

ICPC template

目次

第 I 部 C++

1 C++ Preparation

- 1.1 C++ Compiler 2
- 1.2 C++ Execution 2
- 1.3 C++ Template 2

2 データ構造

- 2.1 stack 2
- 2.2 queue 2
- 2.3 priority queue 2
- 2.4 map 2
- 2.5 set 2
- 2.6 tuple 2
- 2.7 string 2
- 2.8 Union-Find 3
- 2.9 BIT (Fenwick Tree) 3

3 Graph

- 3.1 二部グラフ判定 3
- 3.2 深さ優先探索 (再帰関数型) 3
- 3.3 深さ優先探索 (スタック型) 3
- 3.4 幅優先探索 4
- 3.5 ダイクストラ法 4
- 3.6 ベルマンフォード法 4

4 探索

- 4.1 二分探索 4
- 4.2 bit 全探索 5
- 4.3 順列全探索 5

5 その他

- 5.1 素数判定 5
- 5.2 繰返し二乗法 5
- 5.3 余剰を取る繰返し二乗法 5
- 5.4 mod の逆元 5
- 5.5 modint の使い方 5

第 II 部 Python

1 Python Preparation

- 1.1 Python Compiler 6
- 1.2 Python Execution 6

2 データ構造

- 2.1 UnionFind 6

3 Graph

- 3.1 深さ優先探索 (再帰関数型) 6
- 3.2 深さ優先探索 (スタック型) 6
- 3.3 幅優先探索 7
- 3.4 ダイクストラ法 7
- 3.5 ベルマンフォード法 7

4 探索

- 4.1 二分探索 8
- 4.2 bit 全探索 8
- 4.3 順列全探索 8

5 その他

- 5.1 素数判定 8

第I部

C++

1 C++ Preparation

1.1 C++ Compiler

```
g++ -std=c++17 test.cpp
```

1.2 C++ Execution

```
./test.out <input> output
```

1.3 C++ Template

```
#include <bits/stdc++.h>
#include <atcoder/all>
using namespace std;
using namespace atcoder;
using ll = long long;
using ull = unsigned long long;
using Graph = vector<vector<int>>;
// int 2*10e9
// long long 9*10e18
// unsigned long long 1*10e19
constexpr int INF = 1e9;
constexpr ll LLINF = 4e18;
#define for_(i,a,b) for(int i=(a);i<(b);++i)
#define rep(i, n) for_(i, 0, n)
#define all(a) (a).begin(), (a).end()
#define rall(a) (a).rbegin(), (a).rend()

//4方向
int dx[4] = {1, 0, -1, 0};
int dy[4] = {0, -1, 0, 1};

//8方向
int ddx[8] = {1,1,1,0,0,-1,-1,-1};
int ddy[8] = {1,0,-1,1,-1,1,0,-1};

int main() {
    return 0;
}
```

2 データ構造

2.1 stack

```
stack<int> st;
// [1, 2, 3]を追加
st.push(1);
st.push(2);
cout << st.top() << endl; // 2
st.push(3);
st.pop(); // 3を削除
st.pop(); // 2を削除
cout << st.top() << endl; // 1
if(st.empty()) // 空ならtrue
```

2.2 queue

```
queue<int> que;
// [1, 2, 3]を追加
que.push(1);
que.push(2);
cout << que.front() << endl; // 1
que.push(3);
que.pop(); // 1を削除
que.pop(); // 2を削除
cout << que.front() << endl; // 3
```

2.3 priority queue

```
// 最大値が先頭にくるキュー
priority_queue<int> pq;
// 最小値が先頭にくるキュー
priority_queue<int, vector<int>, greater<int>>
    pq;
// [1, 3, 5, 6]を追加
pq.push(1);
pq.push(3);
cout << pq.top() << endl; // 3
pq.push(5);
pq.push(6);
pq.pop(); // 6を削除
cout << pq.top() << endl; // 5
```

2.4 map

```
map<string, int> mp;
mp["haru"] = 12;
mp["taro"] = 13;
cout << mp["haru"] << endl; // 12
```

