ACM ICPC Team Reference 2010

Team Exceptions
Universidade Federal do Espírito Santo

1. Configuration files and scripts

1.1. .vimrc.

```
set softtabstop=4
set shiftwidth=4
set tabstop=4
set expandtab
set ruler
set cindent
set backspace=indent,eol,start
set showcmd
set nocp
syntax on
hi Normal guibg=black guifg=white
colors zellner
```

1.2. C++ template.

```
#include <cstdio>
#include <string>
#include <cstring>
#include <vector>
#include <algorithm>
#include <map>
#include <utility>
#include <cmath>
#include <queue>
#include <stack>
#include <set>
#include <deque>
#include <iostream>
#include <sstream>

using namespace std;

#define REP(i,n) for(int i=0;i<(int)n;++i)
#define EACH(i,c) for(__typeof((c).begin()) i=(c).begin(); i!=(c).end(); ++i)
#define ALL(c) (c).begin(), (c).end()
#define SIZE(x) (int((x).size()))
const int INF = 0x3F3F3F3F;
const double PI = 2*acos(0);
const double EPS = 1e-10;

inline int cmp(double x, double y=0, double tol=EPS){
   return (x<=y+tol) ? (x+tol<y) ? -1 : 0 : 1;
}

int main(){

   return 0;
}
```

2. Useful functions

2.1. Replace all occurrences.

```
string replaceAll(string s, string f, string t) {
   string r;
   for (int p = 0; (p = s.find(f)) != s.npos; ) {
      r += s.substr(0, p) + t;
      s = s.substr(p + f.size());
   }
   return r + s;
}
```

2.2. Replace the first occurrence.

```
string replace(string s, string f, string t) {
   string r;
   int p = s.find(f);
   if (p != s.npos) {
      r += s.substr(0, p) + t;
      s = s.substr(p + f.size());
   }
   return r + s;
}
```

2.3. Split on all occurrences.

```
vector<string> splitAll(string s, string t) {
   vector<string> v;
   for (int p = 0; (p = s.find(t)) != s.npos; ) {
      v.push_back(s.substr(0, p));
      s = s.substr(p + t.size());
   }
   v.push_back(s);
   return v;
}
```

2.4. Split on first occurrence.

```
vector<string> split(string s, string t) {
   vector<string> v;
   int p = s.find(t);
   if (p != s.npos) {
      v.push_back(s.substr(0, p));
      s = s.substr(p + t.size());
   }
   v.push_back(s);
   return v;
}
```

3. String

3.1. Palindrome.

```
bool is_palindrome(string const &s){
   return equal(ALL(s),s.rbegin());
}
```

3.2. **Longest palindrome.**

```
int longest_palindrome(char *text, int n) {
   int rad[2*n], i, j, k;
   for(i=0,j=0;i<2*n;i+=k,j=max(j-k,0)){
      while(i-j>=0&&i+j+1<2*n&&text[(i-j)/2]==text[(i+j+1)/2]) ++j;
      rad[i] = j;
      for(k=1;i-k>=0&&rad[i]-k>=0&&rad[i-k]!=rad[i]-k;++k)
         rad[i+k] = min(rad[i-k], rad[i]-k);
   }
   return *max_element(rad, rad+2*n);
   // ret. centre of the longest palindrome
}
```

3.3. **Knuth-Morris-Pratt.**

```
int pi[MAXSZ], res[MAXSZ], nres;

void kmp(string text, string pattern) {
   nres = 0;
   pi[0] = -1;
   for(int i = 1; i < pattern.size(); ++i) {
      pi[i] = pi[i-1];
      while(pi[i] >= 0 && pattern[pi[i] + 1] != pattern[i])
         pi[i] = pi[pi[i]];
      if(pattern[pi[i] + 1] == pattern[i]) ++pi[i];
   }
   int k = -1; //k + 1 eh o tamanho do match atual
   for(int i = 0; i < text.size(); ++i) {
      while(k >= 0 && pattern[k + 1] != text[i])
         k = pi[k];
      if(pattern[k + 1] == text[i]) ++k;
      if(k + 1 == pattern.size()) {
         res[nres++] = i - k;
         k = pi[k];
      }
   }
}
```

3.4. **Suffix Tree.**

```
// Larsson-Sadakane's Suffix array Construction: O(n (log n)^2)
struct SAComp {
   const int h, *g;
   SAComp(int h, int* g) : h(h), g(g) {}
   bool operator() (int a, int b) {
      return a == b ? false : g[a] != g[b] ? g[a] < g[b] : g[a+h] < g[b+h];
   }
};
int *buildSA(char* t, int n) {
   int g[n+1], b[n+1], *v = new int[n+1];
   REP(i,n+1) v[i] = i, g[i] = t[i];
   b[0] = 0; b[n] = 0;

   sort(v, v+n+1, SAComp(0, g));
   for(int h = 1; b[n] != n ; h *= 2) {
      SAComp comp(h, g);
      sort(v, v+n+1, comp);
      REP(i, n) b[i+1] = b[i] + comp(v[i], v[i+1]);
      REP(i, n+1) g[v[i]] = b[i];
   }
   return v;
}
// Naive matching O(m log n)
int find(char *t, int n, char *p, int m, int *sa) {
   int a = 0, b = n;
   while (a < b) {
      int c = (a + b) / 2;
      if (strncmp(t+sa[c], p, m) < 0) a = c+1; else b = c;
   }
   return strncmp(t+sa[a], p, m) == 0 ? sa[a] : -1;
}

// Kasai-Lee-Arimura-Arikawa-Park's simple LCP computation: O(n)
int *buildLCP(char *t, int n, int *a) {
```

```
   int h = 0, b[n+1], *lcp = new int[n+1];
   REP(i, n+1) b[a[i]] = i;
   REP(i, n+1) {
      if (b[i]){
         for (int j = a[b[i]-1]; j+h<n && i+h<n && t[j+h] == t[i+h]; ++h);
         lcp[b[i]] = h;
      } else lcp[b[i]] = -1;
      if (h > 0) --h;
   }
   return lcp;
}
// call RMQ = buildRMQ(lcp, n+1)
int *buildRMQ(int *a, int n) {
   int logn = 1;
   for (int k = 1; k < n; k *= 2) ++logn;
   int *r = new int[n * logn];
   int *b = r; copy(a, a+n, b);
   for (int k = 1; k < n; k *= 2) {
      copy(b, b+n, b+n); b += n;
      REP(i, n-k) b[i] = min(b[i], b[i+k]);
   }
   return r;
}
// inner LCP computation with RMQ: O(1)
int minimum(int x, int y, int *rmq, int n) {
   int z = y - x, k = 0, e = 1, s; // y - x >= e = 2^k
   s = ( (z & 0xffff0000) != 0 ) << 4; z >>= s; e <<= s; k |= s;
   s = ( (z & 0x0000ff00) != 0 ) << 3; z >>= s; e <<= s; k |= s;
   s = ( (z & 0x000000f0) != 0 ) << 2; z >>= s; e <<= s; k |= s;
   s = ( (z & 0x0000000c) != 0 ) << 1; z >>= s; e <<= s; k |= s;
   s = ( (z & 0x00000002) != 0 ) << 0; z >>= s; e <<= s; k |= s;
   return min( rmq[x+n*k], rmq[y+n*k-e+1] );
}
// outer LCP computation: O(m - o)
int computeLCP(char *t, int n, char *p, int m, int o, int k) {
   int i = o;
   for (; i < m && k+i < n && p[i] == t[k+i]; ++i);
   return i;
}
// Mamber-Myers's O(m + log n) string matching with LCP/RMQ
#define COMP(h, k) (h == m || (k+h<n && p[h]<t[k+h]))
int find(char *t, int n, char *p, int m, int *sa, int *rmq) {
   int l = 0, lh = 0, r = n, rh = computeLCP(t,n+1,p,m,0,sa[n]);
   if (!COMP(rh, sa[r])) return -1;
   for (int k = (l+r)/2; l+1 < r; k = (l+r)/2) {
      int A = minimum(l+1, k, rmq, n+1), B = minimum(k+1, r, rmq, n+1);
      if (A >= B) {
         if (lh < A) l = k;
         else if (lh > A) r = k, rh = A;
         else {
            int i = computeLCP(t, n+1, p, m, A, sa[k]);
            if (COMP(i, sa[k])) r = k, rh = i; else l = k, lh = i;
         }
      } else {
         if (rh < B) r = k;
         else if (rh > B) l = k, lh = B;
         else {
            int i = computeLCP(t, n+1, p, m, B, sa[k]);
            if (COMP(i, sa[k])) r = k, rh = i; else l = k, lh = i;
         }
      }
   }
   return rh == m ? sa[r] : -1;
}
```

3.5. **Regex JAVA.**

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/*
      [abc] a, b, or c (simple class)
      [^abc] Any character except a, b, or c (negation)
      [a-zA-Z] a through z, or A through Z, inclusive (range)
      [a-d[m-p]] a through d, or m through p: [a-dm-p] (union)
      [a-z&&[def]] d, e, or f (intersection)
      [a-z&&[^bc]] a through z, except for b and c: [ad-z] (subtraction)
      [a-z&&[^m-p]] a through z, and not m through p: [a-lq-z] (subtraction)
      . Any character (may or may not match line terminators)
      \d A digit: [0-9]
      \D A non-digit: [^0-9]
      \s A whitespace character: [ \t\n\x0B\f\r]
      \S A non-whitespace character: [^\s]
      \w A word character: [a-zA-Z_0-9]
      \W A non-word character: [^\w]
```

```
      ^ The beginning of a line
      $ The end of a line
      \b A word boundary
      \B A non-word boundary
      \A The beginning of the input
      \G The end of the previous match
      \Z The end of the input but for the final terminator, if any
      \z The end of the input
*/

public class Regex {

      public static void main(String[] args) throws Exception {
            BufferedReader stdin;
      stdin = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Digite a regex: ");
            String regex = stdin.readLine();
            System.out.println("Digite a palavra a ser verificada na regex: ");
            String word = stdin.readLine();
            Pattern pattern = Pattern.compile(regex);
            Matcher matcher = pattern.matcher(word);
          if(matcher.find()){
            System.out.println("Achou!");
          }else{
            System.out.println("Nao achou!");
          }
        }

}
```

4. Data structures

4.1. Union find.

```
struct UnionFind {
   vector<int> data;
   UnionFind(int size) : data(size, -1) { }
   bool unionSet(int x, int y) {
      x = root(x); y = root(y);
      if (x != y) {
         if (data[y] < data[x]) swap(x, y);
         data[x] += data[y]; data[y] = x;
      }
      return x != y;
   }
   bool findSet(int x, int y) {
      return root(x) == root(y);
   }
   int root(int x) {
      return data[x] < 0 ? x : data[x] = root(data[x]);
   }
   int size(int x) {
      return -data[root(x)];
   }
};
```

4.2. Interval Tree.

```
typedef int position;
typedef int contents;
struct interval {
   position low, high;
   interval(position low, position high) :
      low(low), high(high) { }
};

struct interval_tree {
   vector<position> pos;
   struct node {
      vector<contents> values;
      position B, E, M;
      node *left, *right;
   } *root;

   template <class IN>
```

```
    interval_tree(IN begin, IN end) : pos(begin, end) {
       root = build_tree(0, pos.size()-1);
    }
 ~interval_tree() { release(root); }

 node *build_tree(int i, int j) {
    int m = (i+j)/2;
    node *t = new node;
    t->B = pos[i]; t->E = pos[j]; t->M = pos[m];
    t->left = (i+1 < j ? build_tree(i, m) : NULL);
    t->right = (i+1 < j ? build_tree(m, j) : NULL);
    return t;
 }

 // insert
 void insert(const interval& I, contents c) { insert(root, I, c); }

 void insert(node *v, const interval& I, contents c) {
    if (I.low <= v->B && v->E <= I.high) {
       v->values.push_back( c );
    } else {
       if (I.low < v->M) insert(v->left , I, c);
       if (I.high > v->M) insert(v->right, I, c);
    }
  }

  // query
  template <class OUT>
    void query(position p, OUT out) { query(root, p, out); }

  template <class OUT>
    void query(node *t, position p, OUT out) {
       if (!t || p < t->B || p >= t->E) return;
       copy(t->values.begin(), t->values.end(), out);
       if (p < t->M) query(t->left, p, out);
       else query(t->right, p, out);
    }

  // release
  void release(node *t) {
    if (t->left) release(t->left);
    if (t->right) release(t->right);
    delete t;
  }
};
```

4.3. **Range Minimum Query.**

