# ACM-ICPC-REFERENCE

Searlese

April 2018

# Contents

# 1 CodingResources

## 1.1 IntToBinary

```cpp
typedef long long int lli;

lli bitsInInt(lli n) {
    return floor(log2(n) + 1LL);
}

vector<int> intToBitsArray(lli n) {
    n = abs(n);
    if (!n) {
        vector<int> v;
        return v;
    }
    int length = bitsInInt(n);
    int lastPos = length - 1;
    vector<int> v(length);
    for (lli i = lastPos, j = 0; i > -1LL; i--, j++) {
        lli aux = (n >> i) & 1LL;
        v[j] = aux;
    }
    return v;
}
```

## 1.2 IOoptimizationCPP

```cpp
int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
}
```

## 1.3 PrintVector

```cpp
void printv(vector<int> v) {
    if (v.size() == 0) {
        cout << "[]" << endl;
        return;
    }
    cout << "[" << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << ", " << v[i];
    }
    cout << "]" << endl;
}
```

## 1.4 PriorityQueueOfClass

```cpp
struct Object {
    char first;
    int second;
};

int main() {
    auto cmp = [](const Object& a, const Object& b) {return a.second >
    ↪  b.second;};
    priority_queue<Object, vector<Object>, decltype(cmp)> pq(cmp);
    vector<Object> v = {{'c',3}, {'a', 1}, {'b', 2}};
    sort(v.begin(), v.end(), cmp);
    return 0;
}
```

## 1.5 ReadLineCpp

```cpp
string input() {
    string ans;
    // cin >> ws; // eats all whitespaces.
    getline(cin, ans);
    return ans;
}
```

## 1.6 SortListOfClass

```python
class MyObject:
    def __init__(self, first, second):
        self.first = first
        self.second = second


li = [MyObject('c', 3), MyObject('a', 1), MyObject('b', 2)]

li.sort(key=lambda x: x.first, reverse=False)
```

## 1.7 SortVectorOfClass

```cpp
struct Object {
    char first;
    int second;
};

int main() {
    auto cmp = [](const Object& a, const Object& b) {return a.second >
    ↪  b.second;};
```

```
    vector<Object> v = {{'c',3}, {'a', 1}, {'b', 2}};
    sort(v.begin(), v.end(), cmp);
    printv(v);
    return 0;
}
```

### 1.8 SplitString

```
vector<string> split(string str, char token) {
    stringstream test(str);
    string segment;
    vector<std::string> seglist;

    while (std::getline(test, segment, token))
        seglist.push_back(segment);
    return seglist;
}
```

## 2 Graphs

### 2.1 CycleInDirectedGraph

```
int n; // max node id >= 1
vector<vector<int>> ady; // ady.resize(n + 1)
vector<int> vis; // vis.resize(n + 1)
vector<vector<int>> cycles;
vector<int> cycle;
bool flag = false;
int rootNode = -1;

bool hasDirectedCycle(int u) {
    vis[u] = 1;
    for (auto &v : ady[u]) {
        if (v == u || vis[v] == 2)
            continue;
        if (vis[v] == 1 || hasDirectedCycle(v)) {
            if (rootNode == -1)
                rootNode = v, flag = true;
            if (flag) {
                cycle.push_back(u);
                if (rootNode == u)
                    flag = false;
            }
            return true;
        }
    }
    vis[u] = 2;
    return false;
}

bool hasDirectedCycle() {
    vis.clear();
    for (int u = 1; u <= n; u++)
        if (!vis[u]) {
            cycle.clear();
            if (hasDirectedCycle(u))
                cycles.push_back(cycle);
        }
    return cycles.size() > 0;
}
```

### 2.2 CycleInUndirectedGraph

```
int n; // max node id >= 1
vector<vector<int>> ady; // ady.resize(n + 1)
vector<bool> vis; // vis.resize(n + 1)
vector<vector<int>> cycles;
vector<int> cycle;
bool flag = false;
int rootNode = -1;

bool hasUndirectedCycle(int u, int prev) {
    vis[u] = true;
    for (auto &v : ady[u]) {
        if (v == u || v == prev)
            continue;
        if (vis[v] || hasUndirectedCycle(v, u)) {
            if (rootNode == -1)
                rootNode = v, flag = true;
            if (flag) {
                cycle.push_back(u);
                if (rootNode == u)
                    flag = false;
            }
            return true;
        }
    }
    return false;
}

bool hasUndirectedCycle() {
```

```
    vis.clear();
    for (int u = 1; u <= n; u++)
        if (!vis[u]) {
            cycle.clear();
            if (hasUndirectedCycle(u, -1))
                cycles.push_back(cycle);
        }
    return cycles.size() > 0;
}
```

## 2.3 FloodFill

```
int n, m, oldColor = 0, color = 1;
vector<vector<int>> mat;

vector<vector<int>> movs = {
    {1, 0},
    {0, 1},
    { -1, 0},
    {0, -1}
};

void floodFill(int i, int j) {
    if (i >= mat.size() || i < 0 || j >= mat[i].size() || j < 0 ||
    ↪  mat[i][j] != oldColor)
        return;
    mat[i][j] = color;
    for (auto move : movs)
        floodFill(i + move[1], j + move[0]);
}

void floodFill() {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (mat[i][j] == oldColor)
                floodFill(i, j);
}
```

## 2.4 IsBipartite

```
int n; // max node id >= 1
vector<vector<int>> ady; // ady.resize(n + 1)

bool isBipartite() {
    vector<int> color(n + 1, -1);
    for (int s = 1; s <= n; s++) {
        if (color[s] > -1)
            continue;
        color[s] = 0;
        queue<int> q; q.push(s);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int &v : ady[u]) {
                if (color[v] < 0)
                    q.push(v), color[v] = !color[u];
                if (color[v] == color[u])
                    return false;
            }
        }
    }
    return true;
}
```

## 2.5 TopologicalSort

```
int n; // max node id >= 1
vector<vector<int>> ady; // ady.resize(n + 1)
vector<int> vis; // vis.resize(n + 1)
vector<int> toposorted;

bool toposort(int u) {
    vis[u] = 1;
    for (auto &v : ady[u]) {
        if (v == u || vis[v] == 2)
            continue;
        if (vis[v] == 1 || !toposort(v))
            return false;
    }
    vis[u] = 2;
    toposorted.push_back(u);
    return true;
}

bool toposort() {
    vis.clear();
    for (int u = 1; u <= n; u++)
        if (!vis[u])
            if (!toposort(u))
                return false;
    return true;
}
```

## 2.6 UnionFind

```
struct UnionFind {
    vector<int> dad, size;
    int n;
    UnionFind(int N) : n(N), dad(N), size(N, 1) {
        while (--N) dad[N] = N;
    }

    int root(int u) {
        if (dad[u] == u) return u;
        return dad[u] = root(dad[u]);
    }

    bool areConnected(int u, int v) {
        return root(u) == root(v);
    }

    void join(int u, int v) {
        int Ru = root(u), Rv = root(v);
        if (Ru == Rv) return;
        --n, dad[Ru] = Rv;
        size[Rv] += size[Ru];
    }

    int getSize(int u) {
        return size[root(u)];
    }

    int numberOfSets() {
        return n;
    }

};
```