2.5 set

```
set<int> st;
// [1, 2]を追加
st.insert(1);
st.insert(2);
st.count(1); // 1が含まれていたら1を返す
st.erase(1); // 1を削除
cout << *st.begin() << endl; // 2
```

2.6 tuple

```
tuple<int, string, long long> tp;
tp = {20, "kindai", 100000};
// 要素のアクセス
cout << get<0>(tp) << endl; // 20
cout << get<1>(tp) << endl; // kindai
```

2.7 string

```
string S = "TUNA";
// 文字列の長さを取得
cout << S.size() << endl; // 4
// 文字列が空か判定
if(S.empty()) // 空ならtrue
// 文字列の分割 1文字目以降を取得
cout << S.substr(1) << endl; // UNA
cout << S.substr(1, 2) << endl; // UN
// 文字列の削除 1文字目以降を削除
cout << S.erase(1) << endl; // T
```

2.8 Union-Find

```
// Union-Find
// グリッドでUFを使う時,(x,y)に対して使うなら
// (x-1)*W+(y-1)でハッシュ化できる.
struct UnionFind {
    vector<int> par, rank, siz;
    // 構造体の初期化
    UnionFind(int n) : par(n,-1), rank(n,0),
        siz(n,1) { }
    // 根を求める
    int root(int x) {
        if (par[x]==-1) return x;
        else return par[x] = root(par[x]);
    }
    // x と y が同じグループに属するか (= 根が一致するか)
    bool issame(int x, int y) {
        return root(x)==root(y);
    }
    // x を含むグループと y を含むグループを併合する
    bool unite(int x, int y) {
        int rx = root(x), ry = root(y);
        if (rx==ry) return false;
        // union by rank
        if (rank[rx]<rank[ry]) swap(rx, ry);
        par[ry] = rx; // ry を rx の子とする
        if (rank[rx]==rank[ry]) rank[rx]++;
        siz[rx] += siz[ry];
        return true;
    }
    // x を含む根付き木のサイズを求める
    int size(int x) {
        return siz[root(x)];
    }
};

// union-find木がいくつの連結成分からなるかを返す
long long partial(UnionFind tree){
    long long n = tree.siz.size();
    vector<bool> seen(n, false);
    long long ans = 0;
    for (long long i = 0; i < n; i++){
        if (seen[tree.root(i)]) continue;
        seen[tree.root(i)] = true;
        ans++;
    }
    return ans;
}

// 無向グラフ
// Gがいくつの連結成分からなるかを返す
long long partial(Graph &G){
```

```
    long long siz = G.size();
    UnionFind ki(siz);
    for (long long i = 0; i < siz; i++){
        long long siz2 = G[i].size();
        for (long long j = 0; j < siz2; j++){
            ki.unite(i, G[i][j]);
        }
    }
    long long ret = partial(ki);
    return ret;
}
```

2.9 BIT (Fenwick Tree)

```
// 数列a(a[0],a[1],...,a[n-1])についての区間和と点更新を扱う
// 区間和,点更新,二分探索はO(log{n})
class BIT {
public:
    //データの長さ
    ll n;
    //データの格納先
    vector<ll> a;
    //コンストラクタ
    BIT(ll n):n(n),a(n+1,0){}

    //a[i]にxを加算する
    void add(ll i,ll x){
        i++;
        if(i==0) return;
        for(ll k=i;k<=n;k+=(k & -k)){
            a[k]+=x;
        }
    }

    //a[i]+a[i+1]+...+a[j]を求める
    ll sum(ll i,ll j){
        return sum_sub(j)-sum_sub(i-1);
    }

    //a[0]+a[1]+...+a[i]を求める
    ll sum_sub(ll i){
        i++;
        ll s=0;
        if(i==0) return s;
        for(ll k=i;k>0;k-=(k & -k)){
            s+=a[k];
        }
        return s;
    }
}
```

```
//a[0]+a[1]+...+a[i]>=xとなる最小のiを求める(
// 任意のkでa[k]>=0が必要)
ll lower_bound(ll x){
    if(x<=0){
        //xが0以下の場合は該当するものなし→0を返す
        return 0;
    }else{
        ll i=0;ll r=1;
        // 最大としてありうる区間の長さ取得する
        // n以下の最小の二乗のべき(
        // BITで管理する数列の区間で最大のもの)を求める
        while(r<n) r=r<<1;
        //区間の長さは調べるごとに半分になる
        for(int len=r;len>0;len=len>>1) {
            //その区間を採用する場合
            if(i+len<n && a[i+len]<x){
                x-=a[i+len];
                i+=len;
            }
        }
        return i;
    }
};
```

