Teambook Sindicato de Transporte 2880

Universidad Mayor de San Simón

March 14, 2023

Contents

Chapter 1

Introduction

The following document represents the Teambook for the team Sindicato de Transporte 2880. This version was elaborated for the Latin American Regional phase of 2022's ICPC.

The template for the C++ code is presented:

```
#include <bits/stdc++.h>
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/tree_policy.hpp>
// #include <ext/rope>
#define int ll
#define mp        make_pair
#define pb        push_back
#define all(a)      (a).begin(), (a).end()
#define sz(a)      (int)a.size()
#define eq(a, b)       (fabs(a - b) < EPS)
#define md(a, b)    ((a) % b + b) % b
#define mod(a)      md(a, MOD)
#define _max(a, b) ((a) > (b) ? (a) : (b))
#define srt(a)      sort(all(a))
#define mem(a, h)    memset(a, (h), sizeof(a))
#define f         first
#define s         second
#define forn(i, n)      for(int i = 0; i < n; i++)
#define fore(i, b, e)  for(int i = b; i < e; i++)
#define forg(i, b, e, m)  for(int i = b; i < e; i+=m)
#define index  int mid = (b + e) / 2, l = node * 2 + 1, r = l + 1;
#define DBG(x) cerr<<#x<<" = "<<(x)<<endl
#define RAYA cout<<"=============================="<<'\n'
// int in(){int r=0,c;for(c=getchar();c<=32;c=getchar());if(c=='-')
     return -in();for(;c>32;r=(r<<1)+(r<<3)+c-'0',c=getchar());
    return r;}


using namespace std;
// using namespace __gnu_pbds;
// using namespace __gnu_cxx;

// #pragma GCC target ("avx2")
// #pragma GCC optimization ("O3")
// #pragma GCC optimization ("unroll-loops")

typedef long long     ll;
typedef long double ld;
typedef unsigned long long     ull;
typedef pair<int, int>  ii;
typedef pair<pair<int, int>, int> iii;
typedef vector<int>     vi;
typedef vector<ii>      vii;
typedef vector<ll>      vll;
// typedef tree<int,null_type,less<int>,rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;
// find_by_order kth largest  order_of_key <
// mt19937 rng(chrono::steady_clock::now().time_since_epoch().count
    ());
// rng
const int tam = 200010;
const int MOD = 1000000007;
```

```cpp
const int MOD1 = 998244353;
const double DINF=1e100;
const double EPS = 1e-9;
const long double PI  = acos(-1.0L);



signed main()
{
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    // freopen("asd.txt", "r", stdin);
    // freopen("qwe.txt", "w", stdout);


    return 0;
}
```

../Template.cpp

In order to run the code from terminal, the following command is used:

```
g++ name.cpp -o run && ./run
```

Chapter 2

Data Structures

2.1 Union Find Disjoint Sets

The Union Find Disjoint Sets data structure is used to keep track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It supports two operations:

- **Find**: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
- **Union**: Join two subsets into a single subset.

The implementation is as follows:

```
ll dsu[tam];

int getParent(int x){
    if(dsu[x]<0) return x;
    else return dsu[x] = getParent(dsu[x]);
}

void join(int x, int y){
    x = getParent(x);
    y = getParent(y);
    if(x==y) return;
    if(dsu[x]>dsu[y]) swap(x, y);
    dsu[x] = y;

}
```

```
//With structs
struct unionFind {
  vi p;
  unionFind(int n) : p(n, -1) {}
  int findParent(int v) {
    if (p[v] == -1) return v;
    return p[v] = findParent(p[v]);
  }
  bool join(int a, int b) {
    a = findParent(a);
    b = findParent(b);
    if (a == b) return false;
    p[a] = b;
    return true;
  }
};
```

../Data Structures/UnionFind.cpp

2.2 Binary Indexed (Fenwick) Tree

The Binary Indexed Tree (BIT) is a data structure that can efficiently update elements and calculate prefix sums in a table of numbers. It is also called a Fenwick Tree, as it was first described by Peter Fenwick. The implementation is as follows:

```
#define clr(a,h)    memset(a,(h),sizeof(a))
```

```
int BIT[tam];

void update(int pos, int val)
{
  pos++;
  while(pos < 200010)
  {
    BIT[pos] += val;
    pos += (pos & -pos);
  }
}

int query(int pos)
{
  pos++;
  int res = 0;
  while(pos > 0)
  {
    res += BIT[pos];
    pos -= (pos & -pos);
  }
  return res;
}

int main()
{
  clr(BIT,0);
  for(int i = n - 1; i >=0; i--)
  {
    inv +=query(a[i]);
    update(a[i],1);
  }
}

//to update all the values in a range [i,j] the following
    implementation is used

range_update(i,j,val) = update(i,val); update(j+1,-val);

//to find the range result of a range [i,j] for a Range Update
    Range Query

//rsq(1,j) = rupq.point_query(j)*j-purq.rsq(j)
```

../Data Structures/FenwickTree.cpp

This Data Structure can also be used to find the maximum value in a range of values in an array.

It can also be extended to 2D, 3D, etc. The implementation is as follows:

```
#include <iostream>

using namespace std;

const int tam = 1000;

int BIT[tam][tam];
int n, m;

void update(int row, int col, int val)
{
   row++; col++;
   for (int i = row; i <= n; i += (i & -i))
   {
      for (int j = col; j <= m; j += (j & -j))
      {
         BIT[i][j] += val;
      }
   }
}

int query(int row, int col)
{
   int res = 0;
   row++; col++;
   for (int i = row; i > 0; i -= (i & -i))
   {
      for (int j = col; j > 0; j -= (j & -j))
      {
         res += BIT[i][j];
      }
   }
   return res;
```

```
}
```

../Data Structures/FenwickTree2D.cpp

2.3 Segment Tree

The Segment Tree is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array. It is, in principle, a static structure. It can answer most queries in O(log n), but its true power is answering range updates. For that, it takes O(log n) time per update.

The implementation is as follows:

```
struct node
{
  int val;
};
node join(node a, node b)
{
  a.val += b.val;
  return a;
}
int ar[tam];
node t[4 * tam];
void init(int b, int e, int node)
{
   if(b == e)
   {
      t[node].val = ar[b];
      return;
   }
   int mid = (b + e) / 2, l = node * 2 + 1, r = l + 1;
   init(b, mid, l);
   init(mid + 1, e, r);
   t[node] = join(t[l], t[r]);
}
//b, e always the beginning and end of the segment tree;

node query(int b, int e, int node, int i, int j)
{
   if(b >= i && e <= j)
      return t[node];
   int mid = (b + e) / 2, l = node * 2 + 1, r = l + 1;
   if(mid < i)
      return query(mid + 1, e, r, i, j);
   if(mid >= j)
      return query(b, mid, l, i, j);
   return join(query(b, mid, l, i, j), query(mid + 1, e, r, i, j));
}
void update(int b, int e, int node, int pos, int val)
{
   if(b == e) {t[node].val = val;return;} // Replaces the value,
    could be updated to add val to the current value
   int mid = (b + e) / 2, l = node * 2 + 1, r = l + 1;
   if(mid < pos)
      update(mid + 1, e, r, pos, val);
   else
      update(b, mid, l, pos, val);
   t[node] = join(t[l], t[r]);
}
```

../Data Structures/SegmentTree.cpp

An iterative implementation is also possible:

```
const int N = 1e5;  // limit for array size
int n;  // array size
int t[2 * N];

void build() {  // build the tree
  for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void modify(int p, int value) {  // set value at position p
  for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
}

int query(int l, int r) {  // sum on interval [l, r)
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) res += t[l++];
    if (r&1) res += t[--r];
  }
  return res;
```

```
}
```

../Data Structures/SegmentTreeIterative.cpp

Another useful tool while using the Segment Tree is the Lazy Propagation technique. This technique allows us to perform range updates in O(log n) time. The implementation is as follows:

```
struct segtree {
    int size;

    int NO_OPERATION = LLONG_MAX; // for the case of assignements,
    the neutral element changes according to the type of operation
    that is performed

    vi operations;
    int operation(int a, int b){
        if ( b == NO_OPERATION) //to assign a value to a range, we
    need to return the value that we want to assign
            return a;
        return b;
    }

    void apply(int &x, int v) {
        x = operation(x, v);
    }

    void init(int n) {
        size = 1;
        while (size < n) size *= 2;
        operations.assign(2 * size, 0LL);
    }

    void propagate(int x, int lx, int rx){
        if(rx - lx == 1) return;
        apply(operations[2 * x + 1], operations[x]);
        apply(operations[2 * x + 2], operations[x]);
        operations[x] = NO_OPERATION;
    }

    void modify(int l, int r, int v, int x, int lx, int rx){
        propagate(x, lx, rx);
        if(lx >= r || l >= rx) return;
        if(lx >= l && rx <= r) {
            apply(operations[x], v);
            return;
        }
        int m = (lx + rx) / 2;
        modify(l, r, v, 2 * x + 1, lx, m);
        modify(l, r, v, 2 * x + 2, m, rx);
    }

    void modify(int l, int r, int v) {
        modify(l, r, v, 0, 0, size);
    }

    int get(int i, int x, int lx, int rx){
        propagate(x, lx, rx);
        if(rx - lx == 1) return operations[x];
        int m = (lx + rx) / 2;
        int res;
        if(i < m) res = get(i, 2 * x + 1, lx, m);
        else res = get(i, 2 * x + 2, m, rx);
        return operation(res,operations[x]);
    }

    int get(int i) {
        return get(i, 0, 0, size);
    }

};




signed main()
{
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    // freopen("asd.txt", "r", stdin);
    // freopen("qwe.txt", "w", stdout);

    int n;
```

```
    cin >> n;
    int q;
    cin >> q;

    segtree st;
    st.init(n);

    while(q--){
        int c;
        cin>>c;
        if(c==1){
            int l,r, v;
            cin>>l>>r>>v;
            st.modify(l,r,v);
            continue;
        }

        int i;
        cin >> i;
        cout << st.get(i) << '\n';


    }


    return 0;
}
```

