

## Vector

```

1 // vector 是 C++ 的動態陣列，可以根據需要自動調整大小。
2
3 #include <iostream>
4 #include <vector>
5 using namespace std;
6
7 int main() {
8     vector<int> v; // 告知一個整數型的 vector
9
10    // 插入數據
11    v.push_back(10);
12    v.push_back(20);
13    v.push_back(30);
14
15    // 使用下標訪問數據
16    for (int i = 0; i < v.size(); i++) {
17        cout << v[i] << " "; // 輸出: 10 20 30
18    }
19    cout << endl;
20
21    // 使用迭代器遍歷數據
22    for (vector<int>::iterator it = v.begin();
23         it != v.end(); ++it) {
24        cout << *it << " "; // 輸出: 10 20 30
25    }
26    cout << endl;
27
28    return 0;
29 }
```

## 使用 `vector<vector<int>>` 建立 2D 矩陣

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     // 建立一個 3x3 的 2D 矩陣
7     vector<vector<int>> matrix(3,
8         vector<int>(3, 0)); // 初始值全為 0
9
10    // 填充矩陣
11    for (int i = 0; i < 3; i++) {
12        for (int j = 0; j < 3; j++) {
13            matrix[i][j] = i + j; // 例如矩陣元素等於
14            i+j
15        }
16
17    // 輸出矩陣
18    for (int i = 0; i < 3; i++) {
19        for (int j = 0; j < 3; j++) {
20            cout << matrix[i][j] << " ";
21        }
22    }
23 }
```

```

20     }
21     cout << endl;
22 }
23
24 return 0;
25 }
```

## Map

```

1 // map 是 C++ STL 中的關聯式容器，提供鍵值對的存儲。
2
3 #include <iostream>
4 #include <map>
5 using namespace std;
6
7 int main() {
8     map<string, int> m; // 告知一個 map, key 為
9                     string, value 為 int
10
11    // 插入鍵值對
12    m["apple"] = 5;
13    m["banana"] = 10;
14    m["orange"] = 7;
15
16    // 使用 key 來訪問數據
17    cout << "apple: " << m["apple"] << endl; // 輸出: apple: 5
18
19    // 遍歷 map
20    for (map<string, int>::iterator it =
21         m.begin(); it != m.end(); ++it) {
22        cout << it->first << ":" << it->second
23        << endl; // 輸出每個鍵值對
24    }
25
26    return 0;
27 }
```

## Stack

```

1 // stack 是後進先出的數據結構，適用於需要後進先出 (LIFO)
2             的場景。
3
4 #include <iostream>
5 #include <stack>
6 using namespace std;
7
8 int main() {
9     stack<int> s; // 告知一個整數型的 stack
10
11    // 插入數據
12    s.push(10);
13    s.push(20);
14    s.push(30);
15
16    // 讀取並移除棧頂數據
17 }
```

```

16     while (!s.empty()) {
17         cout << "Top: " << s.top() << endl; // 輸出當前棧頂數據
18         s.pop(); // 移除棧頂數據
19     }
20
21     return 0;
22 }
```

## Priority Queue

```

1 // priority_queue
    是一個優先隊列，預設情況下是大頂堆，根據優先級存儲數據。
2
3 #include <iostream>
4 #include <queue>
5 using namespace std;
6
7 int main() {
8     priority_queue<int> pq; // 告訴一個整數型的優先隊列，預設為大頂堆
9
10    // 插入數據
11    pq.push(30);
12    pq.push(10);
13    pq.push(20);
14
15    // 讀取並移除隊列頂部數據
16    while (!pq.empty()) {
17        cout << "Top: " << pq.top() << endl; // 輸出當前隊列頂部數據
18        pq.pop(); // 移除頂部數據
19    }
20
21    return 0;
22 }
```

## Queue

```

1 // queue 是先進先出的數據結構，適用於需要先進先出 (FIFO)
    的場景。
2
3 #include <iostream>
4 #include <queue>
5 using namespace std;
6
7 int main() {
8     queue<int> q; // 告訴一個整數型的隊列
9
10    // 插入數據
11    q.push(10);
12    q.push(20);
13    q.push(30);
14
15    // 讀取並移除隊首數據
16    while (!q.empty()) {
```

