

# Little Piplup

Gratus907(Wonseok Shin), Coffeetea(Jinje Han), DhDroid(Donghyun Son)

## Contents

<b>1 Settings</b>	<b>2</b>	<b>6 Shortest Path</b>	<b>7</b>
1.1 C++	2	6.1 Dijkstra	7
<b>2 Data Structures</b>	<b>2</b>	6.2 Bellman Ford	8
2.1 Segment Tree	2	6.3 Floyd-Warshall	8
2.2 Fenwick Tree	2	<b>7 Dynamic</b>	<b>8</b>
2.3 Disjoint Set Union	2	7.1 Largest Sum Subarray	8
<b>3 Mathematics</b>	<b>3</b>	7.2 Knapsack	8
3.1 Useful Mathematical Formula	3	7.3 Longest Common Subsequence	8
3.2 Number of Integer Partition	3	7.4 Edit Distance	9
3.3 Extended Euclidean Algorithm	3	<b>8 String</b>	<b>9</b>
3.4 Fast Modulo Exponentiation	3	8.1 KMP Algorithm	9
3.5 Miller-Rabin Primality Testing	3	8.2 Manacher's Algorithm	9
3.6 Pollard-Rho Factorization	4	8.3 Trie	9
3.7 Euler Totient	4	8.4 Rabin-Karp Hashing	9
3.8 Modular Multiplicative Inverse	4	8.5 Aho-Corasick Algorithm	9
<b>4 Geometry</b>	<b>5</b>	<b>9 Miscellaneous</b>	<b>10</b>
4.1 Closest Pair Problem	5	9.1 Useful Bitwise Functions in C++	10
4.2 Smallest Enclosing Circle	5	9.2 List of Useful Numbers	10
4.3 Convex Hull	5	<b>10 Debugging Checkpoints</b>	<b>11</b>
<b>5 Graphs</b>	<b>5</b>		
5.1 Topological Sorting	5		
5.2 Bipartite Checking	5		
5.3 Union-Find	6		
5.4 MST Kruskal Algorithm	6		
5.5 MST Prim Algorithm	6		
5.6 Ford-Fulkerson Algorithm	7		

# 1 Settings

## 1.1 C++

# 2 Data Structures

## 2.1 Segment Tree

To deal with queries on intervals, we use segment tree.

```
int arr[SIZE];
int tree[TREE_SIZE];
int makeTree(int left,int right,int node)
{
    if (left == right)
        return tree[node] = arr[left];
    int mid = (left + right) / 2;
    tree[node] += makeTree(left, mid, node * 2);
    tree[node] += makeTree(mid + 1,right, node * 2 +1);
    return tree[node];
}
```

Updating segment Tree

```
void update(int left,int right,int node, int change_node ,int diff)
{
    if (!(left <= change_node &&change_node <= right))
        return; //No effect on such nodes.
    tree[node] += diff; // This part must be changed with tree function.
    if (left != right)
    {
        int mid = (left + right) / 2;
        update(left, mid, node * 2, change_node, diff);
        update(mid +1,right, node * 2 +1, change_node, diff);
    }
}
```

Answering queries via segment tree

```
/*
Our Search range : start to end
Node has range left to right
We may answer query in O(log n) time.
*/
int Query(int node, int left, int right, int start, int end)
{
    if (right < start || end < left)
```

```
        return 0; //Node is out of range
    if (start <= left && right <= end)
        return tree[node]; //If node is completely in range
    int mid = (left + right) / 2;
    return Query(node * 2, left, mid, start, end)
        +Query(node*2+1,mid+1,right,start,end);
}
```

## 2.2 Fenwick Tree

## 2.3 Disjoint Set Union

## 3 Mathematics

### 3.1 Useful Mathematical Formula

- Catalan Number : Number of valid parentheses strings with  $n$  pairs

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

### 3.2 Number of Integer Partition

```
def partitions(n):
    parts = [1]+[0]*n
    for t in range(1, n+1):
        for i, x in enumerate(range(t, n+1)):
            parts[x] += parts[i]
    return parts[n]
```