../Data Structures/LazyPropagation.cpp

```
//replace the values and add in the range sum query
struct segtree {
    int size;
    vi operations;
    vi values;

    int NEUTRAL_ELEMENT = 0;
    int NO_OPERATION = LLONG_MAX - 1;

    int modify_op(int a, int b, int len) {
        if(b == NO_OPERATION) return a;
        return b * len;
    }
    int calc_op(int a, int b) {
        return a + b;
    }

    void apply_op(int &a, int b, int len) {
        a = modify_op(a, b, len);
    }

    // void build(int x, int lx, int rx) {
    //     if (rx - lx == 1) {
    //         values[x] = 1;
    //         return;
    //     }
    //     int m = (lx + rx) / 2;
    //     build(2 * x + 1, lx, m);
    //     build(2 * x + 2, m, rx);
    //     values[x] = calc_op(values[2 * x + 1], values[2 * x +
2]);
    // }

    void init(int n) {
        size = 1;
        while (size < n) size *= 2;
        operations.assign(2 * size, 0LL);
        values.assign(2 * size, 0LL);
        // build(0, 0, size);
    }

    void propagate(int x, int lx, int rx){
        if(rx - lx == 1) return;
        int m = (lx + rx) / 2;
        apply_op(operations[2 * x + 1], operations[x], 1);
        apply_op(operations[2 * x + 2], operations[x], 1);
        apply_op(values[2 * x + 1], operations[x], m-lx);
        apply_op(values[2 * x + 2], operations[x], rx-m);
        operations[x] = NO_OPERATION;
    }

    //From here the code is the same
```

```
    void modify(int l, int r, int v, int x, int lx, int rx){
        propagate(x, lx, rx);
        if(lx >= r || l >= rx) return;
        if(lx >= l && rx <= r) {
            apply_op(operations[x], v, 1);
            apply_op(values[x], v, rx - lx);
            return;
        }
        int m = (lx + rx) / 2;
        modify(l, r, v, 2 * x + 1, lx, m);
        modify(l, r, v, 2 * x + 2, m, rx);
        values[x] = calc_op(values[2 * x + 1], values[2 * x + 2]);
    }

    int calc(int l, int r, int x, int lx, int rx){
        propagate(x, lx, rx);
        if(lx >= r || l >= rx) return NEUTRAL_ELEMENT;
        if(lx >= l && rx <= r) return values[x];
        int m = (lx + rx) / 2;
        auto s1 = calc(l, r, 2 * x + 1, lx, m);
        auto s2 = calc(l, r, 2 * x + 2, m, rx);
        return calc_op(s1, s2);
    }

    int calc(int l, int r){
        return calc(l, r, 0, 0, size);
    }

    void modify(int l, int r, int v) {
        modify(l, r, v, 0, 0, size);
    }

    int get(int i, int x, int lx, int rx){
        if(rx - lx == 1) return operations[x];
        int m = (lx + rx) / 2;
        int res;
        if(i < m) res = get(i, 2 * x + 1, lx, m);
        else res = get(i, 2 * x + 2, m, rx);
        return res + operations[x];
    }
    int get(int i) {
        return get(i, 0, 0, size);
    }

};




signed main()
{
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    // freopen("asd.txt", "r", stdin);
    // freopen("qwe.txt", "w", stdout);

    int n;
    cin >> n;
    int q;
    cin >> q;

    segtree st;
    st.init(n);

    while(q--){
        int c;
        cin>>c;
        if(c==1){
            int l,r, v;
            cin>>l>>r>>v;
            st.modify(l,r,v);
            continue;
        }

        int l,r;
        cin >> l >> r;
        cout << st.calc(l,r) << '\n';
    }


    return 0;
```

```
}
```

../Data Structures/LazyPropagation2.cpp

In order to do operations on segments without having to worry about the lazy propagation, we can use the following functions:

```
struct segtree {
    int size;
    vi operations;
    void init(int n) {
        size = 1;
        while (size < n) size *= 2;
        operations.assign(2 * size, 0LL);
    }

    void add(int l, int r, int v, int x, int lx, int rx){
        if(lx >= r || l >= rx) return;
        if(lx >= l && rx <= r) {
            operations[x]+=v;
            return;
        }
        int m = (lx + rx) / 2;
        add(l, r, v, 2 * x + 1, lx, m);
        add(l, r, v, 2 * x + 2, m, rx);
    }

    void add(int l, int r, int v) {
        add(l, r, v, 0, 0, size);
    }

    int get(int i, int x, int lx, int rx){
        if(rx - lx == 1) return operations[x];
        int m = (lx + rx) / 2;
        int res;
        if(i < m) res = get(i, 2 * x + 1, lx, m);
        else res = get(i, 2 * x + 2, m, rx);
        return res + operations[x];
    }

    int get(int i) {
        return get(i, 0, 0, size);
    }
};




signed main()
{
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    // freopen("asd.txt", "r", stdin);
    // freopen("qwe.txt", "w", stdout);

    int n;
    cin >> n;
    int q;
    cin >> q;

    segtree st;
    st.init(n);

    while(q--){
        int c;
        cin>>c;
        if(c==1){
            int l,r, v;
            cin>>l>>r>>v;
            st.add(l,r,v);
            continue;
        }

        int i;
        cin >> i;
        cout << st.get(i) << '\n';


    }


    return 0;
}
```

../Data Structures/Operations.cpp

This second implementation only works for operations that have commutative and associative properties. Emphasis on the commutativity of the operation.

2.4 Queue With Minimum

The Queue With Minimum data structure is a queue that supports the following operations:

- **push**: Add an element to the back of the queue.
- **pop**: Remove an element from the front of the queue.
- **min**: Return the minimum element in the queue.

The implementation is as follows:

```
struct quemin
{
   stack<pair<int,int>> bo, to;
   void push(int n)
   {
      if(bo.empty())
         bo.push(mp(n, n));
      else
         bo.push(mp(n, min(bo.top().s, n)));
   }
   void pop()
   {
      if(to.empty())
      {
         while(!bo.empty())
         {
            if(to.empty())
               to.push(mp(bo.top().f, bo.top().f));
            else
               to.push(mp(bo.top().f, min(bo.top().f, to.top().s)))
 ;
            bo.pop();
         }
      }
      to.pop();
   }
   int mini()
   {
      int mini = MOD;
      if(!bo.empty())
         mini = bo.top().s;
      if(!to.empty())
         mini = min(mini, to.top().s);
      return mini;
   }
};

struct quemin
{
   pair<int,int> bo[100010], to[100010];
   int boto = -1, toto = -1, ax;
   void push(int n)
   {
      ax = boto + 1;
      if(boto == -1)
         bo[ax] = mp(n, n);
      else
         bo[ax] = mp(n, min(bo[boto].s, n));
      boto++;
   }
   void pop()
   {
      if(toto == -1)
      {
         while(boto > -1)
         {
            ax = toto + 1;
            if(toto == -1)
               to[ax] = mp(bo[boto].f, bo[boto].f);
            else
               to[ax] = mp(bo[boto].f, min(bo[boto].f, to[toto].s))
 ;
            toto++;
            boto--;
         }
```

```
      }
      if(toto > -1)
         toto--;
   }
   int mini()
   {
      int mini = MOD;
      if(boto > -1)
         mini = bo[boto].s;
      if(toto > -1)
         mini = min(mini, to[toto].s);
      return mini;
   }
};
```

../Data Structures/QueueMin.cpp

2.5 Sparse Table

The Sparse Table is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array. It is, in principle, a static structure. It can answer most queries in O(1), but its true power is answering range updates. For that, it takes O(log n) time per update.

The implementation is as follows:

```
const int tam = 1000010;
const int logTam = 21;
int n;
int ar[tam], table[logTam][tam];
void inispar()
{
   fore(i, 0, n) table[0][i] = ar[i];
   for(int k = 0; (1 << k) < n; k++)
      for(int i = 0; i + (1 << k) < n; i++)
         table[k + 1][i] = min(table[k][i], table[k][i + (1 << k)])
    ;
}
int query(int b, int e)
{
   int lev = 31 - __builtin_clz(e - b + 1);
   return min(table[lev][b], table[lev][e - (1 << lev) + 1]);
}
```

../Data Structures/SparseTable.cpp

```
template<typename it, typename bin_op>
struct sparse_table {

    using T = typename remove_reference<decltype(*declval<it>())>::
    type;
    vector<vector<T>> t; bin_op f;

    sparse_table(it first, it last, bin_op op) : t(1), f(op) {
        int n = distance(first, last);
        t.assign(32-__builtin_clz(n), vector<T>(n));
        t[0].assign(first, last);
        for (int i = 1; i < t.size(); i++)
            for (int j = 0; j < n-(1<<i)+1; j++)
                t[i][j] = f(t[i-1][j], t[i-1][j+(1<<(i-1))]);
    }

    // returns f(a[l..r]) in O(1) time
    T query(int l, int r) {
        int h = floor(log2(r-l+1));
        return f(t[h][l], t[h][r-(1<<h)+1]);
    }
};


sparse_table g(all(vec), [](ll x, ll y){
    return __gcd(x, y);
});


sparse_table g(ar, ar + n, [](ll x, ll y){
    return __gcd(x, y);
});
```

../Data Structures/SparseTableGen.cpp

2.6 Persistent Segment Tree

The Persistent Segment Tree is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array. It is, in principle, a static structure. It can answer most queries in O(log n), but its true power is answering range updates. For that, it takes O(log n) time per update.

The implementation is as follows:

```
struct node{
    ptr iz;
    ptr der;
    int val;   //0.o
    int numero;
    node(){
        numero=-1;
        val=0;
    }
};
node nodos[tam];int cnodos=0;
node NUL;
ptr getnode()
{
    nodos[cnodos].iz=nodos[cnodos].der=&NUL;
    //if (cnodos>=tam)
         //tle(); no
    return &nodos[cnodos++];
}
void clr(){
    NUL.iz=NUL.der=&NUL;
}

void insertar(ptr nuevo,ptr antnodo,int iz,int der,int pos,int
    numero)
{
    if (iz==der)
    {

        (*nuevo).val=(*antnodo).val+1;
        (*nuevo).numero=numero;
        return;
    }
    int mid=(iz+der)/2;
    if (pos<=mid)
    {
        (*nuevo).der=(*antnodo).der;
        (*nuevo).iz=getnode();
        insertar((*nuevo).iz,(*antnodo).iz,iz,mid,pos,numero);
    }
    else
    {
        (*nuevo).iz=(*antnodo).iz;
        (*nuevo).der=getnode();
        insertar((*nuevo).der,(*antnodo).der,mid+1,der,pos,numero);
    }
    (*nuevo).val=(*(*nuevo).iz).val+(*(*nuevo).der).val;
}

int query(ptr nodoa,ptr nodob,ptr resta1,ptr resta2,int kth,int iz,
    int der)
{
    if (iz==der)
    {
   return iz;// numero
    }
    int valiz=(*(*nodoa).iz).val+(*(*nodob).iz).val-(*(*resta1).iz)
    .val-(*(*resta2).iz).val;
    int mid=(iz+der)/2;
    if (kth>valiz)
    {
        query((*nodoa).der,(*nodob).der,(*resta1).der,(*resta2).der
    ,kth-valiz,mid+1,der);//kth-valiz ***
    }
    else
    {
        query((*nodoa).iz,(*nodob).iz,(*resta1).iz,(*resta2).iz,kth
    ,iz,mid);
    }
}
```

../Data Structures/PersistentSegmentTree.cpp

Chapter 3

Mathematics

This chapter is about some useful mathematical tools needed in order to solve problems.