```
// Given an array A[0, N-1] find the position of
// the element with the minimum value between two given indices.
// RMQ(2,7)=3
// a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9]
// 2 | 4 | 3 | 1 | 6 | 7 | 8 | 9 | 1 | 7
int *buildRMQ(int *a, int n) {
 int logn = 1;
 for (int k = 1; k < n; k *= 2) ++logn;
 int *r = new int[n * logn];
 int *b = r; copy(a, a+n, b);
 for (int k = 1; k < n; k *= 2) {
  copy(b, b+n, b+n); b += n;
  REP(i, n-k) b[i] = min(b[i], b[i+k]);
 }
 return r;
}

int minimum(int x, int y, int *rmq, int n) {
 int z = y - x, k = 0, e = 1, s; // y - x >= e = 2^k
 s = ( (z & 0xffff0000) != 0 ) << 4; z >>= s; e <<= s; k |= s;
 s = ( (z & 0x0000ff00) != 0 ) << 3; z >>= s; e <<= s; k |= s;
 s = ( (z & 0x000000f0) != 0 ) << 2; z >>= s; e <<= s; k |= s;
 s = ( (z & 0x0000000c) != 0 ) << 1; z >>= s; e <<= s; k |= s;
 s = ( (z & 0x00000002) != 0 ) << 0; z >>= s; e <<= s; k |= s;
 return min( rmq[x+n*k], rmq[y+n*k-e+1] );
}
```

4.4. **Range Maximum Query.**

```
int r[50010];
```

```
int mm[50010][18]; // or n and log(n) +1
```

```
void construct() {
   int i,j,b;
   for(i=0;i<n;i++) mm[i][0]=r[i];
   for(i=1;i<18;i++)
   {
      for(j=0; (j+(1<<i)-1)<n; j+=(1<<i))
         mm[j][i]=max(mm[j][i-1], mm[j+(1<<(i-1))][i-1]);
   }
}
```

```
int getmax(int a, int b) {
   if(a>b) return -1;
   for(int i=17; i>=0; i--)
   {
      if((a%(1<<i))==0 && (a+(1<<i)-1)<=b)
         return max(mm[a][i], getmax(a+(1<<i), b));
   }
}
```

4.5. **Trie.**

```
struct Trie {
   int value;
   Trie *next[0x100];
   Trie() { fill(next, next+0x100, (Trie*)0); }
};

Trie *find(char *t, Trie *r) {
   for (int i = 0; t[i]; ++i) {
      char c = t[i];
      if (!r->next[c]) r->next[c] = new Trie;
      r = r->next[c];
   }
   return r;
}
```

5. Dynamic Programming

5.1. **Edit distance.**

```
typedef unsigned int ui;

template<class T> ui edit_distance(const T& s1, const T& s2){
   const size_t len1 = s1.size(), len2 = s2.size();
   vector<vector<ui> > d(len1 + 1, vector<ui>(len2 + 1));

   d[0][0] = 0;
   for(ui i = 1; i <= len1; ++i) d[i][0] = i;
   for(ui i = 1; i <= len2; ++i) d[0][i] = i;

   for(ui i = 1; i <= len1; ++i)
      for(ui j = 1; j <= len2; ++j)
         d[i][j] = min( min(d[i-1][j]+1,d[i][j-1]+1),
               d[i-1][j-1]+(s1[i-1]==s2[j-1] ? 0 : 1) );
   return d[len1][len2];
}
```

5.2. **LCS.**

```
// longest common sequence
vector<int> lcs(const vector<int>& a, const vector<int>& b) {
   const int n = a.size(), m = b.size();
   vector< vector<int> > X(n+1, vector<int>(m+1));
   vector< vector<int> > Y(n+1, vector<int>(m+1));
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
         if (a[i] == b[j]) {
            X[i+1][j+1] = X[i][j] + 1;
            Y[i+1][j+1] = 0;
         } else if (X[i+1][j] < X[i][j+1]) {
            X[i+1][j+1] = X[i][j+1];
```

```
            Y[i+1][j+1] = +1;
         } else {
            X[i+1][j+1] = X[i+1][j];
            Y[i+1][j+1] = -1;
         }
      }
   }
   vector<int> c;
   for (int i = n, j = m; i > 0 && j > 0; ) {
      if (Y[i][j] > 0) --i;
      else if (Y[i][j] < 0) --j;
      else { c.push_back(a[i-1]); --i; --j; }
   }
   reverse(c.begin(), c.end());
   return c;
}
```

5.3. LIS.

```
// Longest increasing subsequence
vector<int> lis(vector<int>& seq) {
   int smallest_end[seq.size()+1], prev[seq.size()];
   smallest_end[1] = seq[0];

   int sz = 1;
   for(int i = 1; i < seq.size(); ++i) {
      int lo = 0, hi = sz;
      while(lo < hi) {
         int mid = (lo + hi + 1)/2;
         if(seq[smallest_end[mid]] <= seq[i])
            lo = mid;
         else
            hi = mid - 1;
      }
      prev[i] = smallest_end[lo];
      if(lo == sz)
         smallest_end[++sz] = i;
      else if(seq[i] < seq[smallest_end[lo+1]])
         smallest_end[lo+1] = i;
   }

   vector<int> ret;
   for(int cur = smallest_end[sz]; sz > 0; cur = prev[cur], --sz)
      ret.push_back(seq[cur]);
   reverse(ret.begin(), ret.end());

   return ret;
}
```

5.4. Matrix Chain.

```
// matrix chain
int matrix_chain(vector<int>& p, vector< vector<int> >& s) {
   const int n = p.size()-1;
   vector< vector<int> > X(n, vector<int>(n, inf));
   s.resize(n, vector<int>(n));
   for (int i = 0; i < n; ++i) X[i][i] = 0;
   for (int w = 1; w < n; ++w)
      for (int i = 0, j; j = i+w, j < n; ++i)
         for (int k = i, f; k < j; ++k) {
            int f = p[i]*p[k+1]*p[j+1];
            if (X[i][k] + X[k+1][j] + f < X[i][j]) {
               X[i][j] = X[i][k] + X[k+1][j] + f;
               s[i][j] = k;
            }
         }
   return X[0][n-1];
}
```

5.5. Counting change.

```
#define MAX 100

int s, n, coins[MAX], sol[MAX];
```

```
int solve(){
   REP(i,s+1) sol[i]=INF;
   sol[0] = 0;
   for(int i=1; i<=s; i++) {
      REP(j,n){
         if(coins[j]<=i && sol[i-coins[j]]+1 < sol[i]) {
            sol[i] = sol[i-coins[j]]+1;
         }
      }
   }
   REP(i,s+1){
      printf("%d ",sol[i]);
   puts("");
   return sol[s];
}

void read() {
   scanf("%d %d",&s,&n);
   REP(i,n) scanf("%d",&coins[i]);
   printf("%d",solve());
}
```

5.6. **Knapsack.**

```
#define MAXN 2000
#define MAXK 1000

int matrix[MAXN][MAXK+1];
int cost[MAXN];
int profit[MAXN];

int knapsack(int n,int k) {
   int c;
   REP(i,n) memset(matrix[i],0,sizeof(matrix[i]));
   for(int i=cost[0]; i<=k; i++)
      matrix[0][i] = profit[0];
   for(int i=1;i<n;i++)
      REP(j,k+1){
         if(j-cost[i] >= 0)
            c = matrix[i-1][j-cost[i]]+profit[i];
         else
            c = 0;
         if(c < matrix[i-1][j])
            c = matrix[i-1][j];
         matrix[i][j] = c;
      }
   return matrix[n-1][k];
}
```

6. Math

6.1. **GCD.**

```
// Greatest common divisor
// if gcd(a,b) then a and b are coprime
inline int gcd(int a, int b){
   if(a<0) return gcd(-a,b);
   if(b<0) return gcd(a,-b);
   return (b==0) ? a : gcd(b,a%b);
}
```

6.2. **LCM.**

```
// Least common multiple
// requisits: gcd
inline int lcm(int a, int b){
   if(a<0) return lcm(-a,b);
   if(b<0) return lcm(a,-b);
   return a*(b/gcd(a,b));
}
```

6.3. EXTGCD.

```
// Extended Euclidean algorithm
// Find x and y that solve:
// a*x + b*y = gcd(a,b)
int extgcd(int a, int b, int &x, int &y) {
   int g = a; x = 1; y = 0;
   if(b!=0) g=extgcd(b,a%b,y,x), y-=(a/b)*x;
   return g;
}
```

6.4. Binomial.

```
// Binomial - Pascal Triangle
#define DIM 80
unsigned long long C[DIM+1][DIM+1];

void run(){
   unsigned long long n,k;

   for(n=0; n<=DIM; ++n) {
      C[n][0] = C[n][n] = 1;
      for(k=1; k<n; ++k)
         C[n][k] = C[n-1][k-1] + C[n-1][k];
   }
}
```

6.5. Factorize.

```
// Factorize an integer N
// factorize(60) returns => (2,2), (3,1), (5,1)
template<class T> inline vector<pair<T,int> > factorize(T n){
   vector<pair<T,int> > R;
   for(T i=2;n>1;){
      if(n%i==0){
         int C=0;
         for(;n%i==0;C++,n/=i);
         R.push_back(make_pair(i,C));
      }
      i++;
      if(i>n/i) i=n;
   }
   if(n>1) R.push_back(make_pair(n,1));
   return R;
}
```

6.6. Fractional Library.

```
// Fractional library
struct frac {
   long long num, den;

   frac() : num(0), den(1) { };
   frac(long long num, long long den) { set_val(num, den); }
   frac(long long num) : num(num), den(1) { };

   void set_val(long long _num, long long _den) {
      num = _num/__gcd(_num, _den);
      den = _den/__gcd(_num, _den);
      if(den < 0) { num *= -1; den *= -1; }
   }

   void operator*=(frac f) { set_val(num * f.num, den * f.den); }
   void operator+=(frac f) { set_val(num * f.den + f.num * den, den * f.den); }
   void operator-=(frac f) { set_val(num * f.den - f.num * den, den * f.den); }
   void operator/=(frac f) { set_val(num * f.den, den * f.num); }
};

bool operator<(frac a, frac b) {
   if((a.den < 0) ^ (b.den < 0)) return a.num * b.den > b.num * a.den;
   return a.num * b.den < b.num * a.den;
}
```

```
std::ostream& operator<<(std::ostream& o, const frac f) {
   o << f.num << "/" << f.den;
   return o;
}

bool operator==(frac a, frac b) { return a.num * b.den == b.num * a.den; }
bool operator!=(frac a, frac b) { return !(a == b); }
bool operator<=(frac a, frac b) { return (a == b) || (a < b); }
bool operator>=(frac a, frac b) { return !(a < b); }
bool operator>(frac a, frac b) { return !(a <= b); }
frac operator/(frac a, frac b) { frac ret = a; ret /= b; return ret; }
frac operator*(frac a, frac b) { frac ret = a; ret *= b; return ret; }
frac operator+(frac a, frac b) { frac ret = a; ret += b; return ret; }
frac operator-(frac a, frac b) { frac ret = a; ret -= b; return ret; }
frac operator-(frac f) { return 0 - f; }
```

6.7. Pollard rho.

```
// Pollard
// input: integer to be factored
// output: a non-trivial factor of n
long long pollard_r, pollard_n;
inline long long f(long long val) { return (val*val + pollard_r) % pollard_n; }
inline long long myabs(long long a) { return a >= 0 ? a : -a; }

long long pollard(long long n) {
   srand(unsigned(time(0)));
   pollard_n = n;

   long long d = 1;
   do {
      d = 1;
      pollard_r = rand() % n;

      long long x = 2, y = 2;
      while(d == 1)
         x = f(x), y = f(f(y)), d = __gcd(myabs(x-y), n);
   } while(d == n);

   return d;
}
```

6.8. Miller Rabin.