### 3.3 Extended Euclidean Algorithm

```
int Extended_Euclid(int a, int b, int *x, int *y)
{
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }
    int x1, y1;
    int EEd = Extended_Euclid(b%a, a, &x1, &y1);
    *x = y1 - (b/a) * x1;
    *y = x1;
    return EEd;
}
```

### 3.4 Fast Modulo Exponentiation

Calculating  $x^y \bmod p$  in  $\mathcal{O}(\log y)$  time.

```
/*
Fast Modulo Exponentiation algorithm
Runs on  $\mathcal{O}(\log y)$  time,
calculate  $x^y \bmod p$ 
*/

ll modpow(ll x, ll y, ll p)
{

```

```
    ll res = 1;
    x = x % p;
    while (y > 0)
    {
        if (y & 1)
            res = (res*x) % p;
        y = y>>1;
        x = (x*x) % p;
    }
    return res;
}
```

### 3.5 Miller-Rabin Primality Testing

Base values of  $a$  chosen so that results are tested to be correct up to  $10^{14}$ .

```
bool MRwitness(ll n, ll s, ll d, ll a)
{
    ll x = modpow(a, d, n);
    ll y = -1;

    while (s)
    {
        y = (x * x) % n;
        if (y == 1 && x != 1 && x != n-1)
            return false;
        x = y;
        s--;
    }
    return (y==1);
}

bool Miller_Rabin(ll n)
{
    if (n<2)
        return false;
    if (n == 2 || n == 3 || n == 5 || n == 7 || n == 11 || n == 13 || n == 17)
        return true;
    if (n%2 == 0 || n%3 == 0 || n%5 == 0)
        return false;
    ll d = (n-1) / 2;
    ll s = 1;
    while (d%2==0)
    {

```

```

        d /= 2;
        s++;
    }
    int candidate[7] = {2,3,5,7,11,13,17};
    bool result = true;
    for (auto i : candidate)
    {
        result = result & MRwitness(n,s,d,i);
        if (!result)
            break;
    }
    return result;
}

```

### 3.6 Pollard-Rho Factorization

```

ll PollardRho(ll n)
{
    srand (time(NULL));
    if (n==1)
        return n;
    if (n % 2 == 0)
        return 2;
    ll x = (rand()%(n-2))+2;
    ll y = x;
    ll c = (rand()%(n-1))+1;
    ll d = 1;
    while (d==1)
    {
        x = (modpow(x, 2, n) + c + n)%n;
        y = (modpow(y, 2, n) + c + n)%n;
        y = (modpow(y, 2, n) + c + n)%n;
        d = gcd(abs(x-y), n);
        if (d==n)
            return PollardRho(n);
    }
    return d;
}

```

### 3.7 Euler Totient

Calculating number of integers below  $n$  which is coprime with  $n$ .

```

#define ll long long
ll euler_phi(ll n)

```

```

{
    ll p=2;
    ll ephi = n;
    while(p*p<=n)
    {
        if (n%p == 0)
            ephi = ephi/p * (p-1);
        while(n%p==0)
            n/=p;
        p++;
    }
    if (n!=1)
    {
        ephi /= n;
        ephi *= (n-1);
    }
    return ephi;
}

```

### 3.8 Modular Multiplicative Inverse

```

ll modinv(ll x, ll p)
{
    return modpow(x,p-2,p);
}

```

## 4 Geometry

### 4.1 Closest Pair Problem

### 4.2 Smallest Enclosing Circle

### 4.3 Convex Hull

## 5 Graphs

### 5.1 Topological Sorting

Topological sorting with dfs

```
vector <int> graph[V];
bool visited[V];
vector <int> sorted;

void dfs(int root)
{
    visited[root] = 1;
    for (auto it:graph[root])
    {
        if (!visited[it])
            dfs(it);
    }
    sorted.push_back(root);
}

int main()
{
    int n, m;
    scanf("%d%d",&n,&m);
    for (int i = 0; i<m; i++)
    {
        int small,big;
        scanf("%d%d",&small,&big);
        graph[small].push_back(big);
    }
    for (int i = 1; i<=n; i++)
        if (!visited[i])
            dfs(i);
    reverse(sorted.begin(),sorted.end()); // must reverse!
}
```