3.1 GCD and LCM

In order to find the greatest common divisor (GCD) of two numbers, the Euclidean algorithm can be used. The implementation is as follows:

```

ll gcd(ll a, ll b){return b==0? a:gcd(b,a%b);}

int x, y, d;
void extendedEuclid(int a, int b)//ecuacion diofantica ax + by = d
{
   if(b==0) {x=1; y=0; d=a; return;}
   extendedEuclid(b,a%b);
   int x1=y;
   y = x-(a/b)*y;
   x=x1;
}
```

../Mathematics/Euclid.cpp

Another (and faster) way to find the GCD is by using the following code:

```
int gcd(int a, int b) {
    if (!a || !b)
        return a | b;
    unsigned shift = __builtin_ctz(a | b);
    a >>= __builtin_ctz(a);
    do {
        b >>= __builtin_ctz(b);
        if (a > b)
            swap(a, b);
        b -= a;
    } while (b);
    return a << shift;
}
```

../Mathematics/FastGCD.cpp

The way Halim suggests to find the GCD and the LCM is given by the following code:

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b) { return a / gcd(a, b) * b; }
```

../Mathematics/HalimGCD.cpp

3.2 Prime Numbers

The fastest way to check the primality of a number is by using Erathostenes' sieve. The typical implementation is as follows:

```
bitset<100000> bi;
vi primos;  //primos
vector<ll> pric;  //primos al cuadrado
void criba()
{
   bi.set();
```

```
   for(int i=2;i<100000;i++)
      if(bi[i])
      {
         for(int j=i+i;j<100000;j+=i)
            bi[j]=0;
         primos.push_back(i);
         pric.push_back((ll)i*(ll)i);
      }
}
int euler(int n)
{
   int res=n;
   for(int i=0;pric[i]<=n;i++)
   {
      if(n%primos[i]==0)
      {
         res-= res/primos[i];
         while(n%primos[i]==0) n/=primos[i];
      }
   }
   if(n!=1) res-=res/n;
   return res;
}
```

../Mathematics/Erathostenes.cpp

Nevertheless, the following implementation is faster, since the statement if

```
if (i % prime[j] == 0) break;
```

terminates the loop when p divides i. The inner loop is executed only once for each composite. Hence, the code performs in O(n) complexity, resulting in the 'linear' sieve:

```
// This algorithm allows to find Eratosthenes sieve in O(n logn)
    time.

std::vector <int> prime;
bool is_composite[MAXN];

void sieve (int n) {
   std::fill (is_composite, is_composite + n, false);
   for (int i = 2; i < n; ++i) {
      if (!is_composite[i]) prime.push_back (i);
      for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
         is_composite[i * prime[j]] = true;
         if (i % prime[j] == 0) break;
      }
   }
}

// An application of this lineatr sieve is to find the Euler
    totient function of a number in O(n logn) time.
std::vector <int> prime;
bool is_composite[MAXN];
int phi[MAXN];

void sieve (int n) {
   std::fill (is_composite, is_composite + n, false);
   phi[1] = 1;
   for (int i = 2; i < n; ++i) {
      if (!is_composite[i]) {
         prime.push_back (i);
         phi[i] = i - 1;               //i is prime
      }
      for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
         is_composite[i * prime[j]] = true;
         if (i % prime[j] == 0) {
            phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
    divides i
            break;
         } else {
            phi[i * prime[j]] = phi[i] * phi[prime[j]];  //prime[j]
     does not divide i
         }
      }
   }
}
```

../Mathematics/LinearSieve.cpp

3.3 Modular Arithmetic

The modular inverse is defined by the following equation:

$$a \cdot a^{-1} \equiv 1 \mod m \tag{3.1}$$

The following code shows how to find the modular inverse of a number:

```
int ModPow(int a, int b, int m) {
  int res = 1;
  while (b > 0) {
    if (b & 1) res = (res * a) % m;
    a = (a * a) % m;
    b >>= 1;
  }
  return res;
}

// Language: java
public static int modPow(int a, int b, int m) {
  int res = 1;
  while (b > 0) {
    if ((b & 1) == 1) res = (res * a) % m;
    a = (a * a) % m;
    b >>= 1;
  }
  return res;
}

int ModInverse(int a, int m) {
  return ModPow(a, m - 2, m);
}
```

../Mathematics/ModularInverse.cpp

Some other useful relationships in modular arithmetic are:

- $(a+b) \mod m = (a \mod m + b \mod m) \mod m$
- $(a-b) \mod m = (a \mod m - b \mod m) \mod m$
- $(a*b) \mod m = (a \mod m * b \mod m) \mod m$
- $(a/b) \mod m = (a \mod m * b^{-1} \mod m) \mod m$
- $(a^b) \mod m = (a \mod m)^b \mod m$
- $(a^b) \mod m = (a \mod m)^{b \mod \phi(m)} \mod m$
- $(a^b) \mod m = (a \mod m)^{b \mod (m-1)} \mod m$
- $\frac{a}{k} \equiv \frac{a}{k} \mod m \iff a \equiv k \mod m$
- $\frac{a}{k} \equiv \frac{a}{k} \left(\mod \frac{n}{\gcd(n,k)} \right)$

3.4 Matrix Exponentiation

The following code shows how to find the nth power of a *mat*, noting that a data structure of type matrix is defined as follows:

```
typedef vector<vector<ll>> mat;
mat ans;
void mult(mat m1, mat m2)
{
   assert(m1[0].size() == m2.size());
   ans.clear();
   ll answer = 0;
   fore(i, 0, m1.size())
   {
      vector<ll> fila;
      fore(j, 0, m2[0].size())
      {
         answer = 0;
         fore(k, 0, m2.size())
            answer = (answer + m1[i][k] * m2[k][j]) % MOD;
         fila.pb(answer);
      }
      ans.pb(fila);
   }
}
void pot(mat base, ll exp)
{
   mat res(base.size(), vector<ll>(base.size(), 0));
   fore(i, 0, base.size())
   res[i][i] = 1;
   while(exp)
   {
      if(exp & 1)
      {
         mult(res, base);
```

```
            res = ans;
        }
        mult(base, base);
        base = ans;
        exp /= 2;
    }
    ans = res;
}
```

../Mathematics/MatrixPower.cpp

3.5 Gauss Jordan Elimination

The following code shows how to solve a system of linear equations using Gauss Jordan elimination:

```
// resuelve Ax = b, dada la matriz a de n * (m + 1), n ecuaciones y
     m variables, siendo la ultima columna el vector b
// The function returns the number of solutions of the system (0,1,
    or INF). if there's at least a solution, it's in ans
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be infinity or a
    big number
int gauss (vector < vector<double> > a, vector<double> & ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

../Mathematics/GaussJordan.cpp

3.6 Determinant

The following code shows how to find the determinant of a matrix:

```
#include <iostream>

using std::cin;
using std::cout;
using std::endl;

int **submatrix(int **matrix, unsigned int n, unsigned int x,
    unsigned int y) {
    int **submatrix = new int *[n - 1];
    int subi = 0;
```

```
    for (int i = 0; i < n; i++) {
        submatrix[subi] = new int[n - 1];
        int subj = 0;
        if (i == y) {
            continue;
        }
        for (int j = 0; j < n; j++) {
            if (j == x) {
                continue;
            }
            submatrix[subi][subj] = matrix[i][j];
            subj++;
        }
        subi++;
    }
    return submatrix;
}

int determinant(int **matrix, unsigned int n) {
    int det = 0;
    if (n == 2) {
        return matrix[0][0] * matrix[1][1] - matrix[1][0] * matrix
    [0][1];
    }
    for (int x = 0; x < n; ++x) {
        det += ((x % 2 == 0 ? 1 : -1) * matrix[0][x] * determinant(
    submatrix(matrix, n, x, 0), n - 1));
    }

    return det;
}

int main() {
    int n;
    cin >> n;
    int **matrix = new int *[n];
    for (int i = 0; i < n; ++i) {
        matrix[i] = new int[n];
        for (int j = 0; j < n; ++j) {
            cin >> matrix[i][j];
        }
    }

    cout << determinant(matrix, n);

    return 0;
}
```

../Mathematics/Determinant.cpp

3.7 Numerical Integration

The following code shows how to find the integral of a function f(x) in the interval [a,b] using Simpson's rule:

```
double simpson(double f(double), double a, double b)
{
   int n = 100000;
   double s = f(a) + f(b);
   double h = (b - a) / n;
   fore(i, 1, n)
      s += ((i & 1) ? 4  : 2) * f(a + h * i);
   return s * (h / 3);
}
```

../Mathematics/Simpson.cpp

Chapter 4

Graphs

This chapter shows some of the basec algorithms and implementations required to solve problems that include graphs.

4.1 Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that visits all the nodes of a graph. It is used to find connected components, topological sorting, and to find bridges and articulation points. The algorithm is as follows:

The implementation can be done as follows:

```
vector<vector<int>> g(tam);
vector<bool> vis(tam);

void dfs(int u){
    vis[u]=true;
    ans++;
    for(int v: g[u]){
        if(!vis[v]){
            dfs(v);
        }
    }
}

signed main()
{
    int n,m;
    cin>>n>>m; // n nodes, m edges
    g.assign(tam,vector<int>());
    vis.assign(tam, false);
    for(int i=0; i<m;i++){
        int u,v;
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    ll res = 0;
    for(int i=1; i<=n;i++){
        if(!vis[i]){
            ans=0;
            dfs(i);
            res = max(res,ans);
        }
    }
    g.clear();
    vis.clear();
    return 0;
}
```

../Graphs/DFS.cpp

An application of this algorithm in order to find the shorteast path between two nodes can be done as follows:

```
// The following code represents the implementation of a DFS
    algorithm
// to find the shortest path between two nodes in a graph.
// The graph is represented as an adjacency list.
// The algorithm is implemented using a stack.
```

Algorithm 1 Depth First Search (DFS)

procedure DFS(G)
  $visited \leftarrow \emptyset$
  $time \leftarrow 0$
  $parent \leftarrow \emptyset$
  $low \leftarrow \emptyset$
  $disc \leftarrow \emptyset$
  $AP \leftarrow \emptyset$
  $bridge \leftarrow \emptyset$
  for all $v \in V$ **do**
    $visited[v] \leftarrow false$
    $parent[v] \leftarrow -1$
    $low[v] \leftarrow \infty$
    $disc[v] \leftarrow \infty$
  end for
  for all $v \in V$ **do**
    if $visited[v] = false$ **then**
      DFSUtil(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
    end if
  end for
end procedure
procedure DFSUtil(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
  $visited[v] \leftarrow true$
  $disc[v] \leftarrow time$
  $low[v] \leftarrow time$
  $time \leftarrow time + 1$
  $children \leftarrow 0$
  for all $u \in Adj(v)$ **do**
    if $visited[u] = false$ **then**
      $parent[u] \leftarrow v$
      $children \leftarrow children + 1$
      DFSUtil(G, u, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
      $low[v] \leftarrow min(low[v], low[u])$
      if $parent[v] = -1$ and $children > 1$ **then**
        $AP[v] \leftarrow true$
      end if
      if $parent[v]! = -1$ and $low[u] \geq disc[v]$ **then**
        $AP[v] \leftarrow true$
      end if
      if $low[u] > disc[v]$ **then**
        $bridge[v][u] \leftarrow true$
      end if
    else
      $low[v] \leftarrow min(low[v], disc[u])$
    end if
  end for

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

using namespace std;


vector<int> DFS(vector<vector<int>> &adj, int s, int t) {
  stack<vector<int>> path_stack;
  vector<int> path;
  vector<int> visited(adj.size(), 0);
  path_stack.push({s});
  while (!path_stack.empty()) {
    path = path_stack.top();
    path_stack.pop();
    int last = path[path.size() - 1];
    if (last == t) {
      return path;
    }
    if (visited[last] == 0) {
      visited[last] = 1;
      for (int i = 0; i < adj[last].size(); i++) {
        if (visited[adj[last][i]] == 0) {
          vector<int> new_path(path);
          new_path.push_back(adj[last][i]);
          path_stack.push(new_path);
        }
      }
    }
  }
  return {};
}

int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<int>> adj(n, vector<int>());
  for (int i = 0; i < m; i++) {
    int x, y;
    cin >> x >> y;
    adj[x - 1].push_back(y - 1);
    adj[y - 1].push_back(x - 1);
```

```cpp
  }
  int x, y;
  cin >> x >> y;
  x--, y--;
  vector<int> path = DFS(adj, x, y);
  for (int i = 0; i < path.size(); i++) {
    cout << path[i] + 1 << " ";
  }
}
```

../Graphs/DFS–application.cpp

4.2 Breadth First Search (BFS)

The BFS algorithm is a non-recursive algorithm that visits all the nodes of a graph. It is used to find connected components, topological sorting, and to find bridges and articulation points, to better understand it, a propagating fire can be imagined. The algorithm is as follows:

The implementation can be done as follows:

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

using namespace std;
signed main()
{
    vector<vector<int>> adj;  // adjacency list representation
    int n; // number of nodes
    int s; // source vertex

    queue<int> q;
    vector<bool> used(n);
    vector<int> d(n), p(n);

    q.push(s);
    used[s] = true;
    p[s] = -1;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int u : adj[v]) {
            if (!used[u]) {
```