```
// Prime numbers verify
int fastpow(int base, int d, int n) {
   int ret = 1;
   for(long long pow = base; d > 0; d >>= 1, pow = (pow * pow) % n)
      if(d & 1)
         ret = (ret * pow) % n;
   return ret;
}

bool miller_rabin(int n, int base) {
   if(n <= 1) return false;
   if(n % 2 == 0) return n == 2;

   int s = 0, d = n - 1;
   while(d % 2 == 0) d /= 2, ++s;

   int base_d = fastpow(base, d, n);
   if(base_d == 1) return true;
   int base_2r = base_d;

   for(int i = 0; i < s; ++i) {
      if(base_2r == 1) return false;
      if(base_2r == n - 1) return true;
      base_2r = (long long)base_2r * base_2r % n;
   }

   return false;
}

bool isprime(int n) {
   if(n == 2 || n == 7 || n == 61) return true;
   return miller_rabin(n, 2) && miller_rabin(n, 7) && miller_rabin(n, 61);
}
```

6.9. **Sieve.**

```
// Sieve primes
const unsigned MAX = 1000000020/60, MAX_S = sqrt(MAX/60);
unsigned w[16] = {1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59};
unsigned short notprime[MAX];
vector<int> primes;

void sieve() {
   unsigned mod[16][16], di[16][16], num;
   for(int i = 0; i < 16; i++)
      for(int j = 0; j < 16; j++) {
         di[i][j] = (w[i]*w[j])/60;
         mod[i][j] = lower_bound(w, w + 16, (w[i]*w[j])%60) - w;
      }

   primes.push_back(2); primes.push_back(3); primes.push_back(5);
   memset(notprime, 0, sizeof notprime);
   for(unsigned i = 0; i < MAX; i++)
      for(int j = (i==0); j < 16; j++) {
         if(notprime[i] & (1<<j)) continue;
         primes.push_back(num = 60*i + w[j]);

         if(i > MAX_S) continue;
         for(unsigned k = i, done = false; !done; k++)
            for(int l = 0; l < 16 && !done; l++) {
               unsigned mult = k*num + i*w[l] + di[j][l];
               if(mult >= MAX) done = true;
               else notprime[mult] |= 1<<mod[j][l];
            }
      }
}
```

6.10. **Carmicheal Lambda function.**

```
// Carmichael Lambda
// requisits: lcm
// Given n, returns a^m == 1(mod n)
// Lamda(p^k) = p^{k-1} (p - 1) (p >= 3 or k <= 2)
// Lamda(2^k) = 2^{k-2} (k > 2)
// Lamda(p1^k1 ... pn^kn) = lcm(Lamda(p1^k1), ..., Lamda(pn^kn))
int carmichaelLambda(int n){
   int ans = 1;
   if(n%8==0) n/=2;
   for(int d=2;d<=n;++d){
      if(n%d==0){
         int y=d-1;
         n/=d;
         while(n%d==0) n/=d, y*=d;
         ans = lcm(ans,y);
      }
   }
   return ans;
}
```

6.11. **Euler function.**

```
// Euler Phi
// eulerPhi(n) returns the number
// of co-primes smaller than n
int eulerPhi(int n){
   if(n==0) return 0;
   int ans=n;
   for(int x=2;x*x<=n;++x){
      if(n%x==0){
         ans-=ans/x;
         while(n%x==0) n/=x;
      }
   }
   if(n>1) ans-=ans/n;
   return ans;
}
```

6.12. **Floyd cycle detection.**

```
// Floyd cycle detection
pair<int, int> floyd(int x0) {
   int t = f(x0), h = f(f(x0)), start = 0, length = 1;
   while(t != h)
      t = f(t), h = f(f(h));

   h = t; t = x0;
   while(t != h)
      t = f(t), h = f(h), ++start;

   h = f(t);
   while(t != h)
      h = f(h), ++length;

   return make_pair(start, length);
}
```

6.13. **invMod.**

```
// invMod
// requisits: extgcd
// x a == 1 (mod n
int invMod(Int a, Int m) {
   Int x, y;
   if (extgcd(a, m, x, y) == 1) return (x + m) % m;
   else return 0; // unsolvable
}
```

6.14. **sqrtMod.**

```
// Quadratic residue
// requisits: invMod, powMod and jacobi
// x^2 == a (mod p)
int sqrtMod(int n, int p) {
   int S, Q, W, i, m = invMod(n, p);
   for(Q=p-1,S=0;Q%2==0;Q/=2,++S);
   do{ W=rand()%p; } while(W== 0||jacobi(W,p)!=-1);
   for(int R = powMod(n,(Q+1)/2,p), V=powMod(W,Q,p);;){
      int z=R*R*m%p;
      for(i=0;i<S&&z%p!=1;z*=z,++i);
      if(i==0) return R;
      R=(R*powMod(V,1<<(S-i-1),p))%p;
   }
}
```

6.15. **powMod.**

```
// powMod
// x^k (mod n)
int powMod(int x, int k, int m){
   if(k==0) return 1;
   if(k%2==0) return powMod(x*x%m,k/2,m);
   else return x*powMod(x,k-1,m)%m;
}
```

6.16. **Jacobi function.**

```
// Jacobi Symbol (m/n),
// For any m integer and n positive odd integer:
// (m/n) = (m/p_1)^e_1 * ... * (m/p_k)^e_k
// where n = p_1^e_1 * ... * p_k^e_k
// (m/n) returns:
// * 0 if a == 0(mod p)
```

```
// * 1 if a != 0(mod p) and for some integer x, a == x^2 (mod p)
// * -1 if there is no such x
#define NEGPOW(e) ((e) % 2 ? -1 : 1)
int jacobi(int a, int m){
   if(a==0) return m == 1 ? 1 : 0;
   if(a%2) return NEGPOW((a-1)*(m-1)/4)*jacobi(m%a, a);
   else return NEGPOW((m*m-1)/8)*jacobi(a/2, m);
}
```

6.17. Möbius function.

```
// Mobius
// - mobiusMu(n) = 1 if n is a square-free positive integer
// with an even number of prime factors.
// - mobiusMu(n) = -1 if n is square-free positive integer
// with an odd number of prime factors.
// - mobiusMu(n) = 0 if n is not square-free.
// square-free: integer is one divisible by no perfect square, except 1.
// example: 10 is square-free but 18 is not, as it is divisible by 9 = 3^2
int N = 20000000000;
int mobiusMu(int n){
   static int lookup = 0, p[N], f[N];
   if(!lookup){
      REP(i,N) p[i]=1, f[i]=1;
      for(int i=2;i<N;++i) {
         if(p[i]){
            f[i]=-1;
            for(int j=i+i;j<N;j+=i){
               p[j]=0;
               f[j]*=(j%(i*i)==0) ? 0 : -1;
            }
         }
      }
      lookup=1;
   }
   return f[n];
}
```

6.18. Romberg.

```
// Romberg
// Assume F' = f
// input: interval [a,b] and a function f
// ouput: F(b)-F(a)
long double romberg(int a, int b, double(*func)(double)) {
   long double approx[2][50];
   long double *cur=approx[1], *prev=approx[0];

   prev[0] = 1/2.0 * (b-a) * (func(a) + func(b));
   for(int it = 1; it < 25; ++it, swap(cur, prev)) {
      if(it > 1 && cmp(prev[it-1], prev[it-2]) == 0)
         return prev[it-1];
      cur[0] = 1/2.0 * prev[0];
      long double div = (b-a)/pow(2, it);
      for(long double sample = a + div; sample < b; sample += 2 * div)
         cur[0] += div * func(a + sample);

      for(int j = 1; j <= it; ++j)
         cur[j] = cur[j-1] + 1/(pow(4, it) - 1)*(cur[j-1] + prev[j-1]);
   }

   return prev[24];
}
```

6.19. Polynomials library.

```
// Polynomial library
typedef complex<double> cdouble;
int cmp(cdouble x, cdouble y = 0) {
   return cmp(abs(x), abs(y));
}
const int TAM = 200;
struct poly {
   cdouble poly[TAM]; int n;
```

```
   poly(int n = 0): n(n) { memset(p, 0, sizeof(p)); }
   cdouble& operator [](int i) { return p[i]; }
   poly operator ~() {
      poly r(n-1);
      for (int i = 1; i <= n; i++)
         r[i-1] = p[i] * cdouble(i);
      return r;
   }
   pair<poly, cdouble> ruffini(cdouble z) {
      if (n == 0) return make_pair(poly(), 0);
      poly r(n-1);
      for (int i = n; i > 0; i--) r[i-1] = r[i] * z + p[i];
      return make_pair(r, r[0] * z + p[0]);
   }
   cdouble operator ()(cdouble z) { return ruffini(z).second; }
   cdouble find_one_root(cdouble x) {
      poly p0 = *this, p1 = ~p0, p2 = ~p1;
      int m = 1000;
      while (m--) {
         cdouble y0 = p0(x);
         if (cmp(y0) == 0) break;
         cdouble G = p1(x) / y0;
         cdouble H = G * G - p2(x) - y0;
         cdouble R = sqrt(cdouble(n-1) * (H * cdouble(n) - G * G));
         cdouble D1 = G + R, D2 = G - R;
         cdouble a = cdouble(n) / (cmp(D1, D2) > 0 ? D1 : D2);
         x -= a;
         if (cmp(a) == 0) break;
      }
      return x;
   }
   vector<cdouble> roots() {
      poly q = *this;
      vector<cdouble> r;
      while (q.n > 1) {
         cdouble z(rand() / double(RAND_MAX), rand() / double(RAND_MAX));
         z = q.find_one_root(z); z = find_one_root(z);
         q = q.ruffini(z).first;
         r.push_back(z);
      }
      return r;
   }
};
```

6.20. **Simplex.**

```
typedef long double T;
typedef vector<T> VT;
typedef vector<int> VI;
vector<VT> A;
VT b,c,res;
VI kt,N;
int m;
inline void pivot(int k,int l,int e){
   int x=kt[l]; T p=A[l][e];
   REP(i,k) A[l][i]/=p; b[l]/=p; N[e]=0;
   REP(i,m) if (i!=l) b[i]-=A[i][e]*b[l],A[i][x]=A[i][e]*-A[l][x];
   REP(j,k) if (N[j]){
      c[j]-=c[e]*A[l][j];
      REP(i,m) if (i!=l) A[i][j]-=A[i][e]*A[l][j];
   }
   kt[l]=e; N[x]=1; c[x]=c[e]*-A[l][x];
}

VT doit(int k){
   VT res; T best;
   while (1){
      int e=-1,l=-1; REP(i,k) if (N[i] && c[i]>EPS) {e=i; break;}
      if (e==-1) break;
      REP(i,m) if (A[i][e]>EPS && (l==-1 || best>b[i]/A[i][e]))
         best=b[ l=i ]/A[i][e];
      if (l==-1) /*ilimitado*/ return VT();
      pivot(k,l,e);
   }
   res.resize(k,0); REP(i,m) res[kt[i]]=b[i];
   return res;
}

VT simplex(vector<VT> &AA,VT &bb,VT &cc){
   int n=AA[0].size(),k;
   m=AA.size(); k=n+m+1; kt.resize(m); b=bb; c=cc; c.resize(n+m);
   A=AA; REP(i,m){ A[i].resize(k); A[i][n+i]=1; A[i][k-1]=-1; kt[i]=n+i;}
   N=VI(k,1); REP(i,m) N[kt[i]]=0;
   int pos=min_element(ALL(b))-b.begin();
   if (b[pos]<-EPS){
      c=VT(k,0); c[k-1]=-1; pivot(k,pos,k-1); res=doit(k);
```

```
      if (res[k-1]>EPS) /*impossivel*/ return VT();
      REP(i,m) if (kt[i]==k-1)
         REP(j,k-1) if (N[j] && (A[i][j]<-EPS || EPS<A[i][j])){
            pivot(k,i,j); break;
         }
      c=cc; c.resize(k,0); REP(i,m) REP(j,k) if (N[j]) c[j]-=c[kt[i]]*A[i][j];
   }
   res=doit(k-1); if (!res.empty()) res.resize(n);
   return res;
}
```

7. Graph

7.1. Base element.

7.1.1. *Global definitions.*

```
typedef int Weight;

struct Edge {
   int src, dst;
   Weight weight;
   Edge(int src, int dst, Weight weight) :
       src(src), dst(dst), weight(weight) { }
};

bool operator < (const Edge &e, const Edge &f) {
   return e.weight != f.weight ? e.weight > f.weight :
        e.src != f.src ? e.src < f.src : e.dst < f.dst;
}

typedef vector<Edge> Edges;
typedef vector<Edges> Graph;
typedef vector<Weight> Array;
typedef vector<Array> Matrix;
```

7.2. Connected component.

7.2.1. *Joint point, connected component decomposition double peak.*

```
struct UndirectionalCompare {
   bool operator() (const Edge& e, const Edge& f) const {
      if (min(e.src,e.dst) != min(f.src,f.dst))
         return min(e.src,e.dst) < min(f.src,f.dst);
      return max(e.src,e.dst) < max(f.src,f.dst);
   }
};
typedef set<Edge, UndirectionalCompare> Edgeset;
void visit(const Graph &g, int v, int u,
      vector<int>& art, vector<Edgeset>& bcomp,
      stack<Edge>& S, vector<int>& num, vector<int>& low, int& time) {
   low[v] = num[v] = ++time;
   EACH(e, g[v]) {
      int w = e->dst;
      if (num[w] < num[v]) S.push(*e); // for bcomps
      if (num[w] == 0) {
         visit(g, w, v, art, bcomp, S, num, low, time);
         low[v] = min(low[v], low[w]);
         if ((num[v] == 1 && num[w] != 2) || // for arts
               (num[v] != 1 && low[w] >= num[v])) art.push_back(v);
         if (low[w] >= num[v]) { // for bcomps
            bcomp.push_back(Edgeset());
            while (1) {
               Edge f = S.top(); S.pop();
               bcomp.back().insert(f);
               if (f.src == v && f.dst == w) break;
            }
         }
      } else low[v] = min(low[v], num[w]);
   }
}
void articulationPoint(const Graph& g,
      vector<int>& art, vector<Edgeset>& bcomp) {
   const int n = g.size();
```

```
   vector<int> low(n), num(n);
   stack<Edge> S;
   REP(u, n) if (num[u] == 0) {
```

