Contents

1 ./headers/headers/headers.h 2

2 ./strings/trie/trie.cpp 3

3 ./estructuras/fenwick/fenwickTree.cpp 4

4 ./trees/segment/lazySegmentTree.cpp 5

5 ./trees/segment/segmentTree.cpp 8

6 ./math/simpsonsMethod/simpsonsMethod.cpp 11

7 ./graphs/dinic/dinic.cpp 12

8 ./graphs/dijsktra/dijsktra.cpp 14

9 ./graphs/dfs/dfsRecursive.cpp 15

10 ./graphs/dfs/dfsIterative.cpp 16

11 ./graphs/lca/lca.cpp 17

12 ./graphs/kruskal/kruskal.cpp 19

13 ./graphs/unionFind/unionFind.cpp 21

14 ./graphs/bfs/bfs.cpp 23

15 ./graphs/bellmanFord/bellmanFord.cpp 24

1 ./headers/headers/headers.h

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

typedef vector<vi> graph;
typedef vector<vii> wgraph;


#ifndef declaraciones_h
#define declaraciones_h

#define rep(i,n) for(int i = 0; i < n; i++)
#define repx(i,a,b) for(int i = a; i < b; i++)
#define invrep(i,a,b) for(int i = b; i-- > (int)a)

#define pb push_back
#define eb emplace_back
#define ppb pop_back

#define lg(x) (31 - __builtin_clz(x))
#define lgg(x) (63 - __buitlin_clzll(x))
#define gcd __gcd

//ios::sync_with_stdio(0); cin.tie(0);
//cout.setf(ios::fixed); cout.precision(4);

#define debugx(x) //cerr<<#x<<": "<<x<<endl
#define debugv(v) //cerr<<#v<<":";rep(i,(int)v.size())cerr<<" "<<v[i];cerr<<endl
#define debugm(m) //cerr<<#m<<endl;rep(i,(int)m.size()){cerr<<i<<":";rep(j,(int)m[i].size()
#define debugmp(m) //cerr<<#m<<endl;rep(i,(int)m.size()){cerr<<i<<":";rep(j,(int)m[i].size(
#define print(x) copy(x.begin(), x.end(), ostream_iterator<int>(cout, \")), cout<<endl

#endif
```

2 ./strings/trie/trie.cpp

```
#include "../../headers/headers/headers.h"

class Trie
{
  private:
    vector<unordered_map<char, int>> nodes;
    int next;

  public:
    Trie()
    {
        nodes.eb();
        next = 1;
    }

    bool build(string s)
    {
        int i = 0;
        int v = 0;
        while (i < s.size())
        {
            if (nodes[v].find(s[i]) == nodes[v].end())
            {
                nodes.eb();
                v = nodes[v][s[i]] = next;
                i++;
                next++;
            }
            else
            {
                v = nodes[v][s[i]];
                i++;
            }
        }
    }
};
```

3 ./estructuras/fenwick/fenwickTree.cpp

```cpp
#include "../../headers.h"

class FenwickTree
{
  private:
    vi ft;

  public:
    FenwickTree(int n)
    {
        ft.assign(n + 1, 0);
    }

    int psq(int b)
    {
        int sum = 0;
        for (; b; b -= (b & -b))
            sum += ft[b];
        return sum;
    }

    int rsq(int a, int b)
    {
        return psq(b) - psq(a - 1);
    }

    void add(int k, int v)
    {
        for (; k < ft.size(); k += (k & -k))
            ft[k] += v;
    }

    void range_add(int i, int j, int v)
    {
        add(i, v);
        add(j + 1, -v);
    }
}
```

4 ./trees/segment/lazySegmentTree.cpp

```
#include "../../headers/headers/headers.h"

struct RSQ // Range sum query
{
    static intt const neutro = 0;
    static intt op(intt x, intt y)
    {
        return x + y;
    }
    static intt
    lazy_op(int i, int j, intt x)
    {
        return (j - i + 1) * x;
    }
};

struct RMinQ // Range minimun query
{
    static intt const neutro = 1e18;
    static intt op(intt x, intt y)
    {
        return min(x, y);
    }
    static intt
    lazy_op(int i, int j, intt x)
    {
        return x;
    }
};

template <class t>
class SegTreeLazy
{
    vector<intt> arr, st, lazy;
    int n;

    void build(int u, int i, int j)
    {
        if (i == j)
        {
            st[u] = arr[i];
            return;
        }
        int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
        build(l, i, m);
        build(r, m + 1, j);
        st[u] = t::op(st[l], st[r]);
    }