Algorithm 2 Breadth First Search (BFS)

procedure BFS(G)
$visited \leftarrow \emptyset$
$time \leftarrow 0$
$parent \leftarrow \emptyset$
$low \leftarrow \emptyset$
$disc \leftarrow \emptyset$
$AP \leftarrow \emptyset$
$bridge \leftarrow \emptyset$
for all $v \in V$ **do**
$visited[v] \leftarrow false$
$parent[v] \leftarrow -1$
$low[v] \leftarrow \infty$
$disc[v] \leftarrow \infty$
end for
for all $v \in V$ **do**
if $visited[v] = false$ **then**
BFSUtil(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
end if
end for
end procedure
procedure BFSUtil(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
$visited[v] \leftarrow true$
$disc[v] \leftarrow time$
$low[v] \leftarrow time$
$time \leftarrow time + 1$
$children \leftarrow 0$
for all $u \in Adj(v)$ **do**
if $visited[u] = false$ **then**
$parent[u] \leftarrow v$
$children \leftarrow children + 1$
BFSUtil(G, u, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
$low[v] \leftarrow min(low[v], low[u])$
if $parent[v] = -1$ and $children > 1$ **then**
$AP[v] \leftarrow true$
end if
if $parent[v]! = -1$ and $low[u] \geq disc[v]$ **then**
$AP[v] \leftarrow true$
end if
if $low[u] > disc[v]$ **then**
$bridge[v][u] \leftarrow true$
end if
else
$low[v] \leftarrow min(low[v], disc[u])$
end if
end for

```cpp
                used[u] = true;
                q.push(u);
                d[u] = d[v] + 1;
                p[u] = v;
            }
        }
    }
    return 0;
}
```

../Graphs/BFS.cpp

4.3 Finding Bridges and Articulation Points

The following algorithms are used to find bridges and articulation points in a graph. The implementation of these algorithms is done using DFS and BFS. This algoithms are based on Tarjan's algorithm.

Tarjan's algorithm is an algorithm that is used to find bridges and articulation points in a graph. The algorithm is as follows:

```cpp
int n;
vector<vector<int>> adj;

vector<bool> visited;
vector<int> tin, low;
int timer;
vector<vectotr<int>> comps; // componentes biconexos
stack<int> stk;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    stk.push(v);
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
```

Algorithm 3 Tarjan's Algorithm

```
procedure Tarjan(G)
    visited ← ∅
    time ← 0
    parent ← ∅
    low ← ∅
    disc ← ∅
    AP ← ∅
    bridge ← ∅
    for all v ∈ V do
        visited[v] ← false
        parent[v] ← −1
        low[v] ← ∞
        disc[v] ← ∞
    end for
    for all v ∈ V do
        if visited[v] = false then
            TarjanUtil(G, v, visited, time, parent, low, disc, AP, bridge)
        end if
    end for
end procedure
procedure TarjanUtil(G, v, visited, time, parent, low, disc, AP, bridge)
    visited[v] ← true
    disc[v] ← time
    low[v] ← time
    time ← time + 1
    children ← 0
    for all u ∈ Adj(v) do
        if visited[u] = false then
            parent[u] ← v
            children ← children + 1
            TarjanUtil(G, u, visited, time, parent, low, disc, AP, bridge)
            low[v] ← min(low[v], low[u])
            if parent[v] = −1 and children > 1 then
                AP[v] ← true
            end if
            if parent[v]! = −1 and low[u] ≥ disc[v] then
                AP[v] ← true
            end if
            if low[u] > disc[v] then
                bridge[v][u] ← true
            end if
        else
            low[v] ← min(low[v], disc[u])
        end if
    end for
```

```
            if (low[to] >= tin[v])
            {
               if (p != -1) IS_CUTPOINT(v);

               comps.push_back({v});
               while (comps.back().back() != to)
               {
                  comps.back().push_back(stk.top());
                  stk.pop();
               }

            }
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}

vector<int> id;
int curBCTNode;
vector<vector<int> > tree;
vector<bool> isAP;

void buildTree() {
    curBCTNode = 0;
    id.assign(n, -1);
    tree.clear();
    isAP.clear();
    fore(v, 0, n) {
        if (cutpoint[v]) {
            id[v] = tree.size();
            tree.pb({});
            isAP.pb(true);
        }
    }
    for (auto comp : comps) {
        int v = tree.size();
        tree.pb({});
        isAP.pb(false);
        for (int x : comp) {
            if (cutpoint[x]) {
                tree[v].pb(id[x]);
                tree[id[x]].pb(v);
            }
            else {
                id[x] = v;
            }
        }
    }
}
```

../Graphs/Tarjan_y_BlockCutTree.cpp

4.3.1 Bridges

A bridge is an edge that if it is removed, the graph will be divided into two or more components. The implementation is:

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
```

```
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

../Graphs/FindBridges.cpp

4.3.2 Articulation Points

An articulation point is a node that if it is removed, the graph will be divided into two or more components. The implementation is:

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

../Graphs/FindArticulationPoints.cpp

4.4 Flows

The flow is a concept that is used in many algorithms, it is used to find the maximum flow that could go through a system of nodes.

4.4.1 Dinic

The Dinic algorithm is a useful algorithm to find the maximum flow that could go through a system of nodes. The implementation of this algorithm is:

```
struct flowEdge
{
    int to, rev, f, cap;
```

```
};

vector<vector<flowEdge> > G;


void addEdge(int st, int en, int cap) {
    // Anade arista (st --> en) con su capacidad
    flowEdge A = {en, (int)G[en].size(), 0, cap};
    flowEdge B = {st, (int)G[st].size(), 0, 0};
    G[st].pb(A);
    G[en].pb(B);
}

int nodes, S, T; // asignar estos valores al armar el grafo G
                 // nodes = nodos en red de flujo. Hacer G.clear();
     G.resize(nodes);
vi work, lvl;
int Q[200010];

bool bfs() {
    int qt = 0;
    Q[qt++] = S;
    lvl.assign(nodes, -1);
    lvl[S] = 0;
    for (int qh = 0; qh < qt; qh++) {
        int v = Q[qh];
        for (flowEdge &e : G[v]) {
            int u = e.to;
            if (e.cap <= e.f || lvl[u] != -1) continue;
            lvl[u] = lvl[v] + 1;
            Q[qt++] = u;
        }
    }
    return lvl[T] != -1;
}

int dfs(int v, int f) {
    if (v == T || f == 0) return f;
    for (int &i = work[v]; i < G[v].size(); i++) {
        flowEdge &e = G[v][i];
        int u = e.to;
        if (e.cap <= e.f || lvl[u] != lvl[v] + 1) continue;
        int df = dfs(u, min(f, e.cap - e.f));
        if (df) {
            e.f += df;
            G[u][e.rev].f -= df;
            return df;
        }
    }
    return 0;
}

int maxFlow() {
    int flow = 0;
    while (bfs()) {
        work.assign(nodes, 0);
        while (true) {
            int df = dfs(S, INF);
            if (df == 0) break;
            flow += df;
        }
    }
    return flow;
}
```

../Graphs/Dinic.cpp

This implementation is done in order to do the Dinic algorithm for a graph with a large number of nodes.

This algorithm is based on the idea of the BFS algorithm, it is used to find the shortest path between two nodes, in this case, the shortest path between the source and the sink. The algorithm is as follows:

4.4.2 Ford Fulkerson

The Ford Fulkerson algorithm is a useful algorithm to find the maximum flow that could go through a system of nodes. The implementation of this algorithm is:

```
// This algorithm solves the max flow problem in a directed graph
    in O(max_flow * E)

// Here the graph is represented by an adjacency matrix, but it can
     be easily changed to an adjacency list
```

Algorithm 4 Dinic

```
procedure Dinic(G)
    visited ← ∅
    time ← 0
    parent ← ∅
    low ← ∅
    disc ← ∅
    AP ← ∅
    bridge ← ∅
    for all v ∈ V do
        visited[v] ← false
        parent[v] ← −1
        low[v] ← ∞
        disc[v] ← ∞
    end for
    for all v ∈ V do
        if visited[v] = false then
            DinicUtil(G, v, visited, time, parent, low, disc, AP, bridge)
        end if
    end for
end procedure
procedure DinicUtil(G, v, visited, time, parent, low, disc, AP, bridge)
    visited[v] ← true
    disc[v] ← time
    low[v] ← time
    time ← time + 1
    children ← 0
    for all u ∈ Adj(v) do
        if visited[u] = false then
            parent[u] ← v
            children ← children + 1
            DinicUtil(G, u, visited, time, parent, low, disc, AP, bridge)
            low[v] ← min(low[v], low[u])
            if parent[v] = −1 and children > 1 then
                AP[v] ← true
            end if
            if parent[v]! = −1 and low[u] ≥ disc[v] then
                AP[v] ← true
            end if
            if low[u] > disc[v] then
                bridge[v][u] ← true
            end if
        else
            low[v] ← min(low[v], disc[u])
        end if
    end for
```

```cpp
// The algorithm is based on the push-relabel algorithm, which is a
     variant of the relabel-to-front algorithm

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink
  't' in residual graph. Also fills parent[] to store the
  path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
     as visited
    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < V; v++) {
            if (visited[v] == false && rGraph[u][v] > 0) {
                // If we find a connection to the sink node, then
    there is no point in BFS anymore We just have to set its parent
     and can return true
                if (v == t) {
                    parent[v] = u;
                    return true;
                }
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
```

```
    }
    // We didn't reach sink in BFS starting from source, so return
    false
    return false;
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;
    // Create a residual graph and fill the residual graph with
    given capacities in the original graph as  residual capacities
    in residual graph
    int rGraph[V]
              [V]; // Residual graph where rGraph[i][j] indicates
    residual capacity of edge from i to j (if there is an edge. If
    rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    int max_flow = 0; // There is no flow initially
    // Augment the flow while there is path from source to sink
    while (bfs(rGraph, s, t, parent)) {
        // Find minimum residual capacity of the edges along the
    path filled by BFS. Or we can say find the maximum flow through
     the path found.
        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }
        // update residual capacities of the edges and reverse
    edges along the path
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
        // Add path flow to overall flow
        max_flow += path_flow;
    }
    // Return the overall flow
    return max_flow;
}

int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V]
        = { { 0, 16, 13, 0, 0, 0 }, { 0, 0, 10, 12, 0, 0 },
            { 0, 4, 0, 0, 14, 0 },  { 0, 0, 9, 0, 0, 20 },
            { 0, 0, 0, 7, 0, 4 },   { 0, 0, 0, 0, 0, 0 } };

    cout << "The maximum possible flow is "
         << fordFulkerson(graph, 0, 5);

    return 0;
}
```

../Graphs/FordFulkerson.cpp

In order to better understand the adjecency matrix in the code Figure 4.1 shows the graph that is used in the code.