```
      int time = 0;
      visit(g, u, -1, art, bcomp, S, num, low, time);
   }
}
```

7.2.2. *Bridge, a double-edge connected component decomposition.*

```
void visit(const Graph & g, int v, int u,
      Edges& brdg, vector< vector<int> >& tecomp,
      stack<int>& roots, stack<int>& S, vector<bool>& inS,
      vector<int>& num, int& time) {
   num[v] = ++time;
   S.push(v); inS[v] = true;
   roots.push(v);
   EACH(e, g[v]) {
      int w = e->dst;
      if (num[w] == 0)
         visit(g, w, v, brdg, tecomp, roots, S, inS, num, time);
      else if (u != w && inS[w])
         while (num[roots.top()] > num[w]) roots.pop();
   }
   if (v == roots.top()) {
      brdg.push_back(Edge(u, v));
      tecomp.push_back(vector<int>());
      while (1) {
         int w = S.top(); S.pop(); inS[w] = false;
```

```
         tecomp.back().push_back(w);
         if (v == w) break;
      }
      roots.pop();
   }
}

void bridge(const Graph& g, Edges& brdg, vector< vector<int> >& tecomp) {
   const int n = g.size();
   vector<int> num(n);
   vector<bool> inS(n);
   stack<int> roots, S;
   int time = 0;
   REP(u, n) if (num[u] == 0) {
      visit(g, u, n, brdg, tecomp, roots, S, inS, num, time);
      brdg.pop_back();
   }
}
```

7.2.3. *Strongly connected component decomposition.*

```
void visit(const Graph &g, int v, vector< vector<int> >& scc,
      stack<int> &S, vector<bool> &inS,
      vector<int> &low, vector<int> &num, int& time) {
   low[v] = num[v] = ++time;
   S.push(v); inS[v] = true;
   EACH(e, g[v]) {
      int w = e->dst;
      if (num[w] == 0) {
         visit(g, w, scc, S, inS, low, num, time);
         low[v] = min(low[v], low[w]);
      } else if (inS[w])
         low[v] = min(low[v], num[w]);
   }
   if (low[v] == num[v]) {
```

```
      scc.push_back(vector<int>());
      while (1) {
         int w = S.top(); S.pop(); inS[w] = false;
         scc.back().push_back(w);
         if (v == w) break;
      }
   }
}

void stronglyConnectedComponents(const Graph& g,
      vector< vector<int> >& scc) {
   const int n = g.size();
   vector<int> num(n), low(n);
   stack<int> S;
```

```
   vector<bool> inS(n);
   int time = 0;
   REP(u, n) if (num[u] == 0)
      visit(g, u, scc, S, inS, low, num, time);
}
```

7.3. Shortest path.

7.3.1. *Single-source shortest path (Dijkstra).*

```
void shortestPath(const Graph &g,int s,vector<Weight> &dist,vector<int> &prev){
   int n = g.size();
   dist.assign(n, INF); dist[s] = 0;
   prev.assign(n, -1);
   priority_queue<Edge> Q;
   for (Q.push(Edge(-2, s, 0)); !Q.empty(); ) {
      Edge e = Q.front(); Q.pop();
      if (prev[e.dst] != -1) continue;
      prev[e.dst] = e.src;
      EACH(f,g[e.dst]) {
         if (dist[f->dst] > e.weight+f->weight) {
            dist[f->dst] = e.weight+f->weight;
            Q.push(Edge(f->src, f->dst, e.weight+f->weight));
         }
      }
   }
}

vector<int> buildPath(const vector<int> &prev, int t) {
   vector<int> path;
   for (int u = t; u >= 0; u = prev[u])
      path.push_back(u);
   reverse(path.begin(), path.end());
   return path;
}
```

7.3.2. *Single-source shortest paths (Bellman-Ford).*

```
bool shortestPath(const Graph g,int s,vector<Weight> &dist,vector<int> &prev){
   int n = g.size();
   dist.assign(n, INF+INF); dist[s] = 0;
   prev.assign(n, -2);
   bool negative_cycle = false;
   REP(k, n) REP(i, n) EACH(e,g[i]) {
      if (dist[e->dst] > dist[e->src] + e->weight) {
         dist[e->dst] = dist[e->src] + e->weight;
         prev[e->dst] = e->src;
         if (k == n-1) {
            dist[e->dst] = -INF;
            negative_cycle = true;
         }
      }
   }
   return !negative_cycle;
}

vector<int> buildPath(const vector<int> &prev, int t) {
   vector<int> path;
   for (int u = t; u >= 0; u = prev[u])
      path.push_back(u);
   reverse(path.begin(), path.end());
   return path;
}
```

7.3.3. *k-shortest path.*

```
Weight k_shortestPath(const Graph &g, int s, int t, int k) {
   const int n = g.size();
   Graph h(n); // make reverse graph
```

```
   REP(u, n) EACH(e, g[u])
      h[e->dst].push_back(Edge(e->dst,e->src,e->weight));

   vector<Weight> d(n, INF); d[t] = 0; // make potential
   vector<int> p(n, -1); // using backward dijkstra
   priority_queue<Edge> Q; Q.push(Edge(t, t, 0));
   while (!Q.empty()) {
      Edge e = Q.top(); Q.pop();
      if (p[e.dst] >= 0) continue;
      p[e.dst] = e.src;
      EACH(f, h[e.dst]) if (d[f->dst] > e.weight + f->weight) {
         d[f->dst] = e.weight + f->weight;
         Q.push(Edge(f->src, f->dst, e.weight + f->weight));
      }
   }
   int l = 0; // forward dijkstra-like method
   priority_queue<Edge> R; R.push(Edge(-1,s,0));
   while (!R.empty()) {
      Edge e = R.top(); R.pop();
      if (e.dst == t && ++l == k) return e.weight + d[s];
      EACH(f, g[e.dst])
         R.push(Edge(f->src, f->dst, e.weight+f->weight-d[f->src]+d[f->dst]));
   }
   return -1; // not found
}

////////////////////////////////////////////////

// simpler method
Weight k_shortestPath(const Graph &g, int s, int t, int k) {
   const int n = g.size();
   vector<Weight> dist[n];
   priority_queue<Edge> Q; Q.push(Edge(-1, s, 0));
   while (!Q.empty()) {
      Edge e = Q.top(); Q.top();
      if (dist[e.dst].size() >= k) continue;
      dist[e.dst].push_back(e.weight);
      EACH(f, g[e.dst]) Q.push(Edge(f->src, f->dst, f->weight+e.weight));
   }
}
```

7.3.4. *Shortest path between all points for (Johnson).*

```
bool shortestPath(const Graph &g,
      Matrix &dist, vector<vector<int> > &prev) {
   int n = g.size();
   Array h(n+1);
   REP(k,n) REP(i,n) EACH(e,g[i]) {
      if (h[e->dst] > h[e->src] + e->weight) {
         h[e->dst] = h[e->src] + e->weight;
         if (k == n-1) return false; // negative cycle
      }
   }
   dist.assign(n, Array(n, INF));
   prev.assign(n, vector<int>(n, -2));
   REP(s, n) {
      priority_queue<Edge> Q;
      Q.push(Edge(s, s, 0));
      while (!Q.empty()) {
         Edge e = Q.top(); Q.pop();
         if (prev[s][e.dst] != -2) continue;
         prev[s][e.dst] = e.src;
         EACH(f,g[e.dst]) {
            if (dist[s][f->dst] > e.weight + f->weight) {
               dist[s][f->dst] = e.weight + f->weight;
               Q.push(Edge(f->src, f->dst, e.weight + f->weight));
            }
         }
      }
      REP(u, n) dist[s][u] += h[u] - h[s];
   }
}

vector<int> buildPath(const vector< vector<int> >& prev, int s, int t) {
   vector<int> path;
   for (int u = t; u >= 0; u = prev[s][u])
      path.push_back(u);
   reverse(ALL(path));
   return path;
}
```

7.3.5. *Shortest path between all points for (Floyd Warshall).*

```
void shortestPath(const Matrix &g,Matrix &dist,vector<vector<int> > &inter){
   int n = g.size();
   dist = g;
   inter.assign(n, vector<int>(n,-1));
   REP(k, n) REP(i, n) REP(j, n) {
      if (dist[i][j] < dist[i][k] + dist[k][j]) {
         dist[i][j] = dist[i][k] + dist[k][j];
         inter[i][j] = k;
      }
   }
}
```

```
void buildPath(const vector<vector<int> > &inter,int s,int t,vector<int> &path){
   int u = inter[s][t];
   if (u < 0) path.push_back(s);
   else buildPath(inter, s, u, path), buildPath(inter, u, s, path);
}

vector<int> buildPath(const vector< vector<int> > &inter, int s, int t) {
   vector<int> path;
   buildPath(inter, s, t, path);
   path.push_back(t);
   return path;
}
```

7.4. **Spanning Tree.**

7.4.1. *Minimum spanning tree (Prim).*

```
pair<Weight, Edges> minimumSpanningTree(const Graph &g, int r = 0) {
   int n = g.size();
   Edges T;
   Weight total = 0;

   vector<bool> visited(n);
   priority_queue<Edge> Q;
   Q.push( Edge(-1, r, 0) );
   while (!Q.empty()) {
      Edge e = Q.top(); Q.pop();
      if (visited[e.dst]) continue;
      T.push_back(e);
      total += e.weight;
      visited[e.dst] = true;
      EACH(f, g[e.dst]) if (!visited[f->dst]) Q.push(*f);
   }
   return pair<Weight, Edges>(total, T);
}
```

7.4.2. *Minimum spanning forest (Kruskal).*

```
pair<Weight, Edges> minimumSpanningForest(const Graph &g) {
   int n = g.size();
   UnionFind uf(n);
   priority_queue<Edge> Q;
   REP(u, n) EACH(e, g[u]) if (u < e->dst) Q.push(*e);

   Weight total = 0;
   Edges F;
   while (F.size() < n-1 && !Q.empty()) {
      Edge e = Q.top(); Q.pop();
      if (uf.unionSet(e.src, e.dst)) {
         F.push_back(e);
         total += e.weight;
      }
   }
   return pair<Weight, Edges>(total, F);
}
```

7.4.3. *Minimum diameter spanning tree (Cuninghame-Green).*

```
Weight minimumDiameterSpanningTree(const Graph &g) {
   int n = g.size();
   Matrix dist(n, Array(n, INF)); // all-pair shortest
   REP(u, n) dist[u][u] = 0;
   REP(u, n) EACH(e, g[u]) dist[e->src][e->dst] = e->weight;
   REP(k, n) REP(i, n) REP(j, n)
      dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j]);

   Edges E;
   REP(u, n) EACH(e, g[u]) if (e->src < e->dst) E.push_back(*e);
   int m = E.size();

   Weight H = INF;
   vector<Weight> theta(m);
   REP(r, m) {
      int u = E[r].src, v = E[r].dst;
      Weight d = E[r].weight;
      REP(w, n) theta[r] = max(theta[r], min(dist[u][w], dist[v][w]));
      H = min(H, d + 2 * theta[r]);
   }
   Weight value = INF;
   REP(r, m) if (2 * theta[r] <= H) {
      int u = E[r].src, v = E[r].dst;
      Weight d = E[r].weight;
      vector< pair<Weight,Weight> > list;
      REP(w, n) list.push_back( make_pair(dist[u][w], dist[v][w]) );
      sort(ALL(list), greater< pair<Weight,Weight> >());
      int p = 0;
      value = min(value, 2 * list[0].first);
      REP(k, n) if (list[p].second < list[k].second)
         value = min(value, d + list[p].second + list[k].first), p = k;
      value = min(value, 2 * list[p].second);
   }
   return n == 1 ? 0 : value;
}
```

7.4.4. *Minimum spanning directed tree (Chu-Liu/Edmonds).*

```
void visit(Graph &h, int v, int s, int r,
      vector<int> &no, vector< vector<int> > &comp,
      vector<int> &prev, vector< vector<int> > &next, vector<Weight> &mcost,
      vector<int> &mark, Weight &cost, bool &found) {
   const int n = h.size();
   if (mark[v]) {
      vector<int> temp = no;
      found = true;
      do {
         cost += mcost[v];
         v = prev[v];
         if (v != s) {
            while (comp[v].size() > 0) {
               no[comp[v].back()] = s;
               comp[s].push_back(comp[v].back());
               comp[v].pop_back();
            }
         }
      } while (v != s);
      EACH(j,comp[s]) if (*j != r) EACH(e,h[*j])
         if (no[e->src] != s) e->weight -= mcost[ temp[*j] ];
   }
   mark[v] = true;
   EACH(i,next[v]) if (no[*i] != no[v] && prev[no[*i]] == v)
      if (!mark[no[*i]] || *i == s)
         visit(h, *i, s, r, no, comp, prev, next, mcost, mark, cost, found);
}
Weight minimumSpanningArborescence(const Graph &g, int r) {
   const int n = g.size();
   Graph h(n);
   REP(u,n) EACH(e,g[u]) h[e->dst].push_back(*e);

   vector<int> no(n);
   vector< vector<int> > comp(n);
   REP(u, n) comp[u].push_back(no[u] = u);

   for (Weight cost = 0; ;) {
      vector<int> prev(n, -1);
      vector<Weight> mcost(n, INF);