3 Graph

3.1 二部グラフ判定

3.2 深さ優先探索 (再帰関数型)

```
// 深さ優先探索
vector<bool> seen;
void dfs(const Graph &G, int v) {
    seen[v] = true; // v を訪問済にする

    // v から行ける各頂点 next_v について
    for (auto next_v : G[v]) {
        // next_v が探索済だったらスルー
        if (seen[next_v]) continue;
        dfs(G, next_v); // 再帰的に探索
    }
}
```

3.3 深さ優先探索 (スタック型)

```
// 深さ優先探索
stack<int> st;
st.push(start);
while (!st.empty()) {
    int v = st.top(); st.pop();
    if (seen[v]) continue;
    seen[v] = true;
    for (auto next_v : G[v]) {
        if (seen[next_v]) continue;
        st.push(next_v);
    }
}
```

3.4 幅優先探索

```
// 幅優先探索
// 全頂点を「未訪問」に初期化
vector<int> dist(N, -1);
queue<int> que;

// 初期条件 (頂点 0 を初期ノードとする)
dist[0] = 0;
que.push(0); // 0 を橙色頂点にする

// BFS 開始 (キューが空になるまで探索を行う)
while (!que.empty()) {
    // キューから先頭頂点を取り出す
    int v = que.front();
    que.pop();

    // v から辿れる頂点をすべて調べる
    for (int nv : G[v]) {
        // すでに発見済みの頂点は探索しない
        if (dist[nv] != -1) continue;

        // 新たな白色頂点 nv について距離情報を更新してキューに追加する
        dist[nv] = dist[v] + 1;
        que.push(nv);
    }
}
```

3.5 ダイクストラ法

```
// 負の重みがない場合の最短経路を求める

// 辺を表す構造体
```

```
struct Edge{
    long long to;
    long long cost;
    // その他、必要な情報があれば要素を追加
};
// 隣接リストを表す型
using Gpaph=vector<vector<Edge>>;
// 距離と頂点のペアを表す型
using Pair = pair<long long, long long>;
// 暫定距離を格納する配列
vector<long long> dist;
const long long INF = 1LL << 60;

void dijkstra(const Graph& G, vector<long long>& dist, long long start){
    priority_queue<Pair, vector<Pair>, greater<Pair>> Q;
    dist.assign(G.size(), INF);
    // dist[start]=0をして、qに(0,start)をpush
    Q.emplace(dist[start]=0, start);

    while(!Q.empty()){
        Pair q=Q.top();
        Q.pop();
        long long d=q.first;
        long long v=q.second;

        if(d>dist[v]) continue;

        for(const auto& edge:G[v]){
            long long nextdist = d+edge.cost;
            if(nextdist<dist[edge.to]){
                Q.emplace(dist[edge.to]=nextdist, edge.to);
            }
        }
    }
}
```

3.6 ベルマンフォード法

```
// 負の重みがある場合の最短経路を求める
struct Edge {
    long long from;
    long long to;
    long long cost;
};
using Edges = vector<Edge>;
const long long INF = 1LL << 60;

/* bellman_ford(Es,s,t,dis)
```

```
入力: 全ての辺Es, 頂点数V, 開始点 s, 最短経路を記録するdis
出力: 負の閉路が存在するなら true
計算量: O(|E|V)
副作用: dis が書き換えられる

*/
bool bellman_ford(const Edges &Es, int V, int s, vector<long long> &dis) {
    dis.resize(V, INF);
    dis[s] = 0;
    int cnt = 0;
    while (cnt < V) {
        bool end = true;
        for (auto e : Es) {
            if (dis[e.from] != INF && dis[e.from] + e.cost < dis[e.to]) {
                dis[e.to] = dis[e.from] + e.cost;
                end = false;
            }
        }
        if (end) break;
        cnt++;
    }
    return (cnt == V);
}
```

