fa) {
      if (Tree[Tree[u].fa].l == u) res.push_back('0');
      else res.push_back('1');
    }

    reverse(all(res));
    cout << Tree[i].data << " :" << res << endl;
    mp[Tree[i].data] = res;
  }
}

string encode(string s) {
  string res;
  for (auto ch : s) res += mp[ch];
  return res;
}

void decode(string s) {
  int u = idx;
```

```
    string res;
    for (auto ch : s) {
      if (ch == '0') {
        if (Tree[u].l) u = Tree[u].l;
        else {
          cout << "error!" << endl;
          return;
        }
      }
      else {
        if (Tree[u].r) u = Tree[u].r;
        else {
          cout << "error!" << endl;
          return;
        }
      }

      if (1 <= Tree[u].id && Tree[u].id <= n) {  // 叶子节点
        res.push_back(Tree[u].data);
        u = idx;  // 回到根节点
      }
    }
    cout << (u == idx ? res : "error!") << endl;
  }
};

void solve() {
  int n; cin >> n;
  vector<char> chs(n + 1);
  vector<int> a(n + 1);
  for (int i = 1; i <= n; i++) cin >> chs[i];
  for (int i = 1; i <= n; i++) cin >> a[i];

  HuffmanTree tr(n, a, chs);
  tr.encode();

  string s; cin >> s;
  cout << tr.encode(s) << endl;

  cin >> s;
  tr.decode(s);
}
```

6. 最短路

6.1 Floyd算法求最短路

```
const int MAXN = 1e3 + 5;
namespace Floyd {
  int n;  // 节点数
  ll dis[MAXN][MAXN];  // dis[u][v]表示节点u与v间的最短距离

  void init() {  // 初始化dis[][]
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++)
```

```
        dis[i][j] = i == j ? 0 : INF;
    }
  }

  void floyd() {
    for (int k = 1; k <= n; k++) {
      for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++)
          dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
      }
    }
  }
}
using namespace Floyd;
```

6.2 Floyd算法求传递闭包

```
const int MAXN = 1e3 + 5;
namespace Floyd {
  int n;  // 节点数
  bool d[MAXN][MAXN];  // 图的可达矩阵:d[u][v]表示节点u与v间的是否有边

  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] |= d[i][k] & d[k][j];
      }
    }
  }
}
using namespace Floyd;

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) cin >> d[i][j];

  floyd();
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++)
      cout << d[i][j] << " \n"[j == n];
  }
}
```

6.3 Dijkstra算法

```
struct Dijkstra {
  int n;
  vector<vector<pair<int, int>>> edges;
  vector<bool> state;
  vector<ll> dis;

  Dijkstra(int _n, const vector<vector<pair<int, int>>>& _edges) :n(_n), edges(_edges) {
```

```
    state.resize(n + 1);
    dis = vector<ll>(n + 1, INF);
  }

  void dijkstra(int s) {
    dis[s] = 0;
    for (int i = 0; i < n - 1; i++) {
      int t = -1, j;
      for (j = 1; j <= n; j++) {
        if (!state[j] && (t == -1 || dis[t] > dis[j]))
          t = j;
      }
      if (j == n) break;

      state[t] = true;
      for (auto [v, w] : edges[t])
        dis[v] = min(dis[v], dis[t] + w);
    }
  }
};
```

6.4 SPFA算法

给定一个有重边的无向图,求节点1到节点n的最短路.

```
const int MAXN = 1e3 + 5;
struct SPFA {
  int n;  // 节点数
  int d[MAXN][MAXN];
  ll dis[MAXN];  // 起点到每个节点的最短距离
  bool state[MAXN];  // 记录当前每个节点是否在队列中

  SPFA(int _n) :n(_n) {
    memset(d, INF, sizeof(d));
    memset(state, false, sizeof(state));
  }

  void addEdge(int u, int v, int w) {
    d[u][v] = d[v][u] = min(d[u][v], w);  // 重边保留最短的
  }

  ll spfa(int s) {  // 求起点s到其他节点的最短路
    memset(dis, INF, sizeof(dis));

    queue<int> que;
    dis[s] = 0;
    que.push(s);
    state[s] = true;

    while (que.size()) {
      auto u = que.front(); que.pop();
      state[u] = false;

      for (int v = 1; v <= n; v++) {
        if (d[u][v] != INF) {
          if (dis[v] > dis[u] + d[u][v]) {
```