```

17        cout << "Front: " << q.front() << endl;
        // 輸出當前隊首數據
18        q.pop(); // 移除隊首數據
19    }
20
21    return 0;
22 }
```

## 建樹

```

1 // 使用 DFS 方式建立樹結構
2
3 #include <iostream>
4 #include <vector>
5 using namespace std;
6
7 const int maxn = 1005; // 最大節點數
8 vector<int> tree[maxn]; // 用於存儲樹的鄰接表
9 bool visited[maxn]; // 記錄節點是否已被訪問
10
11 // 建立樹
12 void addEdge(int u, int v) {
13     tree[u].push_back(v);
14     tree[v].push_back(u); // 因為是無向樹，雙向連接
15 }
16
17 // 深度優先搜索 (DFS) 建立樹
18 void dfs(int u) {
19     visited[u] = true;
20     for (int v : tree[u]) {
21         if (!visited[v]) {
22             cout << "Edge: " << u << " - " << v
23             << endl; // 輸出邊
24             dfs(v); // 繼續遞迴遍歷
25         }
26     }
27 }
28
29 int main() {
30     int n = 5; // 節點數
31
32     // 建立樹的邊
33     addEdge(1, 2);
34     addEdge(1, 3);
35     addEdge(2, 4);
36     addEdge(2, 5);
37
38     // 從節點1開始遍歷樹
39     dfs(1);
40
41 }
```

## 建圖

```

1 // 使用鄰接表建立圖
2
3 #include <iostream>
4 #include <vector>
5 using namespace std;
6
7 const int maxn = 1005; // 最大節點數
8 vector<pair<int, int>> graph[maxn]; //
    用於存儲圖的鄰接表，pair 表示(鄰接點，邊權重)
9
10 // 建立圖的邊 (有向圖)
11 void addEdge(int u, int v, int weight) {
12     graph[u].push_back(make_pair(v, weight));
    // 添加一條從 u 到 v 的有向邊，邊的權重為 weight
13 }
14
15 int main() {
16     int n = 5; // 節點數
17
18     // 建立圖的邊
19     addEdge(1, 2, 10); // 節點1到節點2，權重為10
20     addEdge(1, 3, 5);
21     addEdge(2, 4, 3);
22     addEdge(3, 5, 2);
23
24     // 輸出圖的鄰接表
25     for (int u = 1; u <= n; u++) {
26         cout << "Node " << u << " connects to:" << endl;
27         for (auto edge : graph[u]) {
28             cout << "(" << edge.first << ", " << "weight: " << edge.second << ")" << endl;
29         }
30         cout << endl;
31     }
32
33     return 0;
34 }
```

## Bit Index Tree (BIT) - Lowbit

```

1 // Bit Index Tree (BIT) - Lowbit
2 #define lowbit(x) ((x) & (-x))
3 int bit[2 * maxn] = {0}; // 宣告並初始化 bit 陣列
4
5 void add(int p, int v) {
6     while (p < 2 * maxn) {
7         bit[p] += v; // 更新當前節點的值
8         p += lowbit(p); // 跳到下一個需要更新的節點
9     }
10 }
11
12 int sum(int p) {
13     int ret = 0;
14     while (p > 0) {
15         ret += bit[p]; // 累加前綴和
16         p -= lowbit(p); // 跳到上一個節點
17     }
18 }
```

```

17     }
18     return ret;
19 }
```

## Kruskal 最小生成樹 (MST)

```

1 // Kruskal MST (最小生成樹)
2 #include <iostream>
3 #include <vector>
4 #include <algorithm>
5 #include <cmath>
6 using namespace std;
7
8 const int maxn = 1005; // 最大節點數
9
10 struct Point {
11     int x, y;
12 };
13
14 struct Edge {
15     int u, v;
16     double cost;
17     bool operator<(const Edge& e) const {
18         return cost < e.cost; // 按邊的權重排序
19     }
20 };
21
22 int parent[maxn], rank[maxn]; // 並查集
23 Point points[maxn]; // 存儲每個點的座標
24 Edge e[maxn * maxn]; // 存儲所有邊
25 int m; // 邊的數量
26
27 // 初始化並查集
28 void init(int n) {
29     for (int i = 0; i < n; i++) {
30         parent[i] = i;
31         rank[i] = 0;
32     }
33 }
34
35 // 路徑壓縮找根
36 int find(int u) {
37     if (parent[u] != u) {
38         parent[u] = find(parent[u]);
39     }
40     return parent[u];
41 }
42
43 // 合併兩個集合
44 bool uni(int u, int v) {
45     int root_u = find(u);
46     int root_v = find(v);
47
48     if (root_u != root_v) {
49         if (rank[root_u] > rank[root_v]) {
50             parent[root_v] = root_u;
51         } else {
52             parent[root_u] = root_v;
53             if (rank[root_u] == rank[root_v])
54                 rank[root_v]++;
55         }
56     }
57 }
```