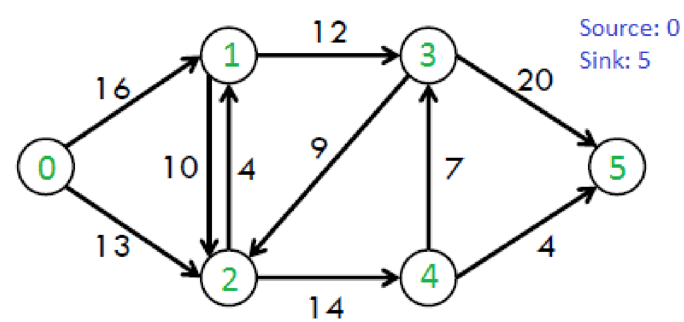

Figure 4.1: Ford Fulkerson

4.5 Dijkstra

The Dijkstra algorithm is a useful algorithm to find the shortest path between two nodes. The implementation of this algorithm is:

```
const int INF = 1e9;
vector<vector<pair<int, int>>> adj;  //To store the node to which
    the edge flows to and the weight of the edge

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}

//In order to restore the path, we need to store the parent of each
     node in the shortest path tree

vector<int> restore_path(int s, int t, vector<int> const& p) {
    vector<int> path;

    for (int v = t; v != s; v = p[v]){
        if(v == -1){
            return {};
        }
        path.push_back(v);
    }
    path.push_back(s);

    reverse(path.begin(), path.end());
    return path;
}
```

../Graphs/Dijkstra.cpp

Another implementation of this algorithm is the one that is done using a priority queue, the implementation of this algorithm is:

```
// Dijkstra Modification thanks to pacha2880

vi dijkstra(int n, vector<vii> &g, int s, vi &par)
{
    vi dis(n, MOD), vis(n);
    dis[s] = 0;
    priority_queue<ii> que;
    que.push({0, s});
    while(!que.empty())
    {
        int node = que.top().s;
        que.pop();
        if(vis[node]) continue;
        vis[node] = 1;
        for(ii go : g[node])
            if(dis[go.f] > dis[node] + go.s)
            {
                dis[go.f] = dis[node] + go.s;
                par[go.f] = node;
                que.push({-dis[go.f], go.f});
            }
    }
```

```
    return dis;
}

vi path;
vi par(n+1, -1);
int t;
if(dis[t] == MOD){
    do {
        path.pb(t);
        t = par[t];
    } while(t != -1);
}
```

../Graphs/Dijkstra–Mod.cpp

4.6 Bellman Ford

The Bellman Ford algorithm is a useful algorithm to find the shortest path between two nodes. Bellman Ford diferenctiates from Dijkstra in the fact that it can be used in graphs with negative edges. The algorithm is as follows:

The implementation of this algorithm is:

```
struct edge
{
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1e9;

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    for (int i=0; i<n-1; ++i)
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF) // is needed only if the graph
    contains negative weight edges: no such verification would
    result in relaxation from the vertices to which paths have not
    yet found, and incorrect distance
```

Algorithm 5 Bellman Ford

procedure BellmanFord(G)
  $dist \leftarrow \emptyset$
  $parent \leftarrow \emptyset$
  for all $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $parent[v] \leftarrow -1$
  end for
  $dist[s] \leftarrow 0$
  for $i = 0$ to $|V| - 1$ **do**
    for all $v \in V$ **do**
      for all $u \in Adj(v)$ **do**
        if $dist[u] > dist[v] + w(v, u)$ **then**
          $dist[u] \leftarrow dist[v] + w(v, u)$
          $parent[u] \leftarrow v$
        end if
      end for
    end for
  end for
  for all $v \in V$ **do**
    for all $u \in Adj(v)$ **do**
      if $dist[u] > dist[v] + w(v, u)$ **then**
        $dist[u] \leftarrow -\infty$
        $parent[u] \leftarrow -1$
      end if
    end for
  end for
end procedure

```
                d[e[j].b] = min (d[e[j].b], d[e[j].a] + e[j].cost);
    // display d, for example, on the screen
}


// An improvement in the time of the algorithm could be introduced
    by keeping the flag so that no time is wasted in visiting all
    edges.

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    for (;;) // equivalent to while (true)
    {
        bool any = false;

        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    any = true;
                }

        if (!any) break;
    }
    // display d, for example, on the screen
}

//To retrieve the path of thr Bellman Ford algorithm, you need to
    keep the previous vertex in the path, and then go back from the
     end to the beginning.

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    vector<int> p (n, -1);

    for (;;)
    {
        bool any = false;
        for (int j = 0; j < m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    p[e[j].b] = e[j].a;
                    any = true;
                }
        if (!any)  break;
    }

    if (d[t] == INF)
        cout << "No path from " << v << " to " << t << ".";
    else
    {
        vector<int> path;
        for (int cur = t; cur != -1; cur = p[cur])
            path.push_back (cur);
        reverse (path.begin(), path.end());

        cout << "Path from " << v << " to " << t << ": ";
        for (size_t i=0; i<path.size(); ++i)
            cout << path[i] << ' ';
    }
}


//Negative cycle detection

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    vector<int> p (n, - 1);
    int x;
    for (int i=0; i<n; ++i)
    {
        x = -1;
        for (int j=0; j<m; ++j)
```

```
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);
                    p[e[j].b] = e[j].a;
                    x = e[j].b;
                }
    }

    if (x == -1)
        cout << "No␣negative␣cycle␣from␣" << v;
    else
    {
        int y = x;
        for (int i=0; i<n; ++i)
            y = p[y];

        vector<int> path;
        for (int cur=y; ; cur=p[cur])
        {
            path.push_back (cur);
            if (cur == y && path.size() > 1)
                break;
        }
        reverse (path.begin(), path.end());

        cout << "Negative␣cycle:␣";
        for (size_t i=0; i<path.size(); ++i)
            cout << path[i] << '␣';
    }
}


// The SPFA (Shortest Path Faster Algorithm) is an improvement of
   Bellman Ford which takes advantage of the fact that not all
   attempts at relaxation will work

const int INF = 1e9;
vector<vector<pair<int, int>>> adj;

bool spfa(int s, vector<int>& d) {
    int n = adj.size();
    d.assign(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;

    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        inqueue[v] = false;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                if (!inqueue[to]) {
                    q.push(to);
                    inqueue[to] = true;
                    cnt[to]++;
                    if (cnt[to] > n)
                        return false;  // negative cycle
                }
            }
        }
    }
    return true;
}
```

../Graphs/BellmanFord.cpp

4.7 Hamiltonian Cycle

The Hamiltonian cycle of undirected graph G ¡= V , E¿ is the cycle containing each vertex in V. -If graph contains a Hamiltonian cycle, it is called Hamiltonian graph otherwise it is non-Hamiltonian.

Finding a Hamiltonian cycle in a graph is a well-known problem with many real-world applications, such as in network routing and scheduling.

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function. Input: A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0. Output: An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

A Hamiltonian cycle is a cycle that visits every node in the graph. The implementation of this algorithm is:

```
/* C++ program for solution of Hamiltonian Cycle problem using
    backtracking */
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
    index 'pos' in the Hamiltonian Cycle constructed so far (stored
     in 'path[]') */
bool isSafe(int v, bool graph[V][V],
         int path[], int pos)
{
   /* Check if this vertex is an adjacent vertex of the previously
    added vertex. */
   if (graph [path[pos - 1]][ v ] == 0)
      return false;

   /* Check if the vertex has already been included.  This step can
     be optimized by creating an array of size V */
   for (int i = 0; i < pos; i++)
      if (path[i] == v)
         return false;

   return true;
}

/* A recursive utility function to solve hamiltonian cycle problem
    */
bool hamCycleUtil(bool graph[V][V],
            int path[], int pos)
{
   /* base case: If all vertices are  in Hamiltonian Cycle */
   if (pos == V)
   {
      // And if there is an edge from the last included vertex to
    the first vertex
      if (graph[path[pos - 1]][path[0]] == 1)
         return true;
      else
         return false;
   }

   // Try different vertices as a next candidate in Hamiltonian
    Cycle. We don't try for 0 as we included 0 as starting point in
     hamCycle()
   for (int v = 1; v < V; v++)
   {
      /* Check if this vertex can be added to Hamiltonian Cycle */
      if (isSafe(v, graph, path, pos))
      {
         path[pos] = v;

         /* recur to construct rest of the path */
         if (hamCycleUtil (graph, path, pos + 1) == true)
            return true;

         /* If adding vertex v doesn't lead to a solution, then
    remove it */
         path[pos] = -1;
      }
   }

   /* If no vertex can be added to Hamiltonian Cycle constructed so
```

```
    far, then return false */
   return false;
}

/* This function solves the Hamiltonian Cycle problem using
    Backtracking. It mainly uses hamCycleUtil() to solve the
    problem. It returns false if there is no Hamiltonian Cycle
    possible, otherwise return true and prints the path. Please
    note that there may be more than one solutions, this function
    prints one of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
   int *path = new int[V];
   for (int i = 0; i < V; i++)
      path[i] = -1;

   /* Let us put vertex 0 as the first vertex in the path. If there
     is a Hamiltonian Cycle, then the path can be started from any
    point of the cycle as the graph is undirected */
   path[0] = 0;
   if (hamCycleUtil(graph, path, 1) == false )
   {
      cout << "\nSolution␣does␣not␣exist";
      return false;
   }

   printSolution(path);
   return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
   cout << "Solution␣Exists:"
         "␣Following␣is␣one␣Hamiltonian␣Cycle␣\n";
   for (int i = 0; i < V; i++)
      cout << path[i] << "␣";

   // Let us print the first vertex again to show the complete
    cycle
   cout << path[0] << "␣";
   cout << endl;
}

// Driver Code
int main()
{
   /* Let us create the following graph
      (0)--(1)--(2)
      | / \ |
      | / \ |
      | / \ |
      (3)-------(4) */
   bool graph1[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 1},
                        {0, 1, 1, 1, 0}};

   // Print the solution
   hamCycle(graph1);

   /* Let us create the following graph
   (0)--(1)--(2)
   | / \ |
   | / \ |
   | / \ |
   (3) (4) */
   bool graph2[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 0},
                        {0, 1, 1, 0, 0}};

   // Print the solution
   hamCycle(graph2);

   return 0;
}
```

../Graphs/Hamiltonian.cpp

4.8 Eulerian Cycle

The problem is same as following question. "Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once". A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to Hamiltonian Path which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in O(V+E) time. Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

Eulerian Cycle: An undirected graph has Eulerian cycle if following two conditions are true.

All vertices with non-zero degree are connected. We don't care about vertices with zero degree because they don't belong to Eulerian Cycle or Path (we only consider all edges). All vertices have even degree. Eulerian Path: An undirected graph has Eulerian Path if following two conditions are true.