```
            dis[v] = dis[u] + d[u][v];

            if (!state[v]) {
              que.push(v);
              state[v] = true;
            }
          }
        }
      }
    }
    return dis[n];
  }
};

void solve() {
  int n;
  while (cin >> n) {
    SPFA solver(n);

    int m; cin >> m;
    while (m--) {
      int u, v, w; cin >> u >> v >> w;
      solver.addEdge(u, v, w);
    }

    cout << solver.spfa(1) << endl;
  }
}
```

7. 最小生成树

7.1 Prim算法

```
const int MAXN = 1e3 + 5;
struct Prim {
  int n;  // 节点数
  int edges[MAXN][MAXN];
  ll dis[MAXN];  // dis[u]表示节点u到连通块的最短距离
  bool state[MAXN];  // 记录当前每个节点是否在连通块中

  Prim(int _n) :n(_n) {
    memset(state, false, sizeof(state));
  }

  ll prim() {  // 求MST,返回MST的边权和
    memset(dis, INF, sizeof(dis));

    ll res = 0;
    for (int i = 0; i < n; i++) {  // 做n次迭代
      int t = -1;  // 不在连通块中的与连通块距离最近的节点
      for (int j = 1; j <= n; j++) {
        if (!state[j] && (t == -1 || dis[t] > dis[j]))
          t = j;
      }

```

```
      if (i && dis[t] == INF) return INFF;  // 图不连通,不存在MST

      if (i) res += dis[t];  // 第一个节点无边,注意先更新res再更新dis[]
      state[t] = true;  // 将节点t加入连通块
      for (int j = 1; j <= n; j++)
        dis[j] = min(dis[j], (ll)edges[t][j]);  // 注意不是dis[j]+edges[t][j]
    }
    return res;
  }
};
```

7.2 Kruskal算法

```
struct Kruskal {
  int n, m;  // 节点数、边数
  struct Edge {
    int u, v, w;

    bool operator<(const Edge& p) const {
      return w < p.w;
    }
  };
  vector<Edge> edges;
  vector<int> fa;

  Kruskal(int  _n) :n(_n), m(0) {
    fa.resize(n + 1);
    iota(all(fa), 0);
  }

  int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
  }

  ll kruskal() {  // 求MST,返回MST的边权和
    sort(all(edges));

    ll res = 0;  // MST的边权和
    int cnt = 0;  // 当前连的边数
    for (int i = 0; i < m; i++) {
      auto [u, v, w] = edges[i];

      u = find(u), v = find(v);
      if (u != v) {
        fa[u] = v;
        res += w;
        cnt++;
      }
    }
    return cnt < n - 1 ? INFF : res;
  }
};
```

8. 拓扑排序

给定一个DAG,用邻接矩阵存图,求拓扑序的第一个节点到其他节点的最长路,输出路径.

```
const double eps = 1e-8;

int sgn(double x) {
  if (fabs(x) < eps) return 0;
  else return x < 0 ? -1 : 1;
}

const int MAXN = 1e3 + 5;
namespace TopologicalSorting {
  int n;
  double edges[MAXN][MAXN];
  int in[MAXN];
  vector<int> res;  // 拓扑序
  double dis[MAXN];
  int pre[MAXN];

  void addEdge(int u, int v, double w) {
    edges[u][v] = w;
    in[v]++;
  }

  void topo() {
    queue<int> que;
    for (int i = 0; i < n; i++)
      if (!in[i]) que.push(i);

    while (que.size()) {
      auto u = que.front(); que.pop();
      res.push_back(u);

      for (int v = 0; v < n; v++) {
        if (sgn(edges[u][v]))
          if (--in[v] == 0) que.push(v);
      }
    }
  }

  void cal() {
    for (auto u : res) {
      for (int v = 0; v < n; v++) {
        if (sgn(edges[u][v])) {
          double tmp = dis[u] + edges[u][v];
          if (dis[v] < tmp) {
            dis[v] = tmp;
            pre[v] = u;
          }
        }
      }
    }
  }
}
using namespace TopologicalSorting;

```