```

51     } else if (rank[root_u] < rank[root_v])
52     {
53         parent[root_u] = root_v;
54     } else {
55         parent[root_v] = root_u;
56         rank[root_u]++;
57     }
58     return true;
59 }
60 }
61
62 // 計算兩個點之間的歐幾里得距離
63 double dist(int u, int v) {
64     int dx = points[u].x - points[v].x;
65     int dy = points[u].y - points[v].y;
66     return sqrt(dx * dx + dy * dy);
67 }
68
69 // Kruskal 最小生成樹算法
70 double kruskal(int n) {
71     init(n);
72     sort(e, e + m); // 按權重對邊排序
73
74     double ans = 0;
75     int edge_count = 0;
76     for (int i = 0; i < m; i++) {
77         int u = e[i].u, v = e[i].v;
78         if (uni(u, v)) { // 如果 u 和 v 不在同一集合中
79             ans += e[i].cost;
80             if (++edge_count == n - 1) break; // 當邊數達到 n-1 時停止
81         }
82     }
83     return ans; // 返回最小生成樹的總權重
84 }

```

## SPFA 最短路徑算法

```

1 // SPFA 最短路徑算法
2 int dist[maxn]; // 存儲到達每個節點的最短距離
3 vector<pair<int, int>> E[maxn]; // 鄰接表存儲圖
4 bool vis[maxn]; // 訪問標記
5 int out[maxn]; // 用來檢測負環
6
7 bool spfa(int s, int n) {
8     queue<int> Q;
9     Q.push(s); // 將起點s加入隊列
10    while (!Q.empty()) {
11        int u = Q.front();
12        Q.pop();
13        vis[u] = 0;
14        out[u]++;
15        if (out[u] > n) return false; // 出現負權重環
16        for (int j = 0; j < E[u].size(); j++) {
17            int v = E[u][j].first;

```

```

18                if (dist[v] > dist[u] +
19                    E[u][j].second) { // 有更短的路徑
20                    dist[v] = dist[u] +
21                        E[u][j].second;
22                    if (vis[v]) continue;
23                    vis[v] = 1;
24                    Q.push(v); // 將v加入隊列
25                }
26            }
27        }
28    }
29    return true;
30 }

```

## 連通集 (Disjoint Set)

```

1 // Disjoint Set 連通集 - 加權合併與路徑壓縮
2 int parent[maxn];
3 fill(parent, parent + maxn, -1); // 初始化 parent
4 // 陣列，表示每個節點是獨立的
5 void weighted_union2(int i, int j) {
6     int temp = parent[i] + parent[j];
7     if (parent[i] > parent[j]) {
8         parent[i] = j; // j為新根
9         parent[j] = temp;
10    } else {
11        parent[j] = i; // i為新根
12        parent[i] = temp;
13    }
14 }
15
16 int find2(int i) {
17     int root, trail, lead;
18     for (root = i; parent[root] >= 0; root =
19         parent[root]); // 找根
20     for (trail = i; trail != root; trail =
21         lead) { // 路徑壓縮
22         lead = parent[trail];
23         parent[trail] = root;
24     }
25     return root;
26 }

```

## Nim 遊戲 SG 函數

```

1 // Nim 遊戲 SG 函數
2 #define maxn 20+5
3 #define maxs 100+5
4 int sg[maxn];
5
6 void SG() {
7     int mex[maxs] = {}; // MEX表初始化
8     sg[0] = 0; // 初始化SG值
9     for (int i = 1; i < maxn; i++) {
10        memset(mex, 0, sizeof(mex)); // 清空MEX數組
11        for (int j = 0; j < i; j++) {

```