      REP(j,n) if (j != r) EACH(e,g[j])
         if (no[e->src] != no[j])
```

```
            if (e->weight < mcost[ no[j] ])
               mcost[ no[j] ] = e->weight, prev[ no[j] ] = no[e->src];

      vector< vector<int> > next(n);
      REP(u,n) if (prev[u] >= 0)
         next[ prev[u] ].push_back(u);

      bool stop = true;
      vector<int> mark(n);
      REP(u,n) if (u != r && !mark[u] && !comp[u].empty()) {
         bool found = false;
         visit(h, u, u, r, no, comp, prev, next, mcost, mark, cost, found);
         if (found) stop = false;
      }
      if (stop) {
         REP(u,n) if (prev[u] >= 0) cost += mcost[u];
         return cost;
      }
   }
}


//////////////////////////////////////////////////////////////////////

void backward_traverse(int v, int s, int r, matrix &g,
      vector<int> &no, vector< vector<int> > &comp,
      vector<int> &prev, vector<weight> &mcost,
      vector<int> &mark, weight &cost, bool &found) {
   const int n = g.size();
   if (mark[v]) {
      vector<int> temp = no;
      found = true;
      do {
         cost += mcost[v];
         v = prev[v];
         if (v != s) {
            while (comp[v].size() > 0) {
               no[comp[v].back()] = s;
               comp[s].push_back(comp[v].back());
               comp[v].pop_back();
            }
         }
      } while (v != s);
      for (int j = 0; j < n; ++j)
         if (j != r && no[j] == s)
            for (int i = 0; i < n; ++i)
               if (no[i] != s && g[i][j] < inf)
                  g[i][j] -= mcost[ temp[j] ];
      }
      mark[v] = true;
      for (int i = 0; i < n; ++i)
         if (no[i] != no[v] && prev[ no[i] ] == v)
            if (!mark[ no[i] ] || i == s)
               backward_traverse(i, s, r, g,
                     no, comp, prev, mcost, mark, cost, found);
}

weight minimum_spanning_arborescence(int r, matrix &g) {
   const int n = g.size();

   vector<int> no(n);
   vector< vector<int> > comp(n);
   for (int i = 0; i < n; ++i) {
      no[i] = i;
      comp[i].push_back(i);
   }
   weight cost = 0;
   while (1) {
      vector<int> prev(n, -1);
      vector<weight> mcost(n, inf);
      for (int i = 0; i < n; ++i) {
         for (int j = 0; j < n; ++j) {
            if (j == r) continue;
            if (no[i] != no[j] && g[i][j] < inf) {
               if (g[i][j] < mcost[ no[j] ]) {
                  mcost[ no[j] ] = g[i][j];
                  prev[ no[j] ] = no[i];
               }
            }
         }
      }
      bool stop = true;
      vector<int> mark(n);
      for (int i = 0; i < n; ++i) {
         if (i == r || mark[i] || comp[i].size() == 0) continue;
         bool found = false;
         backward_traverse(i, i, r, g,
               no, comp, prev, mcost, mark, cost, found);
         if (found) stop = false;
      }
      if (stop) {
         for (int i = 0; i < n; ++i)
```

```
            if (prev[i] >= 0)
               cost += mcost[i];
         return cost;
```

7.4.5. *Minimum Steiner tree (Dreyfus-Wagner).*

```
weight minimum_steiner_tree(const vector<int>& T, const matrix &g) {
 const int n = g.size();
 const int numT = T.size();
 if (numT <= 1) return 0;

 matrix d(g); // all-pair shortest
 for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
    d[i][j] = min( d[i][j], d[i][k] + d[k][j] );

 weight OPT[(1 << numT)][n];
 for (int S = 0; S < (1 << numT); ++S)
  for (int x = 0; x < n; ++x)
   OPT[S][x] = inf;

 for (int p = 0; p < numT; ++p) // trivial case
  for (int q = 0; q < n; ++q)
   OPT[1 << p][q] = d[T[p]][q];

 for (int S = 1; S < (1 << numT); ++S) { // DP step
  if (!(S & (S-1))) continue;
  for (int p = 0; p < n; ++p)
   for (int E = 0; E < S; ++E)
    if ((E | S) == S)
     OPT[S][p] = min( OPT[S][p], OPT[E][p] + OPT[S-E][p] );
  for (int p = 0; p < n; ++p)
   for (int q = 0; q < n; ++q)
    OPT[S][p] = min( OPT[S][p], OPT[S][q] + d[p][q] );
 }
 weight ans = inf;
 for (int S = 0; S < (1 << numT); ++S)
  for (int q = 0; q < n; ++q)
   ans = min(ans, OPT[S][q] + OPT[((1 << numT)-1)-S][q]);
}

/*
weight ans = inf;
for (int S = 0; S < (1 << numT); ++S) {
 weight sub = 0;
 for (int P = 0; P < 4; ++P) {
  int E = 0, p = -1;
  for (int k = 0; k < 8; k += 2)
   if (((S >> k) & 3) == P)
    E += 3 << k, p = T[k];
  sub += (!!E) * OPT[E][p];
 }
 ans = min(ans, sub);
}
return ans;
*/
```

7.5. **Flow Cut.**

7.5.1. *Maximum flow (Edmonds-Karp).*

```
#define RESIDUE(s,t) (capacity[s][t]-flow[s][t])
Weight maximumFlow(const Graph &g, int s, int t) {
   int n = g.size();
   Matrix flow(n, Array(n)), capacity(n, Array(n));
   REP(u,n) EACH(e,g[u]) capacity[e->src][e->dst] += e->weight;
   Weight total = 0;
   while (1) {
      queue<int> Q; Q.push(s);
      vector<int> prev(n, -1); prev[s] = s;
      while (!Q.empty() && prev[t] < 0) {
         int u = Q.front(); Q.pop();
```

```
         EACH(e,g[u]) if (prev[e->dst] < 0 && RESIDUE(u, e->dst) > 0) {
            prev[e->dst] = u;
            Q.push(e->dst);
         }
      }
      if (prev[t] < 0) return total; // prev[x] == -1 <=> t-side
      Weight inc = INF;
      for (int j = t; prev[j] != j; j = prev[j])
         inc = min(inc, RESIDUE(prev[j], j));
      for (int j = t; prev[j] != j; j = prev[j])
         flow[prev[j]][j] += inc, flow[j][prev[j]] -= inc;
      total += inc;
   }
}
```

7.5.2. *Maximum flow (Dinic).*

```
#define RESIDUE(s,t) (capacity[s][t]-flow[s][t])

Weight augment(const Graph &g, const Matrix &capacity, Matrix &flow,
           const vector<int> &level, vector<bool> &finished,
           int u, int t, Weight cur) {
   if (u == t || cur == 0) return cur;
   if (finished[u]) return 0;
   finished[u] = true;
   EACH(e, g[u]) if (level[e->dst] > level[u]) {
      Weight f = augment(g, capacity, flow, level, finished,
            e->dst, t, min(cur, RESIDUE(u, e->dst)));
      if (f > 0) {
         flow[u][e->dst] += f; flow[e->dst][u] -= f;
         finished[u] = false;
         return f;
      }
   }
   return 0;
}

Weight maximumFlow(const Graph &g, int s, int t) {
   int n = g.size();
   Matrix flow(n, Array(n)), capacity(n, Array(n)); // adj. matrix
   REP(u,n) EACH(e,g[u]) capacity[e->src][e->dst] += e->weight;

   Weight total = 0;
   for (bool cont = true; cont; ) {
      cont = false;
      vector<int> level(n, -1); level[s] = 0; // make layered network
      queue<int> Q; Q.push(s);
      for (int d = n; !Q.empty() && level[Q.front()] < d; ) {
         int u = Q.front(); Q.pop();
         if (u == t) d = level[u];
         EACH(e, g[u]) if (RESIDUE(u,e->dst) > 0 && level[e->dst] == -1)
            Q.push(e->dst), level[e->dst] = level[u] + 1;
      }
      vector<bool> finished(n); // make blocking flows
      for (Weight f = 1; f > 0; ) {
         f = augment(g, capacity, flow, level, finished, s, t, INF);
         if (f == 0) break;
         total += f; cont = true;
      }
   }
   return total;
}
```

7.5.3. *Maximum flow (Goldberg-Tarjan).*

```
#define RESIDUE(s,t) (capacity[s][t]-flow[s][t])

#define GLOBAL_RELABELING() { \
   queue<int> Q; Q.push(t); \
   fill(ALL(d), INF); d[t] = 0; \
   while (!Q.empty()) { \
      int u = Q.front(); Q.pop(); \
      EACH(e, g[u]) if (RESIDUE(e->dst, u) > 0 && d[u] + 1 < d[e->dst]) \
      Q.push(e->dst), d[e->dst] = d[u] + 1; \
   } \
}

#define PUSH(u, v) { \
   Weight delta = min(excess[u], RESIDUE(u, v)); \
```

```
   flow[u][v] += delta; flow[v][w] -= delta; \
   excess[u] -= delta; excess[v] += delta; }

Weight maximumFlow(const Graph &g, int s, int t) {
   int n = g.size(), count = 0;
   Matrix flow(n, Array(n)), capacity(n, Array(n)); // adj. matrix
   REP(u,n) EACH(e,g[u]) capacity[e->src][e->dst] += e->weight;

   vector<Weight> excess(n); excess[s] = INF; // initialize step
   vector<int> d(n);
   GLOBAL_RELABELING();
   vector< queue<int> > B(n); B[ d[s] ].push( s );

   for (int b = d[s]; b >= 0; ) {
      if (B[b].empty()) { --b; continue; }
      int v = B[b].front(); B[b].pop();
      if (excess[v] == 0 || v == t) continue;
      EACH(e, g[v]) {
         int w = e->dst; // e is the current edge of v
         if (RESIDUE(v,w) > 0 && d[v] == d[w] + 1) { // (w,v) is admissible
            PUSH(v, w);
            if (excess[w] > 0 && w != t) B[d[w]].push( w );
         }
      }
      if (excess[v] == 0) continue;
      d[v] = n;
      EACH(e, g[v]) if (RESIDUE(v, e->dst) > 0)
         d[v] = min(d[v], d[e->dst] + 1);
      if (d[v] < n) B[b = d[v]].push(v);

      if (++count % n == 0) GLOBAL_RELABELING(); // !!HEURISTICS
   }
   return excess[t];
}
```

7.5.4. *Undirected graph minimum cut across (Nagamochi-Ibaraki/Stoer-Wagner).*

```
Weight minimumCut(const Graph &g) {
   int n = g.size();
   vector< vector<Weight> > h(n, vector<Weight>(n)); // make adj. matrix
   REP(u,n) EACH(e,g[u]) h[e->src][e->dst] += e->weight;
   vector<int> V(n); REP(u, n) V[u] = u;

   Weight cut = INF;
   for(int m = n; m > 1; m--) {
      vector<Weight> ws(m, 0);
      int u, v;
      Weight w;
      REP(k, m) {
         u = v; v = max_element(ws.begin(), ws.end())-ws.begin();
         w = ws[v]; ws[v] = -1;
         REP(i, m) if (ws[i] >= 0) ws[i] += h[V[v]][V[i]];
      }
      REP(i, m) {
         h[V[i]][V[u]] += h[V[i]][V[v]];
         h[V[u]][V[i]] += h[V[v]][V[i]];
      }
      V.erase(V.begin()+v);
      cut = min(cut, w);
   }
   return cut;
}
```

7.5.5. *Thurs Gomory-Hu.*

```
#define RESIDUE(s,t) (capacity[s][t]-flow[s][t])
Graph cutTree(const Graph &g) {
   int n = g.size();
   Matrix capacity(n, Array(n)), flow(n, Array(n));
   REP(u,n) EACH(e,g[u]) capacity[e->src][e->dst] += e->weight;

   vector<int> p(n), prev;
   vector<Weight> w(n);
   for (int s = 1; s < n; ++s) {
      int t = p[s]; // max-flow(s, t)
      REP(i,n) REP(j,n) flow[i][j] = 0;
      Weight total = 0;
```