### 5.2 Bipartite Checking

```
vector <int> graph[20200];
vector <pair <int, int>> edge;
bool visited[20200];
int color[20200];

void dfs(int root)
```

```

{
    for (auto it:graph[root])
    {
        if (!visited[it])
        {
            visited[it] = true;
            color[it] = color[root]%2+1;
            dfs(it);
        }
    }
}

bool is_bipartite(vector <int> &mygraph, int v, int e)
{
    for (int i = 1; i<=v; i++)
    {
        if (!visited[i])
        {
            visited[i] = 1;
            color[i] = 1;
            dfs(i);
        }
    }
    for (int i = 0; i<e; i++)
        if (color[edge[i].first]==color[edge[i].second])
            return false;
    return true;
}

```

### 5.3 Union-Find

```

//Union Find with Pass compression
int parent[V]; //initialize with i
int level[V]; //initialize with 1
int Find(int x) // Finding root node of x
{
    if (x==parent[x])
        return x;
    else
        return parent[x] = Find(parent[x]);
}

void Union(int x, int y)
{

```

```

    int u = Find(x);
    int v = Find(y);
    if (u == v)
        return;
    if (level[u] > level[v])
    {
        Union(v,u);
        return;
    }
    parent[u] = v;
    if (level[u]==level[v])
        level[v]++;
}

```

### 5.4 MST Kruskal Algorithm

```

//Based on Union-Find implementation
//O(E log E) if path-compressed Union Find.
int Kruskal()
{
    int mstlen = 0;
    sort(edgelist.begin(),edgelist.end());
    for (auto it:edgelist)
    {
        if (Find(it.s)==Find(it.e)) // Cycle Detection
            continue;
        else
        {
            Union(it.s,it.e);
            mstlen += it.w;
        }
    }
    return mstlen;
}

```

### 5.5 MST Prim Algorithm

```

vector <pii> Tree[101010];
// Note that we use {weight, destination} pair here.
// This is to use priority_queue!
bool visit[101010];
priority_queue <pii, vector<pii>, greater<pii>> pq;
void add(int i)
{

```

```

    visit[i] = true;
    for (auto it:Tree[i])
        pq.push(it);
}

int Prim(int start)
{
    int mstlen = 0;
    add(start);
    while(!pq.empty())
    {
        int cur = pq.top().second;
        int weight = pq.top().first;
        pq.pop();
        if (visit[cur])
            continue;
        else
        {
            mstlen+=weight;
            add(cur);
        }
    }
    return mstlen;
}

```

## 5.6 Ford-Fulkerson Algorithm

## 6 Shortest Path

### 6.1 Dijkstra

$\mathcal{O}(E \log V)$  Single-Start-Shortest-Path.

Not working for graph with minus weight.

```

const int INF = 987654321;
const int MX = V+something;
struct Edgeout
{
    int dest, w;
    bool operator<(const Edgeout &p) const
    {
        return w > p.w;
    }
};

vector <Edgeout> edgelist[MX];
int V, E, start;
int dist[MX];

bool relax(Edgeout edge, int u)
{
    bool flag = 0;
    int v = edge.dest, w = edge.w;
    if (dist[v] > dist[u] + w && (dist[u] != INF))
    {
        flag = true;
        dist[v] = dist[u]+w;
    }
    return flag;
}

int dijkstra()
{
    fill(dist,dist+MX,INF);
    dist[start] = 0;
    priority_queue<Edgeout> pq;
    pq.push({start,0});
    while(!pq.empty())
    {
        Edgeout x = pq.top();
        int v = x.dest, w = x.w;

```

```

        pq.pop();
        if (w>dist[v])
            continue;
        for (auto ed : edgelist[v])
            if (relax(ed,v))
                pq.push({ed.dest,dist[ed.dest]});
    }
}

```

## 6.2 Bellman Ford

$\mathcal{O}(EV)$  Single-Start-Shortest-Path.