```

12         mex[sg[j] ^ sg[i - j - 1]] = 1; // Nim遊戲狀態轉移
13     }
14     int g = 0;
15     while (mex[g]) g++; // 找到最小的MEX
16     sg[i] = g; // 設置SG值
17 }
18 }
```

## KM Algorithm 最大匹配問題

```

1 // KM Algorithm (最大匹配問題)
2
3 int Lx[maxn], Ly[maxn]; // 頂標和底標
4 int W[maxn][maxn]; // 權重矩陣
5 bool S[maxn], T[maxn]; // 記錄 s 和 t 集合
6 int n; // 節點數
7
8 void update() {
9     int a = 1 << 30; // 初始最小差值設置為無限大
10    for (int i = 1; i <= n; i++) {
11        if (S[i]) {
12            for (int j = 1; j <= n; j++) {
13                if (!T[j]) {
14                    a = min(a, Lx[i] + Ly[j] -
15                            W[i][j]); // 更新最小差值
16                }
17            }
18        }
19        for (int i = 1; i <= n; i++) {
20            if (S[i]) Lx[i] -= a; // 更新頂標
21            if (T[i]) Ly[i] += a; // 更新底標
22        }
23    }
```

## Vector 建圖與多次 DFS 搜索

```

1 // Vector 建圖與多次 DFS 搜索
2 const int maxn = 1005;
3 vector<int> g[maxn], nodes[maxn]; // 建立鄰接表與節點集
4 int n, s, k, p[maxn]; // n為總節點數, s為起始節點, k為最大距離
5 bool c[maxn]; // 節點是否被訪問過
6
7 void dfs1(int u, int f, int dist) {
8     p[u] = f; // 記錄父節點
9     int nd = g[u].size();
10    if (nd == 1 && dist > k)
11        nodes[dist].push_back(u); // 遠距離節點加入
12    for (int i = 0; i < nd; i++) {
13        int v = g[u][i];
14        if (v != f) dfs1(v, u, dist + 1); // 遞迴進行DFS
15    }
16 }
```

```

15 }
16
17 int solve() {
18     int ans = 0;
19     memset(c, false, sizeof(c)); // 初始化訪問數組
20     for (int dist = (n - 1); dist > k; dist--) {
21         for (int i = 0; i < nodes[dist].size(); i++) {
22             int u = nodes[dist][i];
23             if (c[u]) continue;
24             int v = u;
25             for (int j = 0; j < k; j++) v = p[v];
26             dfs2(v, -1, 0); // 執行DFS, 標記被訪問過的節點
27             ans++;
28         }
29     }
30     return ans;
31 }
32
33 void dfs2(int u, int f, int d) {
34     c[u] = true; // 標記節點已訪問
35     for (int i = 0; i < g[u].size(); i++) {
36         int v = g[u][i];
37         if (v != f && d < k) dfs2(v, u, d + 1);
38     }
39 }
```

## Tree DP, DFS

```

1 #define min(a,b) ((a)<(b)?(a):(b))
2 #define max(a,b) ((a)>(b)?(a):(b))
3
4 // 深度優先搜索進行動態規劃計算
5 void dfs(int u) {
6     visited[u] = 1; // 標記節點已訪問
7     dp[u][0] = 0; // 當前節點不選擇
8     dp[u][1] = 1; // 當前節點選擇
9
10    // 遍歷所有相鄰節點
11    for (int i = 0; i < g[u].size(); i++) {
12        int v = g[u][i];
13        if (visited[v]) continue;
14        dfs(v); // 遞迴調用DFS
15        dp[u][0] += dp[v][1]; // 當前節點不選擇時, 子節點必須選擇
16        dp[u][1] += min(dp[v][0], dp[v][1]); // 當前節點選擇時, 子節點可以選擇或不選擇
17    }
18 }
19
20 int solve() {
21     int ans = 0;
22     memset(visited, 0, sizeof(visited)); // 初始化訪問數組
23     for (int i = 0; i < n; i++) {
```

```

24     if (visited[i]) continue;
25     dfs(i); // 對每個未訪問的節點進行DFS
26     ans += min(dp[i][0], dp[i][1]); // 計算最小選擇
27   }
28   return ans;
29 }
```

## Dijkstra - 最短路徑算法