Same as condition (a) for Eulerian Cycle. If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

An Eulerian cycle is a cycle that visits every edge in the graph. The implementation of this algorithm is:

```
// A C++ program to check if a given graph is Eulerian or not
#include<iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
{
   int V; // No. of vertices list<int> *adj; // A dynamic array of
    adjacency lists
public:
   // Constructor and destructor
   Graph(int V) {this->V = V; adj = new list<int>[V]; }
   ~Graph() { delete [] adj; } // To avoid memory leak

   // function to add an edge to graph
   void addEdge(int v, int w);

   // Method to check if this graph is Eulerian or not
   int isEulerian();

   // Method to check if all non-zero degree vertices are connected
   bool isConnected();

   // Function to do DFS starting from v. Used in isConnected();
   void DFSUtil(int v, bool visited[]);
};

void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w);
   adj[w].push_back(v); // Note: the graph is undirected
}

void Graph::DFSUtil(int v, bool visited[])
{
   // Mark the current node as visited and print it
   visited[v] = true;

   // Recur for all the vertices adjacent to this vertex
   list<int>::iterator i;
   for (i = adj[v].begin(); i != adj[v].end(); ++i)
      if (!visited[*i])
         DFSUtil(*i, visited);
}

// Method to check if all non-zero degree vertices are connected.
    It mainly does DFS traversal starting from
bool Graph::isConnected()
{
   // Mark all the vertices as not visited
   bool visited[V];
   int i;
   for (i = 0; i < V; i++)
      visited[i] = false;

   // Find a vertex with non-zero degree
   for (i = 0; i < V; i++)
      if (adj[i].size() != 0)
```

```
        break;

   // If there are no edges in the graph, return true
   if (i == V)
      return true;

   // Start DFS traversal from a vertex with non-zero degree
   DFSUtil(i, visited);

   // Check if all non-zero degree vertices are visited
   for (i = 0; i < V; i++)
   if (visited[i] == false && adj[i].size() > 0)
        return false;

   return true;
}

/* The function returns one of the following values
0 --> If graph is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian) */
int Graph::isEulerian()
{
   // Check if all non-zero degree vertices are connected
   if (isConnected() == false)
      return 0;

   // Count vertices with odd degree
   int odd = 0;
   for (int i = 0; i < V; i++)
      if (adj[i].size() & 1)
         odd++;

   // If count is more than 2, then graph is not Eulerian
   if (odd > 2)
      return 0;

   // If odd count is 2, then semi-eulerian.
   // If odd count is 0, then eulerian
   // Note that odd count can never be 1 for undirected graph
   return (odd)? 1 : 2;
}

// Function to run test cases
void test(Graph &g)
{
   int res = g.isEulerian();
   if (res == 0)
      cout << "graph is not Eulerian\n";
   else if (res == 1)
      cout << "graph has a Euler path\n";
   else
      cout << "graph has a Euler cycle\n";
}

// Driver program to test above function
int main()
{
   // Let us create and test graphs shown in above figures
   Graph g1(5);
   g1.addEdge(1, 0);
   g1.addEdge(0, 2);
   g1.addEdge(2, 1);
   g1.addEdge(0, 3);
   g1.addEdge(3, 4);
   test(g1);

   Graph g2(5);
   g2.addEdge(1, 0);
   g2.addEdge(0, 2);
   g2.addEdge(2, 1);
   g2.addEdge(0, 3);
   g2.addEdge(3, 4);
   g2.addEdge(4, 0);
   test(g2);

   Graph g3(5);
   g3.addEdge(1, 0);
   g3.addEdge(0, 2);
   g3.addEdge(2, 1);
   g3.addEdge(0, 3);
   g3.addEdge(3, 4);
```

```
    g3.addEdge(1, 3);
    test(g3);

    // Let us create a graph with 3 vertices
    // connected in the form of cycle
    Graph g4(3);
    g4.addEdge(0, 1);
    g4.addEdge(1, 2);
    g4.addEdge(2, 0);
    test(g4);

    // Let us create a graph with all vertices
    // with zero degree
    Graph g5(3);
    test(g5);

    return 0;
}
```

../Graphs/Eulerian.cpp

Chapter 5

Dynamic Programming

This chapter shows some useful algorithms and implementations required to solve problems that require Dynamic Programming.

Some of the algorithms and implementations are as follows:

```
signed main()
{
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    // freopen("asd.txt", "r", stdin);
    // freopen("qwe.txt", "w", stdout);

    int n;
    cin>>n;
    int dp[n+1];
    dp[0]=1;
    //dice combinations
    fore(i,1,n+1){
        dp[i]=0;
        fore(j,1,7){
            if(i-j>=0){
                dp[i]+=dp[i-j];
                dp[i]%=MOD;
            }
        }
    }
    cout << dp[n] << '\n';

    return 0;
}
```

../DP/dp1.cpp

```
signed main()
{
    //La cantidad minimas de monedas para llegar a k o unbounded
    knapsack problem.
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    // freopen("asd.txt", "r", stdin);
    // freopen("qwe.txt", "w", stdout);

    int n,k;
    cin>>n>>k;
    vector<int> dp(k+1,1e9);
    int a[n];
    for(int i=0;i<n;i++)cin>>a[i];
    dp[0]=0;
    for(int i=1;i<=k;i++){
        for(int j=0;j<n;j++){
            if(i-a[j]>=0){
                dp[i]=min(dp[i],dp[i-a[j]]+1);
            }
        }
    }
    if(dp[k]==1e9)cout<<-1;
    else cout<<dp[k];
```

```
    return 0;
}
```

../DP/dp2.cpp

```
ll dp[1000001];

const int MOD = (int) 1e9 + 7;

int main(){
   int n, x; cin >> n >> x;
   vi coins(n);
   for (int i = 0; i < n; i++) {
      cin >> coins[i];
   }
   dp[0] = 1;
   for (int weight = 0; weight <= x; weight++) {
      for (int i = 1; i <= n; i++) {
         if(weight - coins[i - 1] >= 0) {
            dp[weight] += dp[weight - coins[i - 1]];
            dp[weight] %= MOD;
         }
      }
   }
   cout << dp[x] << '\n';
}
```

../DP/dp3.cpp

```
//combination de monedas en orden
ll dp[1000001];

const int MOD = (int) 1e9 + 7;

int main(){
   int n, x; cin >> n >> x;
   vi coins(n);
   for (int i = 0; i < n; i++) {
      cin >> coins[i];
   }
   dp[0] = 1;
   for (int i = 1; i <= n; i++) {
      for (int weight = 0; weight <= x; weight++) {
         if(weight - coins[i - 1] >= 0) {  // prevenir casos bound
            dp[weight] += dp[weight - coins[i - 1]];
            dp[weight] %= MOD;
         }
      }
   }
   cout << dp[x] << '\n';
}
```

../DP/dp4.cpp

```
#include <bits/stdc++.h>
using namespace std;
//You are given an integer n On each step, you may subtract one of
    the digits from the number.How many steps are required to make
    the number equal to 0
int main() {
  int n;
  cin >> n;
  vector<int> dp(n+1,1e9);
  dp[0] = 0;
  for (int i = 0; i <= n; i++) {
    for (char c : to_string(i)) {
      dp[i] = min(dp[i], dp[i-(c-'0')]+1);
    }
  }
  cout << dp[n] << endl;
}
```

../DP/dp5.cpp

```
#include <bits/stdc++.h>
using namespace std;

int main() {
// there is one way to reach (0,0), dp[0][0] = 1.
//Consider an n*n grid whose squares may have traps. It is not
    allowed to move to a square with a trap.Your task is to
    calculate the number of paths from the upper-left square to the
     lower-right square. You can only move right or down
  int mod = 1e9+7;
```

```
  int n;
  cin >> n;
  vector<vector<int>> dp(n, vector<int>(n, 0));
  dp[0][0] = 1;
  for (int i = 0; i < n; i++) {
    string row;
    cin >> row;
    for (int j = 0; j < n; j++) {
      if (row[j] == '.') {
   if (i > 0) {
     (dp[i][j] += dp[i-1][j]) %= mod;
   }
   if (j > 0) {
     (dp[i][j] += dp[i][j-1]) %= mod;
   }
      } else {
   dp[i][j] = 0;
      }
    }
  }
  cout << dp[n-1][n-1] << endl;
}
```

../DP/dp6.cpp

```
/You are in a book shop which sells n
 different books. You know the price and number of pages of each
    book.

You have decided that the total price of your purchases will be at
    most x
. What is the maximum number of pages you can buy? You can buy each
     book at most once.#include <bits/stdc++.h>
using namespace std;

int main() {
  int n, x;
  cin >> n >> x;
  vector<int> price(n), pages(n);
  for (int i;i<n;i++) cin >> price[i];
  for (int i;i<n;i++) cin >> pages[i];
  vector<vector<int>> dp(n+1,vector<int>(x+1,0));
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= x; j++) {
      dp[i][j] = dp[i-1][j];
      int left = j-price[i-1];
      if (left >= 0) {
   dp[i][j] = max(dp[i][j], dp[i-1][left]+pages[i-1]);
      }
    }
  }
  cout << dp[n][x] << endl;
}
```

../DP/dp7.cpp

```
//You know that an array has nintegers between 1 and m, and the
    absolute difference between two adjacent values is at most 1.
//Given a description of the array where some values may be unknown
    , your task is to count the number of arrays that match the
    description.
#include <bits/stdc++.h>
using namespace std;

int main() {
  int mod = 1e9+7;
  int n, m;
  cin >> n >> m;
  vector<vector<int>> dp(n,vector<int>(m+1,0));
  int x0;
  cin >> x0;
  if (x0 == 0) {
    fill(dp[0].begin(), dp[0].end(), 1);
//igual memset pero algo raro
  } else {
    dp[0][x0] = 1;
  }
  for (int i = 1; i < n; i++) {
    int x;
    cin >> x;
    if (x == 0) {
      for (int j = 1; j <= m; j++) {
   for (int k : {j-1,j,j+1}) {
     if (k >= 1 && k <= m) {
```

```
        (dp[i][j] += dp[i-1][k]) %= mod;
      }
    }
      }
    } else {
      for (int k : {x-1,x,x+1}) {
    if (k >= 1 && k <= m) {
      (dp[i][x] += dp[i-1][k]) %= mod;
    }
      }
    }
  }
  int ans = 0;
  for (int j = 1; j <= m; j++) {
    (ans += dp[n-1][j]) %= mod;
  }
  cout << ans << endl;
}
```

../DP/dp8.cpp

5.1 Knapsack Problem

The knapsack problem is a problem that consists of finding the maximum value of a set of items that can be placed in a knapsack of a given weight. The problem can be solved using Dynamic Programming.

The implementation can be done as follows:

```
// A Dynamic Programming based solution for 0-1 Knapsack problem
#include <iostream>

using namespace std;

// A utility function that returns maximum of two integers
int max(int a, int b)
{
    return (a > b) ? a : b;
}

// Returns the maximum value that can be put in a knapsack of
    capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w]
                        = max(val[i - 1] + K[i - 1][w - wt[i - 1]],
     K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    return K[n][W];
}

int main()
{
    cout << "Enter␣the␣number␣of␣items␣in␣a␣Knapsack:";
    int n, W;
    cin >> n;
    int val[n], wt[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter␣value␣and␣weight␣for␣item␣" << i << ":";
        cin >> val[i];
        cin >> wt[i];
    }

    //    int val[] = { 60, 100, 120 };
    //    int wt[] = { 10, 20, 30 };
    //    int W = 50;
    cout << "Enter␣the␣capacity␣of␣knapsack";
```

```
    cin >> W;
    cout << knapSack(W, wt, val, n);

    return 0;
}
```

../DP/knapsack.cpp

5.2 Divide and Conquer

The divide and conquer algorithm is a recursive algorithm that divides the problem into smaller subproblems and solves them recursively. The algorithm is as follows:

Algorithm 6 Divide and Conquer

```
procedure DivideAndConquer(A)
    if A has only one element then
        return A
    end if
    B ← DivideAndConquer(A[0..n/2])
    C ← DivideAndConquer(A[n/2+1..n])
    return Merge(B, C)
end procedure
```

The implementation can be done as follows:

```
/*
DP[i][j] = min( DP[i-1][k] + C[k][j] )
K[i][j] <= K[i][j+1]
*/

ll lastDP[tam], DP[tam];
int C[tam][tam]; // Cambiar a una funcion de costo si pre-procesar
    ocupa mucha memoria

void DC(int b, int e, int KL, int KR)
{
   int mid = (b + e) / 2;
   pair<ll, int> best = mp(-1, KL);

   for (int k = KL; k < min(mid, KR+1); k++)
   {
      best = max( best, mp(lastDP[k] + C[k+1][mid], k) );
   }

   DP[mid] = best.first;
   int K = best.second;

   if (b <= mid-1)
      DC(b, mid-1, KL, K);
   if (mid+1 <= e)
      DC(mid+1, e, K, KR);
}
```

../DP/DivideAndConquer.cpp

5.3 Digit DP

Digit DP is a technique that can be used to solve problems that require Dynamic Programming. The technique consists of solving the problem by using Dynamic Programming and the digits of the number.