    void propagate(int u, int i, int j, intt x)
    {
        // nota, las operaciones pueden ser un and, or, ..., etc.
        st[u] += t::lazy_op(i, j, x); // incrementar el valor (+)
        // st[u] = t::lazy_op(i, j, x); // setear el valor
        if (i != j)
        {
            // incrementar el valor
            lazy[u * 2 + 1] += x;
            lazy[u * 2 + 2] += x;
            // setear el valor
            //lazy[u * 2 + 1] = x;
            //lazy[u * 2 + 2] = x;
        }
        lazy[u] = 0;
    }

    intt query(int a, int b, int u, int i, int j)
    {
        if (j < a or b < i)
            return t::neutro;
        int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
        if (lazy[u])
            propagate(u, i, j, lazy[u]);
        if (a <= i and j <= b)
            return st[u];
        intt x = query(a, b, l, i, m);
        intt y = query(a, b, r, m + 1, j);
        return t::op(x, y);
    }

    void update(int a, int b, intt value,
                int u, int i, int j)
    {
        int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
        if (lazy[u])
            propagate(u, i, j, lazy[u]);
        if (a <= i and j <= b)
            propagate(u, i, j, value);
        else if (j < a or b < i)
            return;
        else
        {
            update(a, b, value, l, i, m);
            update(a, b, value, r, m + 1, j);
            st[u] = t::op(st[l], st[r]);
        }
    }

  public:
    SegTreeLazy(vector<intt> &v)
    {
        arr = v;
        n = v.size();
        st.resize(n * 4 + 5);
        lazy.assign(n * 4 + 5, 0);
        build(0, 0, n - 1);
    }

    intt query(int a, int b)
    {
        return query(a, b, 0, 0, n - 1);
    }

    void update(int a, int b, intt value)
    {
        update(a, b, value, 0, 0, n - 1);
    }
};
```

5 ./trees/segment/segmentTree.cpp

```
#include "../../headers/headers/headers.h"

// Se requiere un struct para el nodo (ej: prodsgn).
// Un nodo debe tener tres constructores:
//     Aridad 0: Construye el neutro de la operación
//     Aridad 1: Construye un nodo hoja a partir del input
//     Aridad 2: Construye un nodo según sus dos hijos
//
// Construcción del segment tree:
//     Hacer un arreglo de nodos (usar ctor de aridad 1).
//     ST<miStructNodo> miSegmentTree(arregloDeNodos);
// Update:
//     miSegmentTree.set_point(indice, miStructNodo(input));
// Query:
//     miSegmentTree.query(l, r) es inclusivo exclusivo y da un nodo. Usar la info del nodo


// Logic And Query
struct ANDQ
{
    intt value;
    ANDQ() { value = -1ll; }
    ANDQ(intt x) { value = x; }
    ANDQ(const ANDQ &a,
         const ANDQ &b)
    {
        value = a.value & b.value;
    }
};

// Interval Product (LiveArchive)
struct prodsgn {
    int sgn;
    prodsgn() {sgn = 1;}
    prodsgn(int x) {
        sgn = (x > 0) - (x < 0);
    }
    prodsgn(const prodsgn &a,
            const prodsgn &b) {
        sgn = a.sgn*b.sgn;
    }
};

// Maximum Sum (SPOJ)
```

```
struct maxsum {
    int first, second;
    maxsum() {first = second = -1;}
    maxsum(int x) {
        first = x; second = -1;
    }
    maxsum(const maxsum &a,
           const maxsum &b) {
        if (a.first > b.first) {
            first = a.first;
            second = max(a.second,
                         b.first);
        } else {
            first = b.first;
            second = max(a.first,
                         b.second);
        }
    }
    int answer() {
        return first + second;
    }
};

// Range Minimum Query
struct rminq {
    int value;
    rminq() {value = INT_MAX;}
    rminq(int x) {value = x;}
    rminq(const rminq &a,
          const rminq &b) {
        value = min(a.value,
                    b.value);
    }
};

template <class node>
class ST
{
    vector<node> t;
    int n;

  public:
    ST(vector<node> &arr)
    {
        n = arr.size();
        t.resize(n * 2);
        copy(arr.begin(), arr.end(), t.begin() + n);
        for (int i = n - 1; i > 0; --i)
            t[i] = node(t[i << 1], t[i << 1 | 1]);
    }

    // 0-indexed
    void set_point(int p, const node &value)
    {
        for (t[p += n] = value; p > 1; p >>= 1)
            t[p >> 1] = node(t[p], t[p ^ 1]);
    }