```

1 // Dijkstra 最短路徑算法
2 Dijkstra(G) {
3   for each v in V {
4     d[v] = infinity; // 初始化所有點的距離
5   }
6   d[s] = 0; // 起點距離為0
7   S = {}; // 空集合S
8   Q = V; // 未處理的節點集合
9
10  while (Q is not empty) {
11    u = ExtractMin(Q); // 選擇距離最小的節點u
12    S = S U {u}; // 將u加入集合S
13    for each v in u->Adj[] {
14      if (d[v] > d[u] + w(u,v)) {
15        d[v] = d[u] + w(u,v); // 更新距離
16      }
17    }
18  }
19 }
```

## Kruskal 最小生成樹 (MST, Disjoint Set Union)

```

1 // Kruskal 算法 - 最小生成樹
2 Kruskal() {
3   T = {}; // 初始化空生成樹
4   for each v in V {
5     MakeSet(v); // 初始化每個節點為單獨的集合
6   }
7   sort E by increasing edge weight w; // 按邊權重排序
8   for each (u,v) in E (in sorted order) {
9     if FindSet(u) != FindSet(v) {
10       T = T U {{u,v}}; // 將邊加入生成樹
11       Union(FindSet(u), FindSet(v)); // 合併集合
12     }
13   }
14 }
```

## Priority Queue 操作

```

1 // 優先隊列的操作與應用
2 #include <cstdio>
```

```

3 #include <queue>
4 using namespace std;
5
6 class Item {
7 public:
8   int Q_num, Period, Time;
9   bool operator < (const Item& a) const {
10     return (Time > a.Time || (Time ==
11         a.Time && Q_num > a.Q_num));
12   }
13
14 int main() {
15   priority_queue<Item> pq;
16   char s[20];
17   while (scanf("%s", s) && s[0] != '#') {
18     Item item;
19     scanf("%d%d", &item.Q_num,
20           &item.Period);
21     item.Time = item.Period;
22     pq.push(item);
23   }
24   int K;
25   scanf("%d", &K);
26   while (K--) {
27     Item r = pq.top();
28     pq.pop();
29     printf("%d\n", r.Q_num);
30     r.Time += r.Period;
31     pq.push(r);
32   }
33 }
```

## 二分搜索算法

```

1 // 二分搜索算法與貪心飛機降落時間
2 int I, n, caseNo = 1, order[8];
3 double a[8], b[8], L, maxL;
4
5 double greedyLanding() {
6   double lastLanding = a[order[0]]; // 第一架飛機的降落時間
7   for (int i = 1; i < n; i++) {
8     double target = lastLanding + L;
9     if (target <= b[order[i]]) { // 下一架飛機可以按計畫降落
10       lastLanding = max(a[order[i]],
11                           target); // 更新降落時間
11     } else {
12       return 1; // 無法完成
13     }
14   }
15   return lastLanding - b[order[n - 1]]; // 返回結果
16 }
```

## Priority Queue 合併操作

```

1 // 使用優先隊列合併兩個數組
2 class Item {
3 public:
4     int Q_num, Period, Time;
5     Item(int q, int p, int t) : Q_num(q),
6           Period(p), Time(t) {} // 構造函數
7     bool operator<(const Item& a) const {
8         return (Time > a.Time || (Time ==
9             a.Time && Q_num > a.Q_num));
10    }
11 }
12
13 void merge(int* A, int* B, int* C, int k) {
14     priority_queue<Item> q;
15     for (int i = 0; i < k; i++) {
16         q.push(Item(A[i] + B[0], 0));
17     }
18     for (int i = 0; i < k; i++) {
19         Item item = q.top(); q.pop();
20         C[i] = item.sum; // 保存當前的最小值
21         int b = item.b;
22         if (b + 1 < k) {
23             q.push(Item(item.sum - B[b] + B[b +
24                 1], b + 1)); // 更新優先隊列
25         }
26     }
27 }
28 }
```

## 高斯消去法 (Gaussian Elimination)