The implementation can be done as follows:

```
#include <bits/stdc++.h>
using namespace std;

#define int long long int
#define pb push_back
#define pi pair<int, int>
#define fir first
#define sec second
#define MAXN 2001
#define mod 1000000007

int dp[20][20 * 9][2]; // a,b <= 10^18
vector<int> dig;

int solve(int i, int j, int k)
{
  if (i == dig.size())
    return (k) ? dp[i][j][k] = j : dp[i][j][k] = 0;
  if (dp[i][j][k] != -1)
```

```
    return dp[i][j][k];
  int sum = 0;
  if (k)
    for (int f = 0; f <= 9; f++)
      sum += solve(i + 1, j + f, k);
  if (!k)
    for (int f = 0; f <= dig[i]; f++)
      sum += solve(i + 1, j + f, (dig[i] != f) ? 1 : 0);
  return dp[i][j][k] = sum;
}
void get_digits(int n)
{
  dig.clear();
  while (n)
  {
    dig.pb(n % 10);
    n = n / 10;
  }
  reverse(dig.begin(), dig.end());
}
signed main()
{
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  int a, b;
  cin >> a >> b;
  get_digits(a);
  memset(dp, -1, sizeof(dp));
  int aa = solve(0, 0, 0);
  get_digits(b + 1);
  memset(dp, -1, sizeof(dp));
  int bb = solve(0, 0, 0);
  cout << bb - aa << endl;
  return 0;
}
```

../DP/digitdp.cpp

Chapter 6

Geometry

This chapter is a survey of the main results and algorithms useful to solve geometry problems.

6.1 Points and Lines

Some of the most useful functions used while describing points and lines are:

- `atan2(y,x)`: returns the angle between the positive x-axis and the vector (x,y).
- `hypot(x,y)`: returns the Euclidean distance between the origin and the point (x,y).
- `cross(a,b)`: returns the cross product of the vectors a and b.
- `dot(a,b)`: returns the dot product of the vectors a and b.
- `dist(a,b)`: returns the Euclidean distance between the points a and b.
- `dist2(a,b)`: returns the squared Euclidean distance between the points a and b.
- `ccw(a,b,c)`: returns true if the points a, b and c are in counterclockwise order.
- `collinear(a,b,c)`: returns true if the points a, b and c are collinear.
- `angle(a,b)`: returns the angle between the vectors a and b.
- `angle(a,b,c)`: returns the angle between the vectors a-b and c-b.
- `rotate(a,ang)`: returns the vector a rotated by ang radians.
- `rotate(a,ang,center)`: returns the vector a rotated by ang radians around the point center.
- `reflect(a,m)`: returns the reflection of the point a across the line m.
- `project(a,m)`: returns the projection of the point a onto the line m.
- `closest(a,m)`: returns the closest point on the line m to the point a.
- `intersect(a,b,c,d)`: returns true if the lines a-b and c-d intersect.

```
#include <bits/stdc++.h>
#define EPS       1e-9
struct line {double a,b,c;}; // ax + by + c = 0
bool areParallel(line a, line b)
{
   return((fabs(a.a-b.a)<EPS)&&(fabs(a.b-b.b)<EPS));
}
bool areSame(line a, line b)
{
   return areParallel(a,b)&&(fabs(a.c-b.c)<EPS);
}
struct point
{
   double x,y;
   point() {x=y=0;}
   point(double _x, double _y) : x(_x), y(_y) {}
   point operator+(point a) const
   {
```

```
      a.x+=x;
      a.y+=y;
      return a;
   }
};
double dist(point a, point b)
{
   return hypot(a.x-b.x,a.y-b.y);
}
void toline(point a, point b, line &l) //dados dos puntos
{
   if(fabs(a.x-b.x)<EPS)
      {l.a = 1, l.b = 0, l.c = -a.x; return;}
   l.a = -(a.y - b.y) / (a.x - b.x);
   l.b = 1;
   l.c = -l.a * a.x - a.y;
}
void tolinegr(point a, double gr, line &l) // a linea dado el
    gradiente
{
   l.a = -gr;
   l.b = 1;
   l.c = a.x * gr - a.y;
}
point tovec(point a, point b)
{
   return point(b.x-a.x,b.y-a.y);
}
point translate (point p, point v)
{
   return point(p.x+v.x,p.y+v.y);
}
point scale(point v, double sc)
{
   return point(v.x*sc,v.y*sc);
}
point rotate(point v, double theta) //rotacion antihorario ccw
{
   theta *= acos(-1)/180.0;
   return point(v.x*cos(theta)-v.y*sin(theta),v.x*sin(theta)+ v.y*
    cos(theta));
}
bool areIntersect(line l1, line l2, point &p) //interseccion de
    lineas
{
   if(areParallel(l1,l2)) return false;
   p.x = (-l1.c*l2.b + l2.c*l1.b) / (l1.a*l2.b-l2.a*l1.b);
   if(fabs(l1.b) > EPS) p.y = -(l1.a*p.x + l1.c);
   else
      p.y = -(l2.a*p.x +l2.c);
   return true;
}
point clos(point a, line l, line &pe) //closest point in a line and
     perpendicular line from a
{
   if(fabs(l.a) < EPS)
   {
      pe.a = 1, pe.b = 0, pe.c = -a.x;
      return point(a.x,-l.c);
   }
   if(fabs(l.b) < EPS)
   {
      pe.a = 0, pe.b = 1, pe.c = -a.y;
      return point(-l.c,a.y);
   }
   tolinegr(a, 1/(l.a),pe);
   areIntersect(l,pe,a);
   return a;
}
point reflexion(point p, point a, point b) // del punto p a linea
    ab
{
   line l,li;
   toline(a,b,li);
   point p1 = clos(p,li,l);
   p1 = p1+(tovec(p,p1));
   return p1;
}
double norm_sq(point a)
{
   return a.x * a.x + a.y * a.y;
}
```

```
double dot(point a, point b)
{
   return a.x*b.x + b.y*a.y;
}
double angle(point a, point b, point c) //b el del medio
{
   a = tovec(b,a), b = tovec(b,c);
   double res = dot(a,b);
   res = acos(res / (sqrt(norm_sq(a))*sqrt(norm_sq(b))));
   res*= 180.0/acos(-1);
   return res;
}
double cross(point a, point b) //prosucto cruz
{
   return a.x * b.y - a.y * b.x;
}
bool left(point a, point b, point c) //ccw
{
   c = tovec(b,c);
   a = tovec(b,a);
   return (cross(c,a)>0.0);
}
bool coolinear(point a, point b, point c)
{

   c = tovec(b,c);
   a = tovec(b,a);
   return (fabs(cross(c,a))<EPS);
}
double distToLine(point p, point a, point b, point &c) //con
    producto punto halla el punto
{
   //c = a + u*ab
   point ab = tovec(a,b), ap = tovec(a,p);
   double u = dot(ab,ap) / norm_sq(ab);
   c = translate(a, scale(ab,u));
   return dist(p,c);
}
double distToline1(point p, point a, point b) //con producto cruz
    solo distancia
{
   point ap = tovec(a,p), ab = tovec(a,b);
   return fabs(cross(ab,ap)/hypot(ab.x,ab.y));
}
```

../Geometry/PointsAndLines.cpp

6.1.1 Lines

In order to represent lines and find their intersection, we can use the following struct:

```
struct line
{
  double a, b, c;
  line(point p, point q)
  {
    a = p.y - q.y;
    b = q.x - p.x;
    c = -a * p.x - b * p.y;
  };
  void setOrigin(point p) { c += a * p.x + b * p.y; } //trasladar
    linea como si p fuera el origen
};

double det(double a, double b, double c, double d)
{
   return a * d - b * c;
}

point intersec(line a, line b) //primero estar seguro si no son
    paralelas
{
   double d = -det(a.a, a.b, b.a, b.b);
   return point(det(a.c, a.b, b.c, b.b) / d, det(a.a, a.c, b.a, b.c
    ) / d);
}
```

../Geometry/Line.cpp

6.2 Convex Hull

The convex hull of a set of points is the smallest convex polygon that contains all the points. There are several algorithms to find the convex hull of a set of points. The most common ones are:

- **Graham Scan**: This algorithm finds the convex hull in O(n log n) time. It is based on the following idea: the convex hull of a set of points is the set of points that are on the boundary of the convex hull. Therefore, we can find the convex hull by finding the points that are on the boundary of the convex hull. The algorithm works as follows:

```
struct point
{
   double x, y;
   point() {}
   point(double x, double y) : x(x), y(y){}
};
double dist(point a, point b)
{
   return hypot(a.x - b.x, a.y - b.y);
}
double cross2(point a, point b)
{
   return a.x*b.y - a.y*b.x;
}
point tovec(point a, point b)
{
   return point(b.x - a.x, b.y - a.y);
}
double cross(point a, point b, point c)
{
   return cross2(tovec(a, b), tovec(a, c));
}
bool eq(double a, double b)
{
   return fabs(a-b) < EPS;
}
point mini;
bool comp(point a, point b)
{
   point ta = tovec(mini, a);
   point tb = tovec(mini, b);
   double ana = atan2(ta.y, ta.x), anb = atan2(tb.y, tb.x);
   if(eq(ana, anb))
      return dist(a, mini) < dist(b, mini);
   return ana < anb - EPS;
}
//no hay 3 puntos colineales
vector<point> hull(vector<point> p)
{
   for(int i = 1; i < p.size(); i++)
   {
      if(eq(p[i].y, p[0].y) )
      {
         if(p[i].x < p[0].x - EPS)
            swap(p[i], p[0]);
      }
      else
      {
         if(p[i].y < p[0].y - EPS)
            swap(p[i], p[0]);
      }
   }
   mini = p[0];
   sort(++p.begin(), p.end(), comp);
   p.pb(p[0]);
   vector<point> res;
   res.pb(p[0]);
   res.pb(p[1]);
   for(int i = 2; i < p.size(); i++)
   {
      while(cross(res[res.size()-2], res.back(), p[i]) < EPS)
      {
         res.pop_back();
      }
      res.pb(p[i]);
   }
   return res;
}
```

../Geometry/ConvexHullGraham.cpp

- **Jarvis March**: This algorithm finds the convex hull in O(nh) time, where h is the number of points on the convex hull. It is based on the following idea:

we can find the convex hull by starting at a point and rotating clockwise until we reach the starting point. The algorithm works as follows:

- **Monotone Chain**: This algorithm finds the convex hull in O(n log n) time. It is based on the following idea: we can find the convex hull by finding the upper and lower hulls of the set of points. The algorithm works as follows:

```
// devuelve horario
vector<point> hull(vector<point> p)
{
   int n = p.size();
   vector<point> h;
   sort(all(p));
   fore(i, 0, n)
   {
      while(h.size() >= 2 && p[i].left(h[sz(h) - 2], h.back()))
    h.pop_back();
      h.push_back(p[i]);
   }
   h.pop_back();
   int k = h.size();
   for(int i = n-1; i > -1; i--)
   {
      while(h.size() >= k + 2 && p[i].left(h[sz(h) - 2], h.back
   ())) h.pop_back();
      h.pb(p[i]);
   }
   h.pop_back();
   return h;
}
```