    // inclusive exclusive, 0-indexed
    node query(int l, int r)
    {
        node ansl, ansr;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1)
        {
            if (l & 1)
                ansl = node(ansl, t[l++]);
            if (r & 1)
                ansr = node(t[--r], ansr);
        }
        return node(ansl, ansr);
    }
};
```

6 ./math/simpsonsMethod/simpsonsMethod.cpp

```
#include "../../headers.h"
//Numerical Integration of f in interval [a,b]

double simpsons_rule(function<double(double)> f, double a, double b)
{
    double c = (a + b) / 2;
    double h3 = abs(b - a) / 6;
    return h3 * (f(a) + 4 * f(c) + f(b));
}

double simpsons_rule(function<double(double)> f, double a, double b, int n)
{
    //n sets the precision for the result
    double ans = 0;
    double step = 0, h = (b - a) / n;
    rep(i, n)
    {
        ans += simpsons_rule(f, step, step + h);
        step += h;
    }
    return ans;
}
```

7 ./graphs/dinic/dinic.cpp

```
#include "../../headers/headers/headers.h"
class Dinic
{
    struct edge
    {
        int to, rev;
        ll f, cap;
    };

    vector<vector<edge>> g;
    vector<ll> dist;
    vector<int> q, work;
    int n, sink;

    bool bfs(int start, int finish)
    {
        dist.assign(n, -1);
        dist[start] = 0;
        int head = 0, tail = 0;
        q[tail++] = start;
        while (head < tail)
        {
            int u = q[head++];
            for (const edge &e : g[u])
            {
                int v = e.to;
                if (dist[v] == -1 and e.f < e.cap)
                {
                    dist[v] = dist[u] + 1;
                    q[tail++] = v;
                }
            }
        }
        return dist[finish] != -1;
    }

    ll dfs(int u, ll f)
    {
        if (u == sink)
            return f;
        for (int &i = work[u]; i < (int)g[u].size(); ++i)
        {
            edge &e = g[u][i];
            int v = e.to;
            if (e.cap <= e.f or dist[v] != dist[u] + 1)
                continue;
            ll df = dfs(v, min(f, e.cap - e.f));
            if (df > 0)
            {
                e.f += df;
                g[v][e.rev].f -= df;
                return df;
            }
        }
        return 0;
    }

  public:
    Dinic(int n)
    {
        this->n = n;
        g.resize(n);
        dist.resize(n);
        q.resize(n);
    }

    void add_edge(int u, int v, ll cap)
    {
        edge a = {v, (int)g[v].size(), 0, cap};
        edge b = {u, (int)g[u].size(), 0, 0}; //Poner cap en vez de 0 si la arista es bidir
        g[u].pb(a);
        g[v].pb(b);
    }

    ll max_flow(int source, int dest)
    {
        sink = dest;
        ll ans = 0;
        while (bfs(source, dest))
        {
            work.assign(n, 0);
            while (ll delta = dfs(source, LLONG_MAX))
                ans += delta;
        }
        return ans;
    }
};
```

8 ./graphs/dijsktra/dijsktra.cpp

```cpp
#include "../../headers/headers/headers.h"
//g has vectors of pairs of the form (w, index)
int dijsktra(wgraph g, int start, int end)
{
    int n = g.size();
    vi cost(n, 1e9); //~INT_MAX/2
    priority_queue<ii, greater<ii>> q;

    q.emplace(0, start);
    cost[start] = 0;
    while (not q.empty())
    {
        int u = q.top().second, w = q.top().first;
        q.pop();

        for (auto v : g[u])
        {
            if (cost[v.second] > v.first + w)
            {
                cost[v.second] = v.first + w;
                q.emplace(cost[v.second], v.second);
            }
        }
    }

    return cost[end];
}
```

9 ./graphs/dfs/dfsRecursive.cpp

```
#include "../../headers/headers/headers.h"
//Recursive (create visited filled with 1s)
void dfs_r(graph &g, vi &visited, int u)
{
    cout << u << '\n';
    visited[u] = 0;

    for (int v : g[u])
        if (visited[v])
            dfs_r(g, visited, v);
}
```

10 ./graphs/dfs/dfsIterative.cpp

```
#include "../../headers/headers/headers.h"
//Iterative
void dfs_i(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    stack<int> s;

    s.emplace(start);
    visited[start] = 0;

    while (not s.empty())
    {
        int u = s.top();
        s.pop();

        cout << u << '\n';

        for (int v : g[u])
            if (visited[v])
            {
                s.emplace(v);
                visited[v] = 0;
            }
    }
}
```

11 ./graphs/lca/lca.cpp