```

1 // 高斯消去法計算矩陣的秩
2 int rank(Matrix A, int m, int n) {
3     int i = 0, j = 0;
4     while (i < m && j < n) {
5         int r = i;
6         for (int k = i; k < m; k++) {
7             if (A[k][j]) {
8                 r = k;
9                 break;
10            }
11        }
12        if (A[r][j]) {
13            if (r != i) {
14                for (int k = 0; k <= n; k++)
15                    swap(A[r][k], A[i][k]);
16            }
17            for (int u = i + 1; u < m; u++) {
18                if (A[u][j])
19                    for (int k = i; k <= n; k++)
20                        A[u][k] -= A[i][k];
21            }
22        }
23        i++;
24    }
25 }
```

```

22     }
23     j++;
24 }
25     return i; // 返回矩陣的秩
26 }
```

## 排列組合與對數計算

```

1 // 使用對數進行排列組合的計算
2 #include <csdio>
3 #include <cstring>
4 #include <cmath>
5
6 int main() {
7     freopen("d:\\uva10883\\10883_in.txt", "r",
8         stdin);
9     freopen("d:\\uva10883\\10883_out.txt", "w",
10        stdout);
11    int Cas;
12    scanf("%d", &Cas);
13    for (int j = 1; j <= Cas; j++) {
14        int N;
15        double Res = 0, a, C = 0;
16        scanf("%d", &N);
17        for (int i = 0; i < N; i++) {
18            scanf("%lf", &a);
19            double sum = log2(fabs(a));
20            if (i) {
21                C += log2(double(N - i) / i);
22                sum += C;
23            }
24            if (a < 0) {
25                Res -= pow(2, sum - (N - 1));
26            } else {
27                Res += pow(2, sum - (N - 1));
28            }
29        }
30        printf("Case #%d: %.3lf\n", j, Res);
31    }
32 }
```

## 過篩法與大數運算

```

1 // 使用過篩法進行質數篩選並處理大數運算
2 #include <iostream>
3 using namespace std;
4
5 const int maxn = 250000;
6 const int maxp = 22000 + 5;
7 int fp[maxp];
8 bool vis[maxn];
9
10 int main() {
11     long long a = 0, b = 1, tmp;
12     int n, count_p = 1;
```

```

13     for (int i = 2; count_p < maxp; ++i) {
14         tmp = a + b;
15         a = b;
16         b = tmp;
17
18         if (b > 1e10) {
19             b /= 10;
20             a /= 10;
21         }
22
23
24         if (!vis[i]) {
25             tmp = b;
26             while (tmp >= 1e9) {
27                 tmp /= 10;
28             }
29             fp[count_p++] = tmp;
30             for (int j = i * i; j < maxn; j += i) {
31                 vis[j] = true;
32             }
33         }
34     }
35
36     fp[1] = 2, fp[2] = 3;
37     while (~scanf("%d", &n)) {
38         printf("%d\n", fp[n]);
39     }
40
41     return 0;
42 }
```

## Python 過篩法與大數運算

```

1 maxn = 250000
2 maxp = 22000 + 5
3
4 # 初始化數組 fp 用來存儲質數的前幾位
5 fp = [0] * maxp
6 vis = [False] * maxn # 記錄某數字是否被標記過
7
8 a, b = 0, 1 # 初始的 Fibonacci 數列
9 count_p = 1 # 質數計數器
10
11 i = 2
12 while count_p < maxp:
13     tmp = a + b # 計算 Fibonacci 序列
14     a = b
15     b = tmp
16     if b > 1e10: # 如果 b 太大，對其進行縮小處理
17         b /= 10
18         a /= 10
19
20     if not vis[i]: # 如果 i 沒有被標記為非質數
21         tmp = b
22         while tmp >= 1e9: # 確保數字在合理範圍內
23             tmp /= 10
```

```

24
25     fp[count_p] = tmp # 記錄第 count_p 個質數
26     count_p += 1
27
28     j = i * i
29     while j < maxn:
30         vis[j] = True # 標記合數
31         j += i
32
33     i += 1
34
35 fp[1], fp[2] = 2, 3 # 手動設定前兩個質數
36
37 try:
38     while True:
39         n = int(input()) # 讀取輸入
40         print(fp[n]) # 輸出第 n 個質數
41     except EOFError:
42         pass # 捕捉到 EOF 後退出循環
```

## Flow Algorithm (流量算法)