```cpp
void solve() {
  int m; cin >> n >> m;
  while (m--) {
    int u, v; double w; cin >> u >> v >> w;
    addEdge(u, v, w);
  }

  topo();
  cal();

  for (int i = 0; i < n; i++) {
    if (i == 5) continue;

    int cur = i;
    vector<int> path;
    while (true) {
      path.push_back(cur);
      cur = pre[cur];

      if (cur == 5) {
        path.push_back(cur);
        break;
      }
    }

    reverse(all(path));
    for (auto u : path) cout << u << " \n"[u == path.back()];
  }
}
```

9. 查找

9.1 顺序查找

实现一个顺序表,支持插入元素、删除位置、删除元素、查找元素,其中查找元素需输出比较次数.

```cpp
const int MAXN = 1e5 + 5;
int n;
int a[MAXN];

void print() {
  cout << n << ' ';
  for (int i = 1; i <= n; i++) cout << a[i] << ' ';
  cout << endl;
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];
  print();

  int pos, val; cin >> pos >> val;
  if (pos < 1 || pos > n + 1) cout << "ERROR" << endl;
  else {
    for (int i = n; i >= pos; i--) a[i + 1] = a[i];
```

```
    n++;
    a[pos] = val;
    print();
  }

  cin >> pos;
  if (pos < 1 || pos > n) cout << "ERROR" << endl;
  else {
    for (int i = pos; i <= n; i++) a[i] = a[i + 1];
    n--;
    print();
  }

  cin >> val;
  for (pos = 1; pos <= n && a[pos] != val; pos++);
  if (pos == n + 1) cout << "ERROR" << endl;
  else {
    for (int i = pos; i <= n; i++) a[i] = a[i + 1];
    n--;
    print();
  }

  cin >> val;
  bool ok = false;
  for (int i = n; i >= 1; i--) {  // 注意从后往前查找
    if (a[i] == val) {
      cout << 1 << ' ' << i << ' ' << n - i + 1 << endl;
      ok = true;
      break;
    }
  }
  if (!ok) cout << 0 << ' ' << 0 << ' ' << n + 1 << endl;
}
```

9.2 折半查找

给定一个下标从1开始的有序序列,用折半查找找到元素的位置.

```
void solve() {
  int n; cin >> n;
  vector<int> a(n + 5);
  for (int i = 1; i <= n; i++) cin >> a[i];

  CaseT{
    int x; cin >> x;

    int l = 1, r = n;
    while (l <= r) {
      int mid = l + r >> 1;
      if (a[mid] > x) r = mid - 1;
      else l = mid + 1;
    }
    cout << (a[r] == x ? to_string(r) : "error") << endl;
  }
}
```

9.3 顺序索引查找

索引表和块内查找都采用不带哨兵、从头开始的顺序查找.

```
void solve() {
  int n; cin >> n;  // 序列长度
  vector<int> a(n + 1);
  for (int i = 1; i <= n; i++) cin >> a[i];
  int m; cin >> m;  // 分块数
  vector<int> rangeMax(m + 1);
  for (int i = 1; i <= m; i++) cin >> rangeMax[i];

  vector<int> start(m + 2);  // 每个分块的起始下标
  start[1] = 1, start[m + 1] = n;  // 边界
  for (int i = 1, j = 1; i <= n; i++)
    if (a[i] > rangeMax[j]) start[++j] = i;

  CaseT{
    int x; cin >> x;
    if (x > rangeMax[m]) cout << "error" << endl;
    else {
      int step = 0;  // 比较次数
      int pos;
      for (pos = 1; pos <= m; pos++) {
        step++;
        if (rangeMax[pos] >= x) break;
      }

      bool ok = false;
      for (int i = start[pos]; i <= start[pos + 1]; i++) {
        step++;
        if (a[i] == x) {
          cout << i << '-' << step << endl;
          ok = true;
          break;
        }
      }
      if (!ok) cout << "error" << endl;
    }
  }
}
```

9.4 哈希查找(线性探测法)