Not working for graph with minus cycle  $\rightarrow$  must detect.

```

struct Edge
{
    int u, v, w;
};

vector<Edge> edgelist;
int V, E;
int dist[V+1];

bool relax_all_edge()
{
    bool flag = false;
    for (auto it:edgelist)
    {
        int u = it.u, v = it.v, w = it.w;
        if (dist[v] > dist[u] + w && (dist[u]!=INF))
        {
            flag = true;
            dist[v] = dist[u]+w;
        }
    }
    return flag;
}

int bellman_ford()
{
    fill(dist,dist+V+2,INF);
    dist[1] = 0;
    for (int i = 0; i<V-1; i++)
    {

```

```

        relax_all_edge();
    }
    if (relax_all_edge())
        return -1;
    else
        return 0;
}

```

## 6.3 Floyd-Warshall

Works on adjacency matrix, in  $\mathcal{O}(V^3)$ .

```

int d[120][120];
int n;
void Floyd_Warshall()
{
    for (int i = 1; i<=n; i++)
        for (int j = 1; j<=n; j++)
            d[j][k] = MIN(d[j][k],d[j][i]+d[i][k]);
}

```

## 7 Dynamic

### 7.1 Largest Sum Subarray

Computes sum of largest sum subarray in  $\mathcal{O}(N)$

```

void consecsum(int n)
{
    dp[0] = number[0];
    for (int i = 1; i<n; i++)
        dp[i] = MAX(dp[i-1]+number[i],number[i]);
}

```

```

int maxsum(int n)
{
    consecsum(n);
    int max_sum=-INF;
    for (int i = 0; i<n; i++)
        dp[i] > max_sum ? max_sum = dp[i] : 0;
    return max_sum;
}

```

### 7.2 Knapsack

### 7.3 Longest Common Subsequence

//input : two const char\*



```
//output : their LCS, in c++ std::string type
string lcsf(const char *X,const char *Y)
{
    int m = (int)strlen(X);
    int n = (int)strlen(Y);
    int L[m+1][n+1];
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                L[i][j] = L[i-1][j-1] + 1;
            else
                L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
    }
    int index = L[m][n];

    char lcsstring[index+1];
    lcsstring[index] = 0;

    int i = m, j = n;
    while (i > 0 && j > 0)
    {
        if (X[i-1] == Y[j-1])
        {
            lcsstring[index-1] = X[i-1];
            i--; j--; index--;
        }
        else if (L[i-1][j] > L[i][j-1])
            i--;
        else
            j--;
    }
    string lcsstr = lcsstring;
    return lcsstr;
}
```

## 7.4 Edit Distance

## 8 String

### 8.1 KMP Algorithm

### 8.2 Manacher's Algorithm

### 8.3 Trie

### 8.4 Rabin-Karp Hashing

### 8.5 Aho-Corasick Algorithm

## 9 Miscellaneous

### 9.1 Useful Bitwise Functions in C++

```
int __builtin_clz(int x); // number of leading zero
int __builtin_ctz(int x); // number of trailing zero
int __builtin_clzll(ll x); // number of leading zero
int __builtin_ctzll(ll x); // number of trailing zero
int __builtin_popcount(int x); // number of 1-bits in x
int __builtin_popcountll(ll x); // number of 1-bits in x

lsb(n): (n & -n); // last bit (smallest)
floor(log2(n)): 31 - __builtin_clz(n | 1);
floor(log2(n)): 63 - __builtin_clzll(n | 1);

// compute next perm. ex) 00111, 01011, 01101, 01110, 10011, 10101..
ll next_perm(ll v)
{
    ll t = v | (v-1);
    return (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(v) + 1));
}
```

### 9.2 List of Useful Numbers

< 10 <sup>k</sup>	prime	# of prime	< 10 <sup>k</sup>	prime
1	7	4	10	9999999967
2	97	25	11	9999999977
3	997	168	12	99999999989
4	9973	1229	13	99999999971
5	99991	9592	14	999999999973
6	999983	78498	15	9999999999989
7	9999991	664579	16	99999999999937
8	99999989	5761455	17	999999999999997
9	999999937	50847534	18	9999999999999989

## 10 Debugging Checkpoints