```
      while (1) {
         queue<int> Q; Q.push(s);
         prev.assign(n, -1); prev[s] = s;
         while (!Q.empty() && prev[t] < 0) {
            int u = Q.front(); Q.pop();
            EACH(e,g[u]) if (prev[e->dst] < 0 && RESIDUE(u, e->dst) > 0) {
               prev[e->dst] = u;
               Q.push(e->dst);
            }
         }
         if (prev[t] < 0) goto esc;
         Weight inc = INF;
         for (int j = t; prev[j] != j; j = prev[j])
            inc = min(inc, RESIDUE(prev[j], j));
         for (int j = t; prev[j] != j; j = prev[j])
            flow[prev[j]][j] += inc, flow[j][prev[j]] -= inc;
         total += inc;
      }
esc:w[s] = total; // make tree
   REP(u, n) if (u != s && prev[u] != -1 && p[u] == t)
      p[u] = s;
   if (prev[p[t]] != -1)
      p[s] = p[t], p[t] = s, w[s] = w[t], w[t] = total;
   }
   Graph T(n); // (s, p[s]) is a tree edge of weight w[s]
   REP(s, n) if (s != p[s]) {
      T[ s ].push_back( Edge(s, p[s], w[s]) );
      T[p[s]].push_back( Edge(p[s], s, w[s]) );
   }
   return T;
}

// Gomory-Hu tree O(n)
Weight maximumFlow(const Graph &T, int u, int t, int p = -1, Weight w = INF) {
   if (u == t) return w;
   Weight d = INF;
   EACH(e, T[u]) if (e->dst != p)
      d = min(d, maximumFlow(T, e->dst, t, u, min(w, e->weight)));
   return d;
}
```

7.5.6. *Minimum cost flow (Primal Dual).*

```
#define RESIDUE(u,v) (capacity[u][v] - flow[u][v])
#define RCOST(u,v) (cost[u][v] + h[u] - h[v])
pair<Weight, Weight> minimumCostFlow(const Graph &g, int s, int t) {
   const int n = g.size();
   Matrix capacity(n, Array(n)), cost(n, Array(n)), flow(n, Array(n));
   REP(u,n) EACH(e,g[u]) {
      capacity[e->src][e->dst] += e->capacity;
      cost[e->src][e->dst] += e->cost;
   }
   pair<Weight, Weight> total; // (cost, flow)
   vector<Weight> h(n);

   for (Weight F = INF; F > 0; ) { // residual flow
      vector<Weight> d(n, INF); d[s] = 0;
      vector<int> p(n, -1);
      priority_queue<Edge> Q; // "e < f" <=> "e.cost > f.cost"
      for (Q.push(Edge(-2, s)); !Q.empty(); ) {
         Edge e = Q.top(); Q.pop();
         if (p[e.dst] != -1) continue;
         p[e.dst] = e.src;
         EACH(f, g[e.dst]) if (RESIDUE(f->src, f->dst) > 0) {
            if (d[f->dst] > d[f->src] + RCOST(f->src, f->dst)) {
               d[f->dst] = d[f->src] + RCOST(f->src, f->dst);
               Q.push( Edge(f->src, f->dst, 0, d[f->dst]) );
            }
         }
      }
      if (p[t] == -1) break;

      Weight f = F;
      for (int u = t; u != s; u = p[u])
         f = min(f, RESIDUE(p[u], u));
      for (int u = t; u != s; u = p[u]) {
         total.first += f * cost[p[u]][u];
         flow[p[u]][u] += f; flow[u][p[u]] -= f;
      }
      F -= f;
      total.second += f;
      REP(u, n) h[u] += d[u];
   }
   return total;
}
```

7.6. **Matching.**

7.6.1. *The maximum bipartite graph matching.*

```
bool augment(const Graph& g, int u,
      vector<int>& matchTo, vector<bool>& visited) {
   if (u < 0) return true;
   EACH(e, g[u]) if (!visited[e->dst]) {
      visited[e->dst] = true;
      if (augment(g, matchTo[e->dst], matchTo, visited)) {
         matchTo[e->src] = e->dst;
         matchTo[e->dst] = e->src;
         return true;
      }
   }
   return false;
}
```

```
int bipartiteMatching(const Graph& g, int L, Edges& matching) {
   const int n = g.size();
   vector<int> matchTo(n, -1);
   int match = 0;
   REP(u, L) {
      vector<bool> visited(n);
      if (augment(g, u, matchTo, visited)) ++match;
   }
   REP(u, L) if (matchTo[u] >= 0) // make explicit matching
      matching.push_back( Edge(u, matchTo[u]) );
   return match;
}
```

7.6.2. *Minimum weight maximum bipartite graph matching.*

```
#define EVEN(x) (mu[x] == x || (mu[x] != x && phi[mu[x]] != mu[x]))
#define ODD(x) (mu[x] != x && phi[mu[x]] == mu[x] && phi[x] != x)
#define OUTER(x) (mu[x] != x && phi[mu[x]] == mu[x] && phi[x] == x)
int maximumMatching(const Graph &g, Edges &matching) {
   int n = g.size();
   vector<int> mu(n), phi(n), rho(n), scanned(n);
   REP(v,n) mu[v] = phi[v] = rho[v] = v; // (1) initialize
   for (int x = -1; ; ) {
      if (x < 0) { // (2) select even
         for (x = 0; x < n && (scanned[x] || !EVEN(x)); ++x);
         if (x == n) break;
      }
      int y = -1; // (3) select incident
      EACH(e, g[x])
         if (OUTER(e->dst) || (EVEN(e->dst) && rho[e->dst] != rho[x]))
            y = e->dst;
      if (y == -1) scanned[x] = true, x = -1;
      else if (OUTER(y)) phi[y] = x; // (4) growth
      else {
         vector<int> dx(n, -2), dy(n, -2); // (5,6), !TRICK! x % 2 --> x >= 0
         for (int k = 0, w = x; dx[w] < 0; w = k % 2 ? mu[w] : phi[w])
            dx[w] = k++;
         for (int k = 0, w = y; dy[w] < 0; w = k % 2 ? mu[w] : phi[w])
            dy[w] = k++;
         bool vertex_disjoint = true;
         REP(v,n) if (dx[v] >= 0 && dy[v] > 0) vertex_disjoint = false;
         if (vertex_disjoint) { // (5) augment
            REP(v,n) if (dx[v] % 2) mu[phi[v]] = v, mu[v] = phi[v];
            REP(v,n) if (dy[v] % 2) mu[phi[v]] = v, mu[v] = phi[v];
            mu[x] = y; mu[y] = x; x = -1;
            REP(v,n) phi[v] = rho[v] = v, scanned[v] = false;
         } else { // (6) shrink
            int r = x, d = n;
            REP(v,n)
               if (dx[v] >= 0 && dy[v] >= 0 && rho[v] == v && d > dx[v])
                  d = dx[v], r = v;
            REP(v,n)
               if (dx[v] <= d && dx[v] % 2 && rho[phi[v]] != r)
                  phi[phi[v]] = v;
            REP(v,n)
               if (dy[v] <= d && dy[v] % 2 && rho[phi[v]] != r)
                  phi[phi[v]] = v;
            if (rho[x] != r) phi[x] = y;
            if (rho[y] != r) phi[y] = x;
            REP(v,n) if (dx[rho[v]] >= 0 || dy[rho[v]] >= 0) rho[v] = r;
         }
      }
   }
```

```
   matching.clear();
   REP(u,n) if (u < mu[u]) matching.push_back( Edge(u, mu[u]) );
   return matching.size(); // make explicit matching
}
```

7.6.3. *Bipartite graph edge coloring.*

```
bool augment(vector< vector<int> > &C, int u, int c1, int c2) {
   int v = C[u][c1];
   if (v >= 0) {
      augment(C, v, c2, c1);
      C[v][c2] = u;
      C[u][c1] = -1;
   }
   C[u][c2] = v;
}
int bipartiteColouring(const Graph &g,
      int L, vector< vector<int> > &color) {
   int n = g.size(), deg = 0;
   REP(u, n) deg = max(deg, (int)g[u].size());
   vector< vector<int> > C(n, vector<int>(deg, -1));
   REP(u,L) EACH(e,g[u]) {
      int v = e->dst, cu = 0, cv = 0;
      while (C[u][cu] != -1) ++cu;
      while (C[v][cv] != -1) ++cv;
      if (cu != cv) augment(C, v, cu, cv);
      C[u][cu] = v; C[v][cu] = u;
   }
   color.assign(n, vector<int>(n)); // make explicit color-table
   REP(u, L) REP(c, deg) if (C[u][c] >= 0)
      color[u][C[u][c]] = c+1;
   return deg;
}
```

7.6.4. *Maximum Matching.*

```
#define EVEN(x) (mu[x] == x || (mu[x] != x && phi[mu[x]] != mu[x]))
#define ODD(x) (mu[x] != x && phi[mu[x]] == mu[x] && phi[x] != x)
#define OUTER(x) (mu[x] != x && phi[mu[x]] == mu[x] && phi[x] == x)
int maximumMatching(const Graph &g, Edges &matching) {
   int n = g.size();
   vector<int> mu(n), phi(n), rho(n), scanned(n);
   REP(v,n) mu[v] = phi[v] = rho[v] = v; // (1) initialize
   for (int x = -1; ; ) {
      if (x < 0) { // (2) select even
         for (x = 0; x < n && (scanned[x] || !EVEN(x)); ++x);
         if (x == n) break;
      }
      int y = -1; // (3) select incident
      EACH(e, g[x])
         if (OUTER(e->dst) || (EVEN(e->dst) && rho[e->dst] != rho[x]))
            y = e->dst;
      if (y == -1) scanned[x] = true, x = -1;
      else if (OUTER(y)) phi[y] = x; // (4) growth
      else {
         vector<int> dx(n, -2), dy(n, -2); // (5,6), !TRICK! x % 2 --> x >= 0
         for (int k = 0, w = x; dx[w] < 0; w = k % 2 ? mu[w] : phi[w])
            dx[w] = k++;
         for (int k = 0, w = y; dy[w] < 0; w = k % 2 ? mu[w] : phi[w])
            dy[w] = k++;
         bool vertex_disjoint = true;
         REP(v,n) if (dx[v] >= 0 && dy[v] > 0) vertex_disjoint = false;
         if (vertex_disjoint) { // (5) augment
            REP(v,n) if (dx[v] % 2) mu[phi[v]] = v, mu[v] = phi[v];
            REP(v,n) if (dy[v] % 2) mu[phi[v]] = v, mu[v] = phi[v];
            mu[x] = y; mu[y] = x; x = -1;
            REP(v,n) phi[v] = rho[v] = v, scanned[v] = false;
         } else { // (6) shrink
            int r = x, d = n;
            REP(v,n)
               if (dx[v] >= 0 && dy[v] >= 0 && rho[v] == v && d > dx[v])
                  d = dx[v], r = v;
            REP(v,n)
               if (dx[v] <= d && dx[v] % 2 && rho[phi[v]] != r)
                  phi[phi[v]] = v;
            REP(v,n)
               if (dy[v] <= d && dy[v] % 2 && rho[phi[v]] != r)
                  phi[phi[v]] = v;
```

```
            if (rho[x] != r) phi[x] = y;
            if (rho[y] != r) phi[y] = x;
            REP(v,n) if (dx[rho[v]] >= 0 || dy[rho[v]] >= 0) rho[v] = r;
          }
        }
```

7.6.5. *Minimum weight maximum matching.*

```
// adj[i][j] = w(i,j)
Weight adj[N][N];
int size, sorted[N][N], num[N];
void rec(int match[], int p, int l, Weight w, Weight &opt) {
  int q = p + 1, i;
  if (w >= opt) return;
  for (; q < size; ++q) if (match[q] == -1) break;
  int m = num[q], *list = sorted[q];
  REP(j,m) if (match[i=list[j]] == -1) { // process greedily
    match[q] = i, match[i] = q;
    w += adj[i][q];
    if (l + 1 < size / 2) rec(match, q, l+1, w, opt);
    else opt = min(opt, w);
    w -= adj[i][q];
    match[q] = -1, match[i] = -1;
  }
}
int gs;
bool compareWith(int i, int j) { return adj[gs][i] < adj[gs][j]; }
Weight minimumWeightMatching() {
  for (gs = 0; gs < size; ++gs) { // sort adjacent nodes by edge weight
    for (int j = gs+1; j < size; ++j)
      sorted[gs][num[gs]++] = j;
    sort(sorted[gs], sorted[gs]+num[gs], compareWith);
  }
  int match[size]; fill(match, match+size, -1);
  Weight opt = INF;
  rec(match, -1, 0, 0, opt);
  return opt;
}

////////////////////////////

Weight best[1<<26];
int minimumWeightMatching() {
  fill(best, best+(1<<size), INF);
  best[0] = 0;
  for (int S = 0; S < (1<<size); ++S) {
    int i;
    for (i = 0; S & (1<<i); ++i);
    for (int j = i+1; j < size; ++j) {
      if (S & (1<<j)) continue;
      best[S + (1<<i) + (1<<j)] =
        min(best[S + (1<<i) + (1<<j)], best[S] + adj[i][j]);
    }
  }
  return best[(1<<size)-1];
}
```

7.7. **Tree.**

7.7.1. *Tree isomorphism.*

```
struct Node {
  vector<Node *> child;
};
bool otreeIsomorphism(Node *n, Node *m) {
  if (n->child.size() != m->child.size()) return false;
  REP(i, n->child.size())
    if (!otreeIsomorphism(n->child[i], m->child[i])) return false;
  return true;
}