```
const int MAXN = 1e3 + 5;
struct HashTable {
  const int MOD = 11;
  int len;  // 表长
  int ha[MAXN];

  HashTable(int _len) :len(_len) {
    for (int i = 0; i < len; i++) ha[i] = -1;
  }

  void insert(int x) {
```

```
    int pos = x % MOD;
    while (pos < len && ~ha[pos]) pos = (pos + 1) % len;
    ha[pos] = x;
  }

  pair<int, int> find(int x) {  // 返回查找次数、查找成功的位置
    int pos = x % MOD;
    int step = 1;  // 查找次数
    for (int i = pos; i < len; i++, step++) {
      if (ha[i] == -1) return { step, -1 };
      else if (ha[i] == x) return { step, i + 1 };
    }

    for (int i = 0; i < pos; i++, step++) {
      if (ha[i] == -1) return { step, -1 };
      else if (ha[i] == x) return { step, i + 1 };
    }
    return { step, -1 };
  }

  void print() {
    for (int i = 0; i < len; i++)
      cout << (~ha[i] ? to_string(ha[i]) : "NULL") << " \n"[i == len - 1];
  }
};

void solve() {
  int len, n; cin >> len >> n;
  HashTable hat(len);
  while (n--) {
    int x; cin >> x;
    hat.insert(x);
  }
  hat.print();

  CaseT{
    int x; cin >> x;

    auto [u, v] = hat.find(x);
    if (~v) cout << 1 << ' ' << u << ' ' << v << endl;
    else cout << 0 << ' ' << u << endl;
  }
}
```

9.5 哈希查找(二次探测法)

```
const int MAXN = 1e3 + 5;
struct HashTable {
  const int MOD = 11;
  int len;  // 表长
  int ha[MAXN];

  HashTable(int _len) :len(_len) {
    for (int i = 0; i < len; i++) ha[i] = -1;
  }

```

```cpp
  void insert(int x) {
    int pos = x % MOD;
    for (int cur = 0; ; cur++) {
      for (int i = 0; i < 2; i++) {
        int tmp = ((pos + (i ? -1 : 1) * cur * cur) % len + len) % len;
        if (ha[tmp] == -1) {
          ha[tmp] = x;
          return;
        }
      }
    }
  }

  pair<int, int> find(int x) {  // 返回查找次数、查找成功的位置
    int pos = x % MOD;
    int step = 1;  // 查找次数
    for (int cur = 0; ; cur++) {
      for (int i = 0; i < 2; i++) {
        int tmp = ((pos + (i ? -1 : 1) * cur * cur) % len + len) % len;
        if (cur) step++;  // cur=0时只需一次

        if (ha[tmp] == -1) return { step, -1 };
        else if (ha[tmp] == x) return { step, tmp + 1 };
      }

      if (cur > sqrt(len)) return { step, -1 };
    }
  }

  void print() {
    for (int i = 0; i < len; i++)
      cout << (~ha[i] ? to_string(ha[i]) : "NULL") << " \n"[i == len - 1];
  }
};

void solve() {
  int len, n; cin >> len >> n;
  HashTable hat(len);
  while (n--) {
    int x; cin >> x;
    hat.insert(x);
  }
  hat.print();

  CaseT{
    int x; cin >> x;

    auto [u, v] = hat.find(x);
    if (~v) cout << 1 << ' ' << u << ' ' << v << endl;
    else cout << 0 << ' ' << u << endl;
  }
}
```

9.6 哈希查找(链地址法)

```cpp
const int MAXN = 1e3 + 5;
struct HashTable {
  const int MOD = 11;
  list<int> ha[MAXN];

  void insert(int x) {
    int pos = x % MOD;
    ha[pos].insert(ha[pos].begin(), x);
  }

  pair<int, int> find(int x) {  // 返回查找次数、查找成功的位置
    int pos = x % MOD;
    int step = 1;
    for (auto it = ha[pos].begin(); it != ha[pos].end(); it++, step++)
      if (*it == x) return { step, pos };
    return { -1, -1 };
  }
};

void solve() {
  HashTable hat;
  int n; cin >> n;
  while (n--) {
    int x; cin >> x;
    hat.insert(x);
  }

  CaseT{
    int x; cin >> x;

    auto [u, v] = hat.find(x);
    if (~v) cout << v << ' ' << u << endl;
    else {
      cout << "error" << endl;
      hat.insert(x);
    }
  }
}
```

---

10. 排序

10.1 直插排序

升序排列.