```

1 // 使用 Ford-Fulkerson 演算法來解最大流問題。
2 // 利用增廣路徑不斷增加流量，直到無法找到更多的增廣路徑。
3
4 #include <iostream>
5 #include <limits.h>
6 #include <queue>
7 #include <vector>
8 using namespace std;
9
10 #define V 6 // 節點數，假設我們的圖有 6 個節點
11
12 // BFS 搜索增廣路徑，並填充 parent[] 用來存儲路徑
13 bool bfs(int rGraph[V][V], int s, int t, int
14           parent[]) {
15     bool visited[V]; // 記錄哪些節點已經被訪問
16     fill(visited, visited + V, false); // 初始化所有節點為未訪問
17
18     queue<int> q; // 使用 BFS 搜索
19     q.push(s);
20     visited[s] = true;
21     parent[s] = -1; // 將源節點的父節點設為 -1 表示根節點
22
23     // 進行標準的 BFS 搜索
24     while (!q.empty()) {
25         int u = q.front();
26         q.pop();
27
28         for (int v = 0; v < V; v++) {
29             if (!visited[v] && rGraph[u][v] > 0)
30                 { // 只考慮殘餘流量大於 0 的邊
31                     if (v == t) { // 如果找到增廣路徑
32                         parent[v] = u;
33                         return true;
34                     }
35                 }
36             }
37         }
38     }
39
40     return false;
41 }
```

```

33     q.push(v);
34     parent[v] = u;
35     visited[v] = true;
36   }
37 }
38
39 return false; // 沒有找到增廣路徑
40 }
41
42 // Ford-Fulkerson 演算法的主函數，計算 s 到 t 的最大流
43 int fordFulkerson(int graph[V][V], int s, int t) {
44   int u, v;
45   int rGraph[V][V]; // 殘餘圖
46   for (u = 0; u < V; u++)
47     for (v = 0; v < V; v++)
48       rGraph[u][v] = graph[u][v]; // 初始化殘餘圖
49
50   int parent[V]; // 用來儲存 BFS 找到的路徑
51   max_flow = 0; // 初始化最大流為 0
52
53   // 不斷找增廣路徑，直到找不到為止
54   while (bfs(rGraph, s, t, parent)) {
55     int path_flow = INT_MAX;
56
57     // 計算增廣路徑中的最小殘餘容量
58     for (v = t; v != s; v = parent[v]) {
59       u = parent[v];
60       path_flow = min(path_flow,
61                       rGraph[u][v]);
62     }
63
64     // 更新殘餘圖中的流量
65     for (v = t; v != s; v = parent[v]) {
66       u = parent[v];
67       rGraph[u][v] -= path_flow;
68       rGraph[v][u] += path_flow; // 反向邊增加流量
69     }
70
71     max_flow += path_flow; // 更新最大流
72   }
73
74   return max_flow; // 返回最大流量
75 }
```

## Max Clique Algorithm (最大團 算法)

```
1 // 最大團問題的求解：使用位元遮罩來計算最大團。  
2 // 最大團指的是在無向圖中，一個最大完全子圖。  
3  
4 const int MAXN = 50; // 節點數的最大限制  
5 int n, adj[MAXN][MAXN]; // n 為節點數, adj 為鄰接矩陣  
6 int maxCliqueSize = 0; // 用來記錄找到的最大團的大小  
7
```

```
8 // DFS 搜尋來判斷最大團的大小
9 void dfs(int depth, int *now, int size) {
10    if (depth == n) { // 如果已經遍歷完所有節點
11        maxCliqueSize = max(maxCliqueSize,
12                               size); // 更新最大團大小
13        return;
14    }
15    // 檢查當前節點 depth 是否可以加入到現有的團
16    for (int i = 0; i < size; i++) {
17        if (!adj[now[i]][depth]) return; // 不是完全圖，返回
18    }
19
20    now[size] = depth; // 將當前節點加入團
21    dfs(depth + 1, now, size + 1); // 繼續搜尋加入當前節點的情況
22    dfs(depth + 1, now, size); // 搜尋不加入當前節點的情況
23 }
24
25 // 主函數，計算最大團
26 int maxClique() {
27     int now[MAXN]; // 用來存儲當前的團
28     dfs(0, now, 0); // 從第 0 層開始搜尋
29     return maxCliqueSize; // 返回最大團的大小
30 }
```

## Miller-Rabin Primality Test (米勒-拉賓素性測試)