```
#include "../../headers/headers/headers.h"
class LcaTree
{
    int n;
    vi parent;
    vi level;
    vi root;
    graph P;
public:
    LcaTree(int n){
        this->n = n;
        parent.assign(n,-1);
        level.assign(n,-1);
        P.assign(n,vi(lg(n)+1,-1));
        root.assign(n,-1);
    }
    void addLeaf(int index, int par){
        parent[index] = par;
        level[index] = level[par] + 1;
        P[index][0] = par;
        root[index] = root[par];
        for(int j=1; (1<<j) < n; ++j){
            if(P[index][j-1] != -1)
                P[index][j] = P[P[index][j-1]][j-1];
        }
    }
    void addRoot(int index){
        parent[index] = index;
        level[index] = 0;
        root[index] = index;
    }
    int lca(int u, int v){
        if(root[u] != root[v] || root[u] == -1)
            return -1;
        if(level[u] < level[v])
            swap(u,v);
        int dist = level[u] - level[v];
        while(dist != 0){
            int raise = lg(dist);
            u = P[u][raise];
            dist -= (1<<raise);
        }
        if(u == v)
            return u;
        for(int j = lg(n); j>=0; --j){
            if(P[u][j] != -1 && P[u][j] != P[v][j]){
                u=P[u][j];
                v=P[v][j];
            }
        }
        return parent[u];
    }
};
```

12 ./graphs/kruskal/kruskal.cpp

```cpp
#include "../../headers/headers/headers.h"
struct edge
{
    int u, v;
    ll w;
    edge(int u, int v, ll w) : u(u), v(v), w(w) {}

    bool operator<(const edge &o) const
    {
        return w < o.w;
    }
};

class Kruskal
{
  private:
    ll sum;
    vi p, rank;

  public:
    Kruskal(int n, vector<edge> E)
    {
        sum = 0;
        p.resize(n);
        rank.assign(n, 0);
        rep(i, n) p[i] = i;
        sort(E.begin(), E.end());
        for (auto &e : E)
            UnionSet(e.u, e.v, e.w);
    }
    int findSet(int i)
    {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
    }
    bool isSameSet(int i, int j)
    {
        return findSet(i) == findSet(j);
    }
    void UnionSet(int i, int j, ll w)
    {
        if (not isSameSet(i, j))
        {
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y])
```

```
                p[y] = x;
            else
                p[x] = y;

            if (rank[x] == rank[y])
                rank[y]++;

            sum += w;
        }
    }
    ll mst_val()
    {
        return sum;
    }
};
```

13 ./graphs/unionFind/unionFind.cpp

```
#include "../../headers/headers/headers.h"
class UnionFind
{
  private:
    int numSets;
    vi p, rank, setSize;

  public:
    UnionFind(int n)
    {
        numSets = n;
        rank.assign(n, 0);
        setSize.assign(n, 1);
        p.resize(n);
        rep(i, n) p[i] = i;
    }
    int findSet(int i)
    {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
    }
    bool isSameSet(int i, int j)
    {
        return findSet(i) == findSet(j);
    }
    void UnionSet(int i, int j)
    {
        if (not isSameSet(i, j))
        {
            numSets--;
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y])
            {
                p[y] = x;
                setSize[x] += setSize[y];
            }
            else
            {
                p[x] = y;
                setSize[y] += setSize[x];
                if (rank[x] == rank[y])
                    rank[y]++;
            }
        }
    }
    int numSets()
    {
        return numSets;
    }
    int setOfSize(int i)
    {
        return setSize[i];
    }
};
```

14 ./graphs/bfs/bfs.cpp

```cpp
#include "../../headers/headers/headers.h"

void bfs(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    queue<int> q;

    q.emplace(start);
    visited[start] = 0;
    while (not q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v : g[u])
        {
            if (visited[v])
            {
                q.emplace(v);
                visited[v] = 0;
            }
        }
    }
}
```

15 ./graphs/bellmanFord/bellmanFord.cpp

```cpp
#include "../../headers/headers/headers.h"
bool bellman_ford(wgraph &g, int start)
{
    int n = g.size();
    vector<int> dist(n, 1e9); //~INT_MAX/2
    dist[start] = 0;
    rep(i, n - 1) rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        dist[v] = min(dist[v], dist[u] + w);
    }

    bool hayCicloNegativo = false;
    rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        if (dist[v] > dist[u] + w)
            hayCicloNegativo = true;
    }

    return hayCicloNegativo;
}
```