```cpp
const int MAXN = 1e3 + 5;
int n;
int a[MAXN];

void print() {
  for (int i = 0; i < n; i++)
    cout << a[i] << " \n"[i == n - 1];
```

```
}

void insertSort() {
  for (int i = 1; i < n; i++) {
    for (int j = i - 1; j >= 0 && a[j] > a[j + 1]; j--)
      swap(a[j], a[j + 1]);
    print();
  }
}

void solve() {
  cin >> n;
  for (int i = 0; i < n; i++) cin >> a[i];

  insertSort();
}
```

10.2 希尔排序

降序排列.

```
const int MAXN = 1e3 + 5;
int n;
int a[MAXN];

void print() {
  for (int i = 0; i < n; i++)
    cout << a[i] << " \n"[i == n - 1];
}

void shellSort() {
  int gap = n;
  while (gap > 1) {
    gap /= 2;
    for (int i = gap; i < n; i++) {
      int tmp;
      if (a[i - gap] < a[i]) {
        tmp = a[i];
        int j = i - gap;
        while (j >= 0 && a[j] < tmp) {
          a[j + gap] = a[j];
          j -= gap;
        }
        a[j + gap] = tmp;
      }
    }
    print();
  }
}

void solve() {
  cin >> n;
  for (int i = 0; i < n; i++) cin >> a[i];

  shellSort();
  cout << endl;
```

```
}
```

10.3 冒泡排序

升序排列.

```
const int MAXN = 1e3 + 5;
int n;
int a[MAXN];
int ans;  // 交换次数

void print() {
  for (int i = 0; i < n; i++)
    cout << a[i] << " \n"[i == n - 1];
}

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

void solve() {
  while (cin >> n) {
    for (int i = 0; i < n; i++) cin >> a[i];

    ans = 0;
    bubbleSort();
    cout << ans << endl;
  }
}
```

10.4 快速排序

升序排列.

```
const int MAXN = 1e3 + 5;
int n;
int a[MAXN];

void print() {
  for (int i = 1; i <= n; i++)
    cout << a[i] << " \n"[i == n];
}

void quickSort(int l, int r) {
  if (l >= r) return;

  int pivot = a[l], st = l, ed = r;
```

```
  while (l < r) {
    for (int i = r; i > l; i--, r--) {
      if (a[i] < pivot) {
        a[l++] = a[i];
        break;
      }
    }

    for (int i = l; i < r; i++, l++) {
      if (a[i] > pivot) {
        a[r--] = a[i];
        break;
      }
    }
  }

  a[l] = pivot;
  print();

  quickSort(st, l - 1), quickSort(l + 1, ed);
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  quickSort(1, n);
  cout << endl;
}
```

输出每趟排好序的元素及其位置.

```
const int MAXN = 1e3 + 5;
int n;
int a[MAXN];

int partition(int l, int r) {
  int pivot = a[l];
  while (l < r) {
    while (l < r && a[r] >= pivot) r--;
    a[l] = a[r];
    while (l < r && a[l] < pivot) l++;
    a[r] = a[l];
  }
  a[l] = pivot;

  cout << pivot << ' ' << l << endl;
  return l;
}

void quickSort(int l, int r) {
  if (l <= r) {
    int pos = partition(l, r);
    quickSort(l, pos - 1), quickSort(pos + 1, r);
  }
```

```
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  quickSort(1, n);
  cout << endl;
}
```

10.5 堆排序

降序排列,建小根堆.