4 探索

4.1 二分探索

```
vector<int> a = { 1,4,4,7,7,8,8,11,13,19};
// lower_bound:key以上の値が初めて現れる位置
auto iter = lower_bound(all(a),4);
// key以上の最小の値を出力
cout << *iter << endl; // 4
// key以上の最小の値が初めて現れる位置を出力
cout << a.begin() - iter << endl; // 1

// upper_bound:
// keyより大きい値が初めて現れる位置
auto iter1 = upper_bound(all(a), 4);
// keyより大きい最小の値を出力
cout << *iter1 << endl; // 7
// keyより大きい最小の値が初出する位置を出力
cout << a.begin() - iter1 << endl; // 3
```

4.2 bit 全探索

```
for(int bit = 0; bit < (1 << n); bit++){
    // データ数 n の bit 全探索
    for(int i = 0; i < n; i++){
        if(bit & (1 << i)){
            // 処理を書く
        }
    }
}
```

4.3 順列全探索

```
vector<int> A(4);
A = {1, 2, 3, 4};
do{
    // ここに処理を書く
}while(next_permutation(A.begin(), A.end()));
```

5 その他

5.1 素数判定

```
// エラトステネスの篩  $O(N)$ 
vector<bool> Eratosthenes(int N) {
    // テーブル
    vector<bool> isprime(N+1, true);

    // 0, 1 は予めふるい落としておく
    isprime[0] = isprime[1] = false;

    // ふるい
    for (int p = 2; p <= N; ++p) {
        // 合成数であるものはスキップする
        if (!isprime[p]) continue;

        // p 以外の p の倍数から素数ラベルを剥奪
        for (int q = p * 2; q <= N; q += p) {
            isprime[q] = false;
        }
    }

    // 1 以上 N 以下の整数が素数かどうか
    return isprime;
}
```

5.2 繰返し二乗法

```
long long pow(long long x, long long n) {
    long long ret = 1;
    while (n > 0) {
        // n の最下位 bit が 1 ならば  $x^{(2^i)}$  をかける
        if (n & 1) ret *= x;
        x *= x;
        n >>= 1; // n を 1bit 左にずらす
    }
    return ret;
}
```

5.5 modint の使い方

```
using mint = modint998244353;

int main(){
    mint ans = 4321;
    ans /= 9876;
    // 4321 / 9876 mod 998244353 を出力
    cout << ans.val() << endl;
}
```

5.3 余剰を取る繰返し二乗法

```
const int MOD = 1000000007;
long long pow(long long x, long long n) {
    long long ret = 1;
    while (n > 0) {
        // n の最下位 bit が 1 ならば  $x^{(2^i)}$  をかける
        if (n & 1) ret = ret * x % MOD;
        x = x * x % MOD;
        n >>= 1; // n を 1bit 左にずらす
    }
    return ret;
}
```

5.4 mod の逆元

```
// mod. m での a の逆元  $a^{-1}$  を計算する
long long modinv(long long a, long long m) {
    long long b = m, u = 1, v = 0;
    while (b) {
        long long t = a / b;
        a -= t * b; swap(a, b);
        u -= t * v; swap(u, v);
    }
    u %= m;
    if (u < 0) u += m;
    return u;
}
```