////////////////////////////////
```

```
struct Node {
  vector<Node *> child;
  vector<int> code;
};
void code(Node *n) {
  int size = 1;
  vector< pair<vector<int>, int> > codes;
  REP(i, n->child.size()) {
    code(n->child[i]);
    codes.push_back( make_pair(n->child[i]->code, i) );
    size += codes[i].first[0];
  }
  sort(codes.rbegin(), codes.rend()); // !reverse
  n->code.push_back(size);
  for (int i = 0; i < n->child.size(); ++i) {
    swap(n->child[i], n->child[ codes[i].second ]);
    n->code.insert(n->code.end(),
      codes[i].first.begin(), codes[i].first.end());
  }
}
bool utreeIsomorphism(Node *n, Node *m) {
  code(n); code(m); return n->code == m->code;
}
```

7.7.2. *Off-line least common ancestor (Tarjan).*

```
struct Query {
   int u, v, w;
   Query(int u, int v) : u(u), v(v), w(-1) { }
};

void visit(const Graph &g, int u, int w,
      vector<Query> &qs, vector<int> &color,
      vector<int> &ancestor, UnionFind &uf) {
   ancestor[ uf.root(u) ] = u;
   EACH(e, g[u]) if (e->dst != w) {
      visit(g, e->dst, u, qs, color, ancestor, uf);
      uf.unionSet( e->src, e->dst );
      ancestor[ uf.root(u) ] = u;
   }
   color[u] = 1;
   EACH(q, qs) {
      int w = (q->v == u ? q->u : q->u == u ? q->v : -1);
      if (w >= 0 && color[w]) q->w = ancestor[ uf.root(w) ];
   }
}

void leastCommonAncestor(const Graph &g, int r, vector<Query> &qs) {
   UnionFind uf(g.size());
   vector<int> color(g.size()), ancestor(g.size());
   visit(g, r, -1, qs, color, ancestor, uf);
}
```

7.7.3. *Height of the tree.*

```
Weight visit(const Graph &g, Graph& T, int i, int j) {
  if (T[i][j].weight >= 0) return T[i][j].weight;
  T[i][j].weight = g[i][j].weight;
  int u = T[i][j].dst;
  REP(k, T[u].size()) {
    if (T[u][k].dst == i) continue;
    T[i][j].weight = max(T[i][j].weight, visit(g,T,u,k)+g[i][j].weight);
  }
  return T[i][j].weight;
}
vector<Weight> height(const Graph& g) {
  const int n = g.size();
  Graph T(g); // memoise on tree
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < T[i].size(); ++j)
      T[i][j].weight = -1;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < T[i].size(); ++j)
      if (T[i][j].weight < 0)
        T[i][j].weight = visit(g, T, i, j);

  vector<Weight> ht(n); // gather results
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < T[i].size(); ++j)
      ht[i] = max(ht[i], T[i][j].weight);
  return ht;
}
```

7.7.4. *Tree diameter.*

```
typedef pair<Weight, int> Result;
Result visit(int p, int v, const Graph &g) {
   Result r(0, v);
   EACH(e, g[v]) if (e->dst != p) {
      Result t = visit(v, e->dst, g);
      t.first += e->weight;
      if (r.first < t.first) r = t;
   }
   return r;
}
Weight diameter(const Graph &g) {
   Result r = visit(-1, 0, g);
   Result t = visit(-1, r.second, g);
   return t.first; // (r.second, t.second) is farthest pair
}
```

7.8. **Road Tour.**

7.8.1. *Directed Euler Path.*

```
void visit(Graph& g, int a, vector<int>& path) {
   while (!g[a].empty()){
      int b = g[a].back().dst;
      g[a].pop_back();
      visit(g, b, path);
   }
   path.push_back(a);
}
bool eulerPath(Graph g, int s, vector<int> &path) {
   int n = g.size(), m = 0;
   vector<int> deg(n);
   REP(u, n){
      m += g[u].size();
      EACH(e, g[u]) --deg[e->dst]; // in-deg
      deg[u] += g[u].size(); // out-deg
   }
   int k = n - count(ALL(deg), 0);
   if (k == 0 || (k == 2 && deg[s] == 1)) {
      path.clear();
      visit(g, s, path);
      reverse(ALL(path));
      return path.size() == m + 1;
   }
   return false;
}
```

7.8.2. *Undirected Euler Path.*

```
void visit(const Graph &g, vector<vector<int> > &adj,int s,vector<int> &path){
   EACH(e, g[s]) if (adj[e->src][e->dst]) {
      --adj[e->src][e->dst];
      --adj[e->dst][e->src];
      visit(g, adj, e->dst, path);
   }
   path.push_back(s);
}
bool eulerPath(const Graph &g, int s, vector<int> &path) {
   int n = g.size();
   int odd = 0, m = 0;
   REP(i, n) {
      if (g[i].size() % 2 == 1) ++odd;
      m += g[i].size();
   }
   m /= 2;
   if (odd == 0 || (odd == 2 && g[s].size() % 2 == 0)) {
      vector< vector<int> > adj(n, vector<int>(n));
      REP(u, n) EACH(e, g[u]) ++adj[e->src][e->dst];
      path.clear();
      visit(g, adj, s, path);
      reverse(ALL(path));
      return path.size() == m + 1;
   }
   return false;
}
```

7.8.3. *Undirected Chinese Postman problem.*

```
Weight chinesePostman(const Graph &g) {
   Weight total = 0;
   vector<int> odds;
   REP(u, g.size()) {
      EACH(e, g[u]) total += e->weight;
      if (g[u].size() % 2) odds.push_back(u);
   }
   total /= 2;
   int n = odds.size(), N = 1 << n;
   Weight w[n][n]; // make odd vertices graph
   REP(u,n) {
      int s = odds[u]; // dijkstra's shortest path
      vector<Weight> dist(g.size(), INF); dist[s] = 0;
      vector<int> prev(g.size(), -2);
      priority_queue<Edge> Q;
      Q.push( Edge(-1, s, 0) );
      while (!Q.empty()) {
         Edge e = Q.top(); Q.pop();
         if (prev[e.dst] != -2) continue;
         prev[e.dst] = e.src;
         EACH(f,g[e.dst]) {
            if (dist[f->dst] > e.weight+f->weight) {
               dist[f->dst] = e.weight+f->weight;
               Q.push(Edge(f->src, f->dst, e.weight+f->weight));
            }
         }
      }
      REP(v,n) w[u][v] = dist[odds[v]];
   }
   Weight best[N]; // DP for general matching
   REP(S,N) best[S] = INF;
   best[0] = 0;

   for (int S = 0; S < N; ++S)
      for (int i = 0; i < n; ++i) if (!(S&(1<<i)))
         for (int j = i+1; j < n; ++j) if (!(S&(1<<j)))
            best[S|(1<<i)|(1<<j)] = min(best[S|(1<<i)|(1<<j)], best[S]+w[i][j]);
   return total + best[N-1];
}
```

7.8.4. *Shortest Hamiltonian.*

```
const int M = 20;
Weight best[1<<M][M];
int prev[1<<M][M];

void buildPath(int S, int i, vector<int> &path) {
   if (!S) return;
   buildPath(S^(1<<i), prev[S][i], path);
   path.push_back(i);
}

Weight shortestHamiltonPath(Matrix w, int s, vector<int> &path) {
   int N = 1 << n;
   REP(S,N) REP(i,n) best[S][i] = INF;
   best[1<<s][s] = 0;
   REP(S,N) REP(i,n) if (S&(1<<i)) REP(j,n)
      if (best[S|(1<<j)][j] > best[S][i] + w[i][j])
         best[S|(1<<j)][j] = best[S][i] + w[i][j],
            prev[S|(1<<j)][j] = i;
   int t = min_element(best[N-1], best[N-1]+n) - best[N-1];
   path.clear(); buildPath(N-1, t, path);
   return best[N-1][t];
}
```

7.9. **Other.**

7.9.1. *Linear extension.*

```
bool visit(const Graph &g, int v, vector<int> &order, vector<int> &color) {
   color[v] = 1;
   EACH(e, g[v]) {
      if (color[e->dst] == 2) continue;
```

```
      if (color[e->dst] == 1) return false;
      if (!visit(g, e->dst, order, color)) return false;
   }
   order.push_back(v); color[v] = 2;
   return true;
}
bool topologicalSort(const Graph &g, vector<int> &order) {
   int n = g.size();
   vector<int> color(n);
   REP(u, n) if (!color[u] && !visit(g, u, order, color))
      return false;
   reverse(ALL(order));
   return true;
}
```

7.9.2. *Graph isomorphism.*

```
typedef vector< vector<int> > Matrix;

typedef pair<int, int> VertexInfo;
#define index second
#define degree first
bool permTest(int k, const Matrix &g, const Matrix &h,
   vector<VertexInfo> &gs, vector<VertexInfo> &hs) {
  const int n = g.size();
  if (k >= n) return true;
  for (int i = k; i < n && hs[i].degree == gs[k].degree; ++i) {
   swap(hs[i], hs[k]);
   int u = gs[k].index, v = hs[k].index;
   for (int j = 0; j <= k; ++j) {
    if (g[u][gs[j].index] != h[v][hs[j].index]) goto esc;
    if (g[gs[j].index][u] != h[hs[j].index][v]) goto esc;
   }
   if (permTest(k+1, g, h, gs, hs)) return true;
esc:swap(hs[i], hs[k]);
  }
  return false;
}
bool isomorphism(const Matrix &g, const Matrix &h) {
  const int n = g.size();
  if (h.size() != n) return false;
  vector<VertexInfo> gs(n), hs(n);
  REP(i, n) {
   gs[i].index = hs[i].index = i;
   REP(j, n) {
    gs[i].degree += g[i][j];
    hs[j].degree += h[i][j];
   }
  }
  sort(ALL(gs)); sort(ALL(hs));
  REP(i, n) if (gs[i].degree != hs[i].degree) return false;

  return permTest(0, g, h, gs, hs);
}
```

8. Geometry

8.1. **Base.**

```
struct point{
      double x,y;
      point(double x=0, double y=0): x(x), y(y){}

      point operator +(point q){ return point(x+q.x,y+q.y); }
      point operator -(point q){ return point(x-q.x,y-q.y); }
      point operator *(double t){ return point(x+t,y*t); }
      point operator /(double t){ return point(x/t,y/t); }
      double operator *(point q){ return x*q.x+y*q.y; }
      double operator %(point q){ return x*q.y-y*q.x; }

      int cmp(point q) const{
            if(int t= ::cmp(x,q.x)) return t;
            return ::cmp(y,q.y);
      }

      bool operator ==(point q) const { return cmp(q)==0; }
      bool operator !=(point q) const { return cmp(q)!=0; }
```

```
      bool operator <(point q) const { return cmp(q)<0; }
      bool operator <=(point q) const { return cmp(q)<=0; }

      friend ostream& operator <<(ostream& o, point p){
            return o << "(" << p.x << ", " << p.y << ")";
      }

      static point pivot;
};

point point::pivot;

double abs(point p){ return hypot(p.x,p.y); }
double arg(point p){ return atan2(p.y,p.x); }
```

8.2. **Functions.**

```
// decide se q esta sobre o segmento fechado pr
bool between(point p, point q, point r){
      return ccw(p,q,r) == 0 && cmp((p-q)*(r-q)) <= 0;
}

// calcula a distancia do ponto r ao segmento pq
double seg_distance(point p, point q, point r){
      point A = r-q, B=r-p, C=q-p;
      double a = A*A, b = B*B, c = C*C;
      if(cmp(b,a+c)>=0) return sqrt(a);
      else if(cmp(a,b+c)>=0) return sqrt(b);
      else return fabs(A%B)/sqrt(c);
}

// classifica o ponto p em relacao ao poligono T
// return: 0 - p esta no exterior
// -1 - p esta na fronteira
// 1 - p esta no interior
int in_poly(point p, polygon &T){
      double a=0; int N = T.size();
      REP(i,N){
            if(between(T[i],p,T[(i+1)%N])) return -1;
            a += angle(T[i],p,T[(i+1)%N]);
      }
      return cmp(a)!=0;
}

typedef vector<point> polygon;

inline int ccw(point p, point q, point r){
      return cmp((p-r)%(q-r));
}

inline double angle(point p, point q, point r){
      point u = p-q, v = r-q;
      return atan2(u%v,u*v);
}

double dist(point a, point b){
   point c = a-b;
   return sqrt(c.x*c.x+c.y*c.y);
}
```

```
}

// comparacao radial
bool radial_lt(point p, point q){
      point P = p-point::pivot, Q = q-point::pivot;
      double R = P%Q;
      if(cmp(R)) return R>0;
      return cmp(P*P,Q*Q)<0;
}

// calcula a area orientado do poligono T
// T deve estar orientado positivamente
double poly_area(polygon &T){
      double s=0; int n=T.size();
      REP(i,n){
            s += T[i]%T[(i+1)%n];
      }
      return s/2;
}