```
const int MAXN = 1e3 + 5;
struct HeapSort {
  int n;
  int heap[MAXN];

  HeapSort(int _n) :n(_n) {
    for (int i = 0; i < n; i++) cin >> heap[i];

    init();
    heapSort();
  }

  void sift(int u, int length) {
    int ls = u * 2 + 1, rs = u * 2 + 2;
    if (ls >= length) return;  // 不存在左儿子

    if (rs < length) {  // 存在右儿子
      if (heap[ls] < heap[rs]) {
        if (heap[ls] < heap[u]) {
          swap(heap[ls], heap[u]);
          sift(ls, length);
        }
      }
      else {
        if (heap[rs] < heap[u]) {
          swap(heap[rs], heap[u]);
          sift(rs, length);
        }
      }
    }
    else {  // 只有左儿子
      if (heap[ls] < heap[u]) {
        swap(heap[ls], heap[u]);
        sift(ls, length);
      }
    }
  }

  void print() {
    cout << n << ' ';
    for (int i = 0; i < n; i++)
      cout << heap[i] << " \n"[i == n - 1];
```

```
  }

  void init() {  // 调整为初始堆
    for (int i = n / 2; i >= 0; i--) sift(i, n);
    print();
  }

  void heapSort() {
    for (int i = n - 1; i >= 1; i--) {
      swap(heap[0], heap[i]);
      sift(0, i);
      print();
    }
  }
};
```

10.6 选择排序

升序排列.

```
const int MAXN = 1e3 + 5;
int n;
int a[MAXN];

void print() {
  for (int i = 1; i <= n; i++)
    cout << a[i] << " \n"[i == n];
}

void selectSort() {
  for (int i = 1; i <= n; i++) {
    int minidx = i;
    for (int j = i + 1; j <= n; j++)
      if (a[j] < a[minidx]) minidx = j;
    swap(a[i], a[minidx]);
    print();
  }
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  selectSort();
  cout << endl;
}
```

10.7 归并排序

```
const int MAXN = 1e3 + 5;
int n;
string a[MAXN], tmp[MAXN];

void print() {
```

```
  for (int i = 0; i < n; i++)
    cout << a[i] << " \n"[i == n - 1];
}

void mergeSort() {
  int lim;  // >=n的最小的2的幂次
  for (lim = 1; lim < n; lim *= 2);

  for (int len = 2; len <= lim; len *= 2) {  // len为当前归并的长度
    for (int i = 0; i < n; i += len) {
      int l = i, r = i + len / 2;
      if (r > n) break;

      int mid = r - 1;
      int idx = l;
      while (l <= mid && r < min(i + len, n)) {
        if (a[l] > a[r]) tmp[idx++] = a[l++];
        else tmp[idx++] = a[r++];
      }

      while (l <= mid) tmp[idx++] = a[l++];
      while (r < min(i + len, n)) tmp[idx++] = a[r++];

      for (int j = i; j < min(i + len, n); j++)
        a[j] = tmp[j];
    }
    print();
  }
}

void solve() {
  cin >> n;
  for (int i = 0; i < n; i++) cin >> a[i];

  mergeSort();
  cout << endl;
}
```

10.8 基数排序

```
const int MAXN = 1e3 + 5;
int n;
int a[MAXN];

void print(vector<int> a[10]) {
  for (int i = 0; i < 10; i++) {
    cout << i << ':';
    if (!a[i].size()) {
      cout << "NULL" << endl;
      continue;
    }

    for (int j = 0; j < a[i].size(); j++) {
      cout << "->";
      cout << a[i][j];
      if (j == a[i].size() - 1) cout << "->^" << endl;
```

```
    }
  }
}

void print() {
  for (int i = 1; i <= n; i++) cout << a[i] << " \n"[i == n];
}

void radixSort() {
  int lim = 0;
  for (int i = 1; i <= n; i++)
    while ((int)pow(10, lim) <= a[i]) lim++;

  for (int i = 0; i < lim; i++) {
    vector<int> radix[10];
    for (int j = 1; j <= n; j++)
      radix[a[j] / (int)pow(10, i) % 10].push_back(a[j]);
    print(radix);

    int idx = 1;
    for (int j = 0; j < 10; j++)
      for (auto ai : radix[j]) a[idx++] = ai;
    print();
  }
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  radixSort();
  cout << endl;
}
```

11. 顺序表

```
template<class T>
class Array {
private:
  int capacity;  // 最大长度
  int length;  // 当前长度
  T* arr;

public:
  Array() :capacity(100), length(0), arr(nullptr) {}

  ~Array() {
    if (arr) delete[] arr;
  }

  void setCapacity(int n) {  // 设置最大容量
    T* tmp = new T[capacity = n];
    memcpy(tmp, arr, sizeof(T) * length);
    delete[] arr;
    arr = tmp;
```