../Geometry/ConvexHullMonotone.cpp

6.3 Polygon

A polygon is a closed plane figure that is bounded by a finite chain of straight line segments closing in a loop to form a closed chain or circuit. A polygon is simple if it does not intersect itself. A polygon is convex if it contains no line segment that is strictly inside the polygon. A polygon is monotone if it can be decomposed into a sequence of monotone polygons. A polygon is simple and convex if it is both simple and convex. A polygon is simple and monotone if it is both simple and monotone. A polygon is convex and monotone if it is both convex and monotone. A polygon is simple, convex and monotone if it is both simple, convex and monotone.

```
struct point{
   double x,y;
   point(){x=y=0;}
   point(double X, double Y): x(X), y(Y) {}
   point operator+(point a) const
   {
      a.x+=x;
      a.y+=y;
      return a;
   }
   bool operator<(point a) const
   {
      return (a.x == x? a.y < y : a.x < x);
   }
};
double dist(point a, point b)
{
   return hypot(a.x-b.x, a.y-b.y);
}
point tovec(point a, point b)
{
   return point(b.x-a.x,b.y-a.y);
}
double norm(point a)
{
   return hypot(a.x,a.y);
}
double dot(point a, point b)
{
   return a.x*b.x + a.y*b.y;
}
double cross(point a, point b)
{
   return a.x*b.y - a.y*b.x;
}
bool ccw(point a, point b, point c)
{
   return cross(tovec(a,b),tovec(a,c)) >= 0; //depende si se acepta
     colinear o no
```

```
}
double an(point a, point b, point c)
{
   a = tovec(b,a), b = tovec(b,c);
   return acos(dot(a,b)/(norm(a)*norm(b)));
}
double perimeter(const vector<point> &p)
{
   double result = 0.0;
   for(int i = 0; i<p.size()-1; i++)
   {
      result += dist(p[i],p[i+1]);
   }
   return result;
}
double area(const vector<point> &p)
{
   double result = 0.0;
   for(int i=0;i<p.size()-1;i++)
   {
      result += p[i].x*p[i+1].y - p[i].y*p[i+1].x;;
   }
   return fabs(result)/2.0;
}
point lineIntersectSeg(point p, point q, point A, point B)
{
   double a = B.y - A.y;
   double b = A.x - B.x;
   double c = B.x * A.y - A.x * B.y;
   double u = fabs(a * p.x + b * p.y + c);
   double v = fabs(a * q.x + b * q.y + c);
   return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) /
    (u+v));
}
vector<point> cutPolygon(point a, point b, const vector<point> &Q)
{
   vector<point> P;
   for (int i = 0; i < (int)Q.size(); i++) {
      double left1 = cross(tovec(a, b), tovec(a, Q[i])), left2 = 0;
      if (i != (int)Q.size()-1) left2 = cross(tovec(a, b), tovec(a,
     Q[i+1]));
      if (left1 > -EPS) P.push_back(Q[i]); // Q[i] is on the left
    of ab ; left1 < EPS para la derecha
      if (left1 * left2 < -EPS) // edge (Q[i], Q[i+1]) crosses line
     ab
         P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
   }
   if (!P.empty() && (P.back().x != P.front().x || P.back().y != P.
    front().y))
      P.push_back(P.front()); // make P's first point = P's last
    point
   return P;
}
bool isConvex(const vector<point> &p)
{
   int sz = p.size();
   if(sz<=3) return false;
   bool left = ccw(p[0],p[1],p[2]);
   cout<<left<<endl;
   for(int i = 1; i < sz - 1; i++)
   {
      cout<<i<<' '<<ccw(p[i],p[i+1],p[((i+2)==sz)? 1:i+2])<<endl;
      if(ccw(p[i],p[i+1],p[((i+2)==sz)? 1:i+2])!=left)
         return false;
   }
   return true;
}
bool isIn(const vector<point> &p, point a)
{
   double ang = 0;
   int sz = p.size();
   if(sz == 0) return false;
   for(int i = 0; i<sz-1;i++)
   {
      if(ccw(a,p[i],p[i+1]))
         ang += an(p[i],a,p[i+1]);
      else
         ang -= an(p[i],a,p[i+1]);
   }
   cout<<ang<<endl;
   return fabs(ang - 2.0*PI) < EPS;
}
```

../Geometry/Polygon.cpp

6.3.1 Triangles and Circles

```
struct point
      {
         double x,y;
         point() {x=0.0; y = 0.0;}
         point(int _x, int _y) : x(_x), y(_y) {}
         point operator+(point b) const
         {
            b.x += x;
            b.y+=y;
            return b;
         }
      };
      struct line
      {
         double a,b,c;
      };
      double dist(point a, point b)
      {
         return hypot(fabs(a.x-b.x),fabs(a.y-b.y));
      }
      point tovec(point a, point b)
      {
         return point(b.x-a.x,b.y-a.y);
      }
      point translate(point a, point b)
      {
         a= a+b;
         return a;
      }
      point scale(point a, double s)
      {
         a.x*=s;
         a.y*= s;
         return a;
      }
```

```
  void pointsToLine(point a, point b, line &l) //linea dados 2
puntos
  {
     if(fabs(a.x-b.x)<EPS)
     {
        l.a = 1, l.b = 0, l.c = -a.x;
     }
     else
     {
        l.a = -(a.y-b.y) / (a.x - b.y), l.b = 1, l.c = -l.a * a
.x - a.y;
     }
  }
  double rInCircle(double ab, double bc, double ca)
  {
     double s = (ab+bc+ca)/2;
     return sqrt(s*(s-ab)*(s-bc)*(s-ca));
  }
  double rIncircle(point a, point b, point c)
  {
     return rInCircle(dist(a,b),dist(b,c),dist(c,a));
  }
  bool areParallel(line a, line b)
  {
     return (fabs(a.a-b.a)<EPS)&&(a.b == b.b);
  }
  bool areIntersect(line a, line b, point &c)
  {
     if(areParallel(a,b)) return false;
     c.x = (b.c*a.b-a.c*b.b) / (a.a*b.b-b.a*a.b);
     if(a.b == 0.0) c.y = -(b.b*c.x + b.c);
     else        c.y = -(a.b*c.x + a.c);
     return true;
  }
  double areaTri1(double a, double b, double c) //heron
  {
     double s = (a+b+c)/2;
     return sqrt(s*(s-a)*(s-b)*(s-c));
  }
  double areaTri(point a, point b, point c)
  {
```

```
    return areaTri1(dist(a,b),dist(b,c),dist(a,c));
  }
  line perp(line a, point p) //perpendicular
  {
    line res;
    if(a.b=0)
    {
      res.a = 0, res.b = 1, res.c = -p.y;
    }
    else
      if(fabs(a.a)<EPS)
      {
        res.a = 1, res.b = 0, res.c = -p.y;
      }
      else
      {
        res.a = -1.0/a.a, res.b = 1, res.c = -res.a*p.x-p.y;
      }

  }
  bool circumCircle(point a, point b, point c, point &ctr,
double &r) //circuncentro completo
  {
    double area = areaTri(a,b,c);
    if(fabs(area)<EPS) return 0;
    line l1, l2;
    pointsToLine(a,b,l2);
    pointsToLine(a,c,l2);
    point p1 = point((a.x+b.x)/2.0,(a.y+b.y)/2.0), p2 = point
((a.x+c.x)/2.0,(a.y+c.y)/2.0);
    l1 = perp(l1,p1), l2 = perp(l2,p2);
    areIntersect(l1,l2,ctr);
    r = dist(a,b)*dist(b,c)*dist(a,c)/(4.0*areaTri(a,b,c));
    return true;
  }
  bool isInCircum(point a, point b, point c, point p) //si esta
 dentro del circulo circunscrito
  {
    double r;
    point ctr;
    if(!circumCircle(a,b,c,ctr,r)) return false;
    return dist(ctr,p) <= r ;
  }
  bool inCircle(point a, point b, point c, point &ctr) //
incentro
  {
    double r = rIncircle(a,b,c);
    if(r< EPS) return false;
    line l1,l2;
    point p1;
    double ratio = dist(a,b) / dist(a,c);
    p1 = translate(b, scale(tovec(b,c),ratio/(1+ratio)));
    pointsToLine(a,p1,l1);
    ratio = dist(b,a) / dist(b,c);
    p1 = translate(a, scale(tovec(a,c),ratio/(1+ratio)));
    pointsToLine(b,p1,l2);
    areIntersect(l1,l2,ctr);
    return true;
  }
  line toLinep(point a, point b, point c) //para mediatriz
  {
    line l;
    if(b.x == c.x)
    {
      l.a = 0, l.b = 1 , l.c = -a.y;
    }
    else
      if(b.y == c.y)
      {
        l.a = 1, l.b = 0, l.c = -a.x;
      }
      else
      {
        l.a = 1/((b.y-a.y)/(b.x-a.x)), l.b = 1, l.c = -l.a*a
.x-a.y;
      }
      return l;
  }
  point circun(point a, point b, point c) //circuncentro
  {
    line l1, l2;
    l1 = toLinep(point((a.x+b.x)/2,(a.y+b.y)/2),a,b);
```

```
      l2 = toLinep(point((a.x+c.x)/2,(a.y+c.y)/2),a,c);
      areIntersect(l1,l2,a);
      return a;
   }
   bool circle2PtsRad(point a, point b, double r, point &c) //
 dados 2 puntos y un radio
   {
      double det = (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);
      det = r * r / det - 0.25;
      if(det < 0.0) return false;
      det = sqrt(det);
      c.x = (a.x + b.x) * 0.5 + (b.y-a.y) * det;
      c.y = (a.y + b.y) * 0.5 + (a.x-b.x) * det;
      return true;
   }
```

../Geometry/Triangle.cpp

6.4 Polar Sort

The polar sort is a sorting algorithm that sorts a set of points by their angle with respect to a given point. The algorithm works as follows:

```
/*typedef double T;
typedef complex<T> pt;
#define x real()
#define y imag()*/

//typedef long long ll;
//typedef long double ll;

struct point
{
   ll x, y;
   point() {}
   point(ll x, ll y): x(x), y(y) {}
   point operator -(point p) {return point(x - p.x, y - p.y);}
   point operator +(point p) {return point(x + p.x, y + p.y);}
   ll sq() {return x * x + y * y;}
   double abs() {return sqrt(sq());}
   ll operator ^(point p) {return x * p.y - y * p.x;}
   ll operator *(point p) {return x * p.x + y * p.y;}
   point operator *(ll a) {return point(x * a, y * a);}
   bool operator <(const point& p) const {return x == p.x ? y < p.y
     : x < p.x;}
   bool left(point a, point b) {return ((b - a) ^ (*this - a)) >=
    0;}
   ostream& operator<<(ostream& os) {
      return os << "("<< x << "," << y << ")";
   }

};

void polarSort(vector<point>& v) {
  sort(v.begin(), v.end(), [] (point a, point b) {
    const point origin{0, 0};
    bool ba = a < origin, bb = b < origin;
    if (ba != bb) { return ba < bb; }
    return (a^b) > 0;
  });
}
```

../Geometry/PolarSort.cpp