第II部

Python

1 Python Preparation

1.1 Python Compiler

1.2 Python Execution

2 データ構造

2.1 UnionFind

```
class UnionFind:
    def __init__(self, n):
        self.par = [-1] * n
        self.rank = [0] * n
        self.siz = [1] * n

    def root(self, x):
        if self.par[x] == -1:
            return x
        else:
            self.par[x] = self.root(self.par[x])
            return self.par[x]

    def issame(self, x, y):
        return self.root(x) == self.root(y)

    def unite(self, x, y):
        rx = self.root(x)
        ry = self.root(y)
        if rx == ry:
            return False
        if self.rank[rx] < self.rank[ry]:
            rx, ry = ry, rx
        self.par[ry] = rx
        if self.rank[rx] == self.rank[ry]:
            self.rank[rx] += 1
        self.siz[rx] += self.siz[ry]
        return True

    def size(self, x):
        return self.siz[self.root(x)]

    def partial(tree):
        n = len(tree.siz)
```

```
        seen = [False] * n
        ans = 0
        for i in range(n):
            if not seen[tree.root(i)]:
                seen[tree.root(i)] = True
                ans += 1
        return ans
```

```
def partial_graph(G):
    siz = len(G)
    uf = UnionFind(siz)
    for i in range(siz):
        for j in G[i]:
            uf.unite(i, j)
    return partial(uf)
```

```
# 使用例
# グラフ G を隣接リストとして表現
G = [
    [1, 2], # ノード0の隣接ノード
    [0, 2], # ノード1の隣接ノード
    [0, 1], # ノード2の隣接ノード
    [4],   # ノード3の隣接ノード
    [3],   # ノード4の隣接ノード
]

# グラフの連結成分の数を計算
num_connected_components = partial_graph(G)
print(num_connected_components) # 出力: 2
```

3 Graph

3.1 深さ優先探索 (再帰関数型)

```
def dfs(G, v, seen):
    seen[v] = True # v を訪問済にする

    # v から行ける各頂点 next_v について
    for next_v in G[v]:
        # next_v が探索済だったらスルー
        if seen[next_v]:
            continue
        dfs(G, next_v, seen) # 再帰的に探索
```

```
# 使用例
# グラフ G を隣接リストとして表現
G = [
    [1, 2], # ノード0の隣接ノード
```

```
    [0, 2], # ノード1の隣接ノード
    [0, 1, 3], # ノード2の隣接ノード
    [2, 4], # ノード3の隣接ノード
    [3]     # ノード4の隣接ノード
]
```

```
# 頂点の訪問状態を保持するリスト
seen = [False] * len(G)
```

```
# ノード0からDFSを開始
dfs(G, 0, seen)
```

```
# 結果の出力
print(seen)
# 出力: [True, True, True, True, True]
```

3.2 深さ優先探索 (スタック型)

```
def dfs_iterative(G, start):
    # 頂点の訪問状態を保持するリスト
    seen = [False] * len(G)
    # スタックの初期化
    st = [start]
```

```
    while st:
        v = st.pop()
        if seen[v]:
            continue
        seen[v] = True
        for next_v in G[v]:
            if not seen[next_v]:
                st.append(next_v)
```

```
    return seen
```

```
# 使用例
# グラフ G を隣接リストとして表現
G = [
    [1, 2], # ノード0の隣接ノード
    [0, 2], # ノード1の隣接ノード
    [0, 1, 3], # ノード2の隣接ノード
    [2, 4], # ノード3の隣接ノード
    [3]     # ノード4の隣接ノード
]
```

```
# ノード0からDFSを開始
seen = dfs_iterative(G, 0)
```

```
# 結果の出力
print(seen) # 出力: [True, True, True, True, True]
```

3.3 幅優先探索

```
from collections import deque

def bfs(G, start):
    N = len(G)
    # 全頂点を「未訪問」に初期化
    dist = [-1] * N
    que = deque()

    # 初期条件 (頂点 start を初期ノードとする)
    dist[start] = 0
    que.append(start) # start を橙色頂点にする

    # BFS 開始 (キューが空になるまで探索を行う)
    while que:
        # キューから先頭頂点を取り出す
        v = que.popleft()

        # v から辿れる頂点をすべて調べる
        for nv in G[v]:
            # すでに発見済みの頂点は探索しない
            if dist[nv] != -1:
                continue

            # 新たな白色頂点 nv について距離情報を更新してキューに追加する
            dist[nv] = dist[v] + 1
            que.append(nv)

    return dist

# 使用例
# グラフ G を隣接リストとして表現
G = [
    [1, 2], # ノード0の隣接ノード
    [0, 2], # ノード1の隣接ノード
    [0, 1, 3], # ノード2の隣接ノード
    [2, 4], # ノード3の隣接ノード
    [3] # ノード4の隣接ノード
]

# ノード0からBFSを開始
distances = bfs(G, 0)

# 結果の出力
print(distances) # 出力: [0, 1, 1, 2, 3]
```