```
1 // 米勒-拉賓素數測試演算法：一種高效的隨機化素數判定方法。
2 // 利用快速幕與模數運算來測試一個數是否為素數。
3
4 typedef long long ll;
5
6 // 快速模數乘法，防止溢出
7 ll mulmod(ll a, ll b, ll mod) {
8     ll res = 0;
9     a %= mod;
10    while (b) {
11        if (b & 1) res = (res + a) % mod; // 如果b的最低位為1，則加上a
12        a = (a + a) % mod; // a * 2
13        b >>= 1; // 右移 b
14    }
15    return res;
16 }
17
18 // 快速幕演算法，計算 base^exp % mod
19 ll powmod(ll base, ll exp, ll mod) {
20     ll res = 1;
21     base %= mod;
22     while (exp) {
23         if (exp & 1) res = mulmod(res, base,
24             mod); // 如果exp的最低位為1，乘上base
```

```

24     base = mulmod(base, base, mod); // base *
25         base
26     exp >= 1; // 右移 exp
27 }
28 }  

29  

30 // 米勒-拉賓測試演算法
31 bool miller_rabin(ll n, int k) {
32     if (n < 2) return false; // 如果 n < 2 不是素數
33     if (n != 2 && n % 2 == 0) return false; // 偶數不是素數
34
35     ll d = n - 1;
36     while (d % 2 == 0) d /= 2; // 分解 d 使得 n-1 =
37         d * 2^r
38
39     // 重複測試 k 次
40     while (k--) {
41         ll a = 2 + rand() % (n - 3); //
42             隨機選擇一個測試基數 a
43         ll x = powmod(a, d, n); // 計算 a^d % n
44
45         if (x == 1 || x == n - 1) continue; //
46             可能是素數
47
48         while (d != n - 1) {
49             x = mulmod(x, x, n); // 平方，計算 x^2 % n
50             d *= 2;
51
52             if (x == 1) return false; // 合數
53             if (x == n - 1) break; // 可能是素數
54         }
55
56         if (x != n - 1) return false; // 如果 x
57             最終不等於 n-1 則為合數
58     }
59
60     return true; // 通過所有測試，n 可能是素數
61 }
```

## Convex Hull Algorithm (凸包算法)

1 // 凸包演算法：使用 *Andrew's monotone chain*  
演算法來計算一組點的凸包。

```

2 // 凸包是能包圍所有點的最小凸多邊形。
3
4 #include <vector>
5 #include <algorithm>
6 using namespace std;
7
8 struct Point {
9     int x, y; // 定義一個點 (x, y)
10    bool operator<(const Point& p) const {
11        return x < p.x || (x == p.x && y <
12            p.y); // 按照 x 進行排序，若 x 相同則按 y 排序
13    }
14
15 // 計算向量 OA 和 OB 的叉積，用來判斷轉向
16 int cross(const Point &O, const Point &A,
17           const Point &B) {
18    return (A.x - O.x) * (B.y - O.y) - (A.y -
19        O.y) * (B.x - O.x);
20 }
21
22 // 計算一組點的凸包，返回凸包的點集
23 vector<Point> convex_hull(vector<Point>& P) {
24    int n = P.size(), k = 0;
25    vector<Point> H(2 * n); // 最多有 2*n 個點
26
27    // 構建下半凸包
28    for (int i = 0; i < n; ++i) {
29        while (k >= 2 && cross(H[k-2], H[k-1],
30            P[i]) <= 0) k--; // 移除不滿足條件的點
31        H[k++] = P[i]; // 將當前點加入到凸包
32    }
33
34    // 構建上半凸包
35    for (int i = n - 2, t = k + 1; i >= 0; --i)
36    {
37        while (k >= t && cross(H[k-2], H[k-1],
38            P[i]) <= 0) k--; // 移除不滿足條件的點
39        H[k++] = P[i]; // 將當前點加入到凸包
40    }
41    H.resize(k - 1); // 刪除最後一個重複點
42    return H; // 返回凸包的點集
43 }
```