```
  }

  int getLength() {  // 获取当前的元素个数
    return length;
  }

  T& operator[](int idx) {  // 支持通过[]访问顺序表元素,默认下标合法
    return arr[idx];
  }

  void create(int n, int maxLength = 0) {  // 创建一个长度为n的顺序表,最大长度为maxLength
    capacity = maxLength ? maxLength : 2 * n;
    arr = new T[capacity];
    length = n;
  }

  void input() {
    for (int i = 0; i < length; i++) cin >> arr[i];
  }

  void print() {
    for (int i = 0; i < length; i++) cout << arr[i] << ' ';
  }

  bool insertToPos(T val, int pos) {  // 在下标pos位置插入元素val
    if (pos < 0 || pos > length || length == capacity) return false;

    for (int i = length - 1; i >= pos; i--) arr[i + 1] = arr[i];
    arr[pos] = val;
    length++;
    return true;
  }

  bool removeByPos(int pos) {  // 删除下标pos的元素
    if (pos < 0 || pos >= length) return false;

    for (int i = pos; i < length; i++) arr[i] = arr[i + 1];
    length--;
    return true;
  }
};

void print(Array<int>& arr) {
  cout << arr.getLength() << ' ';
  arr.print();
  cout << endl;
}
```

12. 单链表

```
#define npt nullptr

class Node {
private:
  int data;
```

```
  Node* nxt;

public:
  friend class LinkedList;

  Node(int _data = 0) :data(_data) {
    nxt = npt;
  }

  int getData() {
    return data;
  }
};

class LinkedList {
private:
  Node* head;  // 头节点,为链表的第0个节点
  int length;  // 当前链表长度(即元素个数,不含头节点)

public:
  LinkedList(int _length = 0) :length(_length) {
    head = new Node;
    if (length) create(length);
  }

  ~LinkedList() {
    Node* cur = head->nxt;
    while (cur) {
      Node* trash = cur;
      cur = cur->nxt;
      delete trash;
    }
  }

  int getLength() {  // 获取当前的元素个数
    return length;
  }

  void create(int n) {  // 新建一个长度为n的单链表,尾插法
    length = n;
    Node* cur = head;
    while (n--) {
      Node* tmp = new Node;
      cin >> tmp->data;
      cur->nxt = tmp;
      cur = tmp;
    }
  }

  void create(const vector<int>& a) {  // a[]的下标从0开始
    length = a.size();
    Node* cur = head;
    for (auto ai : a) {
      Node* tmp = new Node(ai);
      cur->nxt = tmp;
      cur = tmp;
    }
  }
```

```
64
65    void print() {  // 打印链表
66      vector<int> res;
67      for (Node* cur = head->nxt; cur; cur = cur->nxt)
68        res.push_back(cur->data);
69
70      for (int i = 0; i < length; i++)
71        cout << res[i] << ' ';
72      cout << endl;
73    }
74
75    Node* operator[](int pos) {  // 查找第pos个节点
76      if (pos < 0 || pos > length) return npt;
77
78      Node* cur = head;
79      for (int cnt = 0; cnt < pos; cnt++, cur = cur->nxt);
80      return cur;
81    }
82
83    // 在第pos个节点之后插入一个数据域为data的节点,返回是否插入成功
84    bool insertToPos(int data, int pos) {  // pos=0表示插入到链表头(头节点之后)
85      Node* cur = (*this)[pos];  // 第pos个节点
86      if (!cur) return false;
87
88      Node* tmp = new Node(data);
89      tmp->nxt = cur->nxt;
90      cur->nxt = tmp;
91      length++;
92      return true;
93    }
94
95    bool removeByPos(int pos) {  // 删除第pos个节点
96      Node* cur = (*this)[pos - 1];  // 欲删除的节点的前驱节点
97      if (!cur || !(cur->nxt)) return false;
98
99      Node* trash = cur->nxt;
100     cur->nxt = trash->nxt;
101     delete trash;
102     length--;
103     return true;
104   }
105 };
```