3.4 ダイクストラ法

```
import heapq

class Edge:
    def __init__(self, to, cost):
        self.to = to
        self.cost = cost

def dijkstra(G, start):
    INF = float('inf')
    dist = [INF] * len(G)
    dist[start] = 0
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))

    while priority_queue:
        d, v = heapq.heappop(priority_queue)

        if d > dist[v]:
            continue

        for edge in G[v]:
            nextdist = d + edge.cost
            if nextdist < dist[edge.to]:
                dist[edge.to] = nextdist
                heapq.heappush(priority_queue, (nextdist, edge.to))

    return dist

# 使用例
# グラフ G を隣接リストとして表現
G = [
    [Edge(1, 2), Edge(2, 4)], # ノード0
                                # の隣接ノードとコスト
    [Edge(2, 1), Edge(3, 7)], # ノード1
                                # の隣接ノードとコスト
    [Edge(3, 3)], # ノード2
                  # の隣接ノードとコスト
    [] # ノード3
       # の隣接ノードとコスト
]

# ノード0からダイクストラ法を開始
start_node = 0
distances = dijkstra(G, start_node)

# 結果の出力
print(distances) # 出力: [0, 2, 3, 6]
```

3.5 ベルマンフォード法

```
class Edge:
    def __init__(self, from_node, to_node, cost):
        self.from_node = from_node
        self.to_node = to_node
        self.cost = cost

def bellman_ford(edges, V, start):
    INF = float('inf')
    dist = [INF] * V
    dist[start] = 0

    for i in range(V):
        updated = False
        for edge in edges:
            if dist[edge.from_node] != INF and \
               dist[edge.from_node] + edge.cost < dist[edge.to_node]:
                dist[edge.to_node] = dist[edge.from_node] + edge.cost
                updated = True
        if not updated:
            break

    # Check for negative weight cycles
    for edge in edges:
        if dist[edge.from_node] != INF and \
           dist[edge.from_node] + edge.cost < dist[edge.to_node]:
            return True, dist # Negative weight cycle detected

    return False, dist

# 使用例
edges = [
    Edge(0, 1, 2),
    Edge(0, 2, 4),
    Edge(1, 2, 1),
    Edge(1, 3, 7),
    Edge(2, 3, 3),
    Edge(3, 4, -5),
    Edge(4, 1, 2)
]

V = 5 # グラフの頂点数
start_node = 0
has_negative_cycle, distances = bellman_ford(edges, V, start_node)

if has_negative_cycle:
    print("負の閉路があります")
else:
```

```
print("最短距離:", distances)
```

4 探索

4.1 二分探索

```
import bisect

a = [1, 4, 4, 7, 7, 8, 8, 11, 13, 19]

# lower_bound: key以上の値が初めて現れる位置
key = 4
iter_idx = bisect.bisect_left(a, key)
print(a[iter_idx]) # 4
print(iter_idx)    # 1

# upper_bound:
# keyより大きい値が初めて現れる位置
iter1_idx = bisect.bisect_right(a, key)
print(a[iter1_idx]) # 7
print(iter1_idx)    # 3
```

4.2 bit 全探索

```
n = 3 # データ数

for bit in range(1 << n):
    # データ数 n の bit 全探索
    for i in range(n):
        if bit & (1 << i):
            # 処理を書く
            print(f"bit: {bin(bit)}, i: {i}")
```

4.3 順列全探索

```
import itertools

A = [1, 2, 3, 4]

# permutationsを使って全ての順列を生成
for perm in itertools.permutations(A):
    # ここに処理を書く
    print(perm)
```

5 その他

5.1 素数判定

```
def eratosthenes(N):
    # テーブル
    isprime = [True] * (N + 1)

    # 0, 1 は予めふるい落としておく
    isprime[0] = isprime[1] = False

    # ふるい
    for p in range(2, N + 1):
        # 合成数であるものはスキップする
        if not isprime[p]:
            continue

        # p 以外の p の倍数から素数ラベルを剥奪
        for q in range(p * 2, N + 1, p):
            isprime[q] = False

    # 1 以上 N 以下の整数が素数かどうか
    return isprime

# 使用例
N = 30
isprime = eratosthenes(N)
print(isprime)

# 素数のリストを取得
primes = [i for i, prime in enumerate(isprime)
           if prime]
print(primes)
```