// encontra o ponto de intersecao das retas pq e rs
point line_intersect(point p, point q, point r, point s){
      point a = q-p, b = s-r, c = point(p%q,r%s);
      return point(point(a.x,b.x)%c, point(a.y,b.y)%c)/(a%b);
}
```

8.3. **Line intesect.**

```
// decide se os segmentos fechados pq e rs tem pontos em comum
bool seg_intersect(point p, point q, point r, point s){
      point A = q-p, B = s-r, C = r-p, D=s-q;
      int a = cmp(A%C)+2*cmp(A%D);
      int b = cmp(B%C)+2*cmp(B%D);
      if(a==3||a==-3||b==3||b==-3) return false;
      int t=(p<r)+(p<s)+(q<r)+(q<s);
      return t!=0 && t!=4;
}
```

8.4. **Closest points.**

```
bool compy(point a, point b){
   return cmp(a.y,b.y) ? cmp(a.y,b.y) < 0 : cmp(a.x,b.x) < 0;
}

pair<point, point> closest_points_rec(vector<point>& px, vector<point>& py) {
   pair<point, point> ret;
   double d;

   if(px.size() <= 3) {
      double best = 1e10;
      for(int i = 0; i < px.size(); ++i)
         for(int j = i + 1; j < px.size(); ++j)
            if(dist(px[i], px[j]) < best) {
               ret = make_pair(px[i], px[j]);
               best = dist(px[i], px[j]);
            }

      return ret;
   }

   point split = px[(px.size() - 1)/2];
   vector<point> qx, qy, rx, ry;
   for(int i = 0; i < px.size(); ++i)
      if(px[i] <= split) qx.push_back(px[i]);
      else rx.push_back(px[i]);

   for(int i = 0; i < py.size(); ++i)
      if(py[i] <= split) qy.push_back(py[i]);
      else ry.push_back(py[i]);

   ret = closest_points_rec(qx, qy);
   pair<point, point> rans = closest_points_rec(rx, ry);
   double delta = dist(ret.first, ret.second);

   if((d = dist(rans.first, rans.second)) < delta) {
      delta = d;
      ret = rans;
   }

   vector<point> s;
   for(int i = 0; i < py.size(); ++i)
      if(cmp(abs(py[i].x - split.x), delta) <= 0)
         s.push_back(py[i]);

   for(int i = 0; i < s.size(); ++i)
      for(int j = 1; j <= 7 && i + j < s.size(); ++j)
         if((d = dist(s[i], s[i+j])) < delta) {
            delta = d;
            ret = make_pair(s[i], s[i+j]);
         }

   return ret;
}

pair<point, point> closest_points(vector<point> pts) {
   if(pts.size() == 1) return make_pair(point(-INF, -INF), point(INF, INF));

   sort(pts.begin(), pts.end());
   for(int i = 0; i + 1 < pts.size(); ++i)
      if(pts[i] == pts[i+1])
         return make_pair(pts[i], pts[i+1]);

   vector<point> py = pts;
   sort(py.begin(), py.end(), compy);
   return closest_points_rec(pts, py);
}
```

8.5. **Convex Hull.**

```
// determina o fecho convexo de um conjunto de pontos no plano
// destroi a lista de pontos T
polygon convex_hull(vector<point> &T){
      int j=0, k, n=T.size(); polygon U(n);

      point::pivot = *min_element(ALL(T));
      sort(ALL(T), radial_lt);
      for(k=n-2;k>=0&&ccw(T[0],T[n-1],T[k])==0;k--);
      reverse((k+1)+ALL(T));
      REP(i,n){
            // troque o >= por > para manter pontos colineares
            while(j>1&&ccw(U[j-1],U[j-2],T[i])>=0) j--;
            U[j++]=T[i];
      }
      U.erase(j+ALL(U));
      return U;
}
```

8.6. **Spanning Circle.**

```
// encontra o menor circulo que contem todos os pontos dados
typedef pair<point,double> circle;

bool in_circle(circle C, point p){
      return cmp(abs(p-C.first), C.second) <= 0;
}

point circumcenter(point p, point q, point r){
      point a = p-r, b = q-r, c = point(a*(p+r)/2, b*(q+r)/2);
      return point(c%point(a.y,b.y), point(a.x,b.x)%c)/(a%b);
}

circle spanning_circle(vector<point> &T){
      int n = T.size();
      random_shuffle(ALL(T));
      circle C(point(),-INF);
      REP(i,n) if(!in_circle(C,T[i])){
            C = circle(T[i],0);
            REP(j,i) if(!in_circle(C,T[j])){
                  C = circle((T[i]+T[j])/2, abs(T[i]-T[j])/2);
                  REP(k,j) if(!in_circle(C,T[k])){
                        point o = circumcenter(T[i],T[j],T[k]);
                        C = circle(o,abs(o-T[k]));
                  }
            }
      }
      return C;
}
```

8.7. **Polygon intersect.**

```
// determina o poligono intersecao dos dois poligonos convexos P e Q
// tanto P quanto Q devem estar orientados positivamente
polygon poly_intersect(polygon &P, polygon &Q){
      int m = Q.size(), n = P.size();
      int a=0, b=0, aa=0, ba=0, inflag=0;
      polygon R;
      while((aa<n||ba<m) && aa<2*n && ba<2*m){
            point p1 = P[a], p2 = P[(a+1)%n], q1 = Q[b], q2 = Q[(b+1)%m];
            point A = p2-p1, B=q2-q1;
            int cross = cmp(A%B), ha = ccw(p2,q2,p1), hb = ccw(q2,p2,q1);
            if(cross==0 && ccw(p1,q1,p2)==0 && cmp(A*B)<0){
                  if(between(p1,q1,p2)) R.push_back(q1);
                  if(between(p1,q2,p2)) R.push_back(q2);
                  if(between(q1,p1,q2)) R.push_back(p1);
                  if(between(q1,p2,q2)) R.push_back(p2);
                  if(R.size()<2) return polygon();
                  inflag = 1; break;
            }else if(cross!=0 && seg_intersect(p1,p2,q1,q2)){
                  if(inflag==0) aa = ba = 0;
                  R.push_back(line_intersect(p1,p2,q1,q2));
                  inflag = (hb>0) ? 1 : -1;
            }
            if(cross==0 && hb<0 && ha<0) return R;
            bool t = cross = 0 && hb == 0 && ha == 0;
```

```
        if(t ? (inflag==1) : (cross >=0) ? (ha<=0) : (hb>0)){
              if(inflag==-1) R.push_back(q2);
              ba++; b++; b%=m;
        }else{
              if(inflag==1) R.push_back(p2);
              aa++; a++; a%=n;
        }
  }
```

```
        if(inflag==0){
              if(in_poly(P[0],Q)) return P;
              if(in_poly(Q[0],P)) return Q;
        }
        R.erase(unique(ALL(R)),R.end());
        if(R.size()>1 && R.front() == R.back()) R.pop_back();
        return R;
  }
```

9. Useful mathematical facts

9.1. **Prime counting function ($\pi(x)$).** The prime counting function is asymptotic to $\frac{x}{\log x}$, by the prime number theorem.

x	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8
$\pi(x)$	4	25	168	1.229	9.592	78.498	664.579	5.761.455

9.2. **Partition function.** The partition function $p(x)$ counts show many ways there are to write the integer x as a sum of integers.

x	36	37	38	39	40	41	42
p(x)	17.977	21.637	26.015	31.185	37.338	44.583	53.174
x	43	44	45	46	47	100	
p(x)	63.261	75.175	89.134	105.558	125.754	190.569.292	

9.3. **Catalan numbers.** Catalan numbers are defined by the recurrence:

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$$

A closed formula for Catalan numbers is:

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

9.4. **Stirling numbers of the first kind.** These are the number of permutations of I_n with exactly k disjoint cycles. They obey the recurrence:

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$$

9.5. **Stirling numbers of the second kind.** These are the number of ways to partition I_n into exactly k sets. They obey the recurrence:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = k\begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$$

A "closed" formula for it is:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$$

9.6. **Bell numbers.** These count the number of ways to partition I_n into subsets. They obey the recurrence:

$$\mathcal{B}_{n+1} = \sum_{k=0}^{n}\binom{n}{k}\mathcal{B}_k$$

x	5	6	7	8	9	10	11	12
$\mathcal{B}_x$	52	203	877	4.140	21.147	115.975	678.570	4.213.597

9.7. **Turán's theorem.** No graph with n vertices that is K_{r+1}-free can have more edges than the Turán graph: A k-partite complete graph with sets of size as equal as possible.

9.8. **Generating functions.** A list of generating functions for useful sequences:

$(1,1,1,1,1,1,\ldots)$	$\frac{1}{1-z}$
$(1,-1,1,-1,1,-1,\ldots)$	$\frac{1}{1+z}$
$(1,0,1,0,1,0,\ldots)$	$\frac{1}{1-z^2}$
$(1,0,\ldots,0,1,0,1,0,\ldots,0,1,0,\ldots)$	$\frac{1}{1-z^2}$
$(1,2,3,4,5,6,\ldots)$	$\frac{1}{(1-z)^2}$
$(1,\binom{m+1}{m},\binom{m+2}{m},\binom{m+3}{m},\ldots)$	$\frac{1}{(1-z)^{m+1}}$
$(1,c,\binom{c+1}{2},\binom{c+2}{3},\ldots)$	$\frac{1}{(1-z)^c}$
$(1,c,c^2,c^3,\ldots)$	$\frac{1}{1-cz}$
$(0,1,\frac{1}{2},\frac{1}{3},\frac{1}{4},\ldots)$	$\ln\frac{1}{1-z}$

A neat manipulation trick is:

$$\frac{1}{1-z}G(z) = \sum_n \sum_{k\le n} g_k z^n$$

9.9. **Polyominoes.** How many free (rotation, reflection), one-sided (rotation) and fixed n-ominoes are there?

n	3	4	5	6	7	8	9	10
free	2	5	12	35	108	369	1.285	4.655
one-sided	2	7	18	60	196	704	2.500	9.189
fixed	6	19	63	216	760	2.725	9.910	36.446

9.10. **The twelvefold way (from Stanley).** How many functions $f\colon N \to X$ are there?

N	X	Any f	Injective	Surjective
dist.	dist.	x^n	$(x)_n$	$x!\left\{ {n \atop x} \right\}$
indist.	dist.	$\binom{x+n-1}{n}$	$\binom{x}{n}$	$\binom{n-1}{n-x}$
dist.	indist.	$\left\{ {n \atop 1} \right\} + \ldots + \left\{ {n \atop x} \right\}$	$[n \leq x]$	$\left\{ {n \atop k} \right\}$
indist.	indist.	$p_1(n) + \ldots p_x(n)$	$[n \leq x]$	$p_x(n)$

Where $\binom{a}{b} = \frac{1}{b!}(a)_b$ and $p_x(n)$ is the number of ways to partition the integer n using x summands.

9.11. **Common integral substitutions.** And finally, a list of common substitutions:

$\int F(\sqrt{ax+b})dx$	$u = \sqrt{ax+b}$	$\frac{2}{a}\int uF(u)du$
$\int F(\sqrt{a^2-x^2})dx$	$x = a\sin u$	$a\int F(a\cos u)\cos u du$
$\int F(\sqrt{x^2+a^2})dx$	$x = a\tan u$	$a\int F(a\sec u)\sec^2 u du$
$\int F(\sqrt{x^2-a^2})dx$	$x = a\sec u$	$a\int F(a\tan u)\sec u\tan u du$
$\int F(e^{ax})dx$	$u = e^{ax}$	$\frac{1}{a}\int \frac{F(u)}{u}du$
$\int F(\ln x)dx$	$u = \ln x$	$\int F(u)e^u du$

9.12. **Table of non-trigonometric integrals.** Some useful integrals are:

$\int \frac{dx}{x^2+a^2}$	$\frac{1}{a}\arctan\frac{x}{a}$
$\int \frac{dx}{x^2-a^2}$	$\frac{1}{2a}\ln\frac{x-a}{x+a}$
$\int \frac{dx}{a^2-x^2}$	$\frac{1}{2a}\ln\frac{a+x}{a-x}$
$\int \frac{dx}{\sqrt{a^2-x^2}}$	$\arcsin\frac{x}{a}$
$\int \frac{dx}{\sqrt{x^2-a^2}}$	$\ln\left(u+\sqrt{x^2-a^2}\right)$
$\int \frac{dx}{x\sqrt{x^2-a^2}}$	$\frac{1}{a}\text{arcsec}\left\lvert\frac{u}{a}\right\rvert$
$\int \frac{dx}{x\sqrt{x^2+a^2}}$	$-\frac{1}{a}\ln\left(\frac{a+\sqrt{x^2+a^2}}{x}\right)$
$\int \frac{dx}{x\sqrt{a^2+x^2}}$	$-\frac{1}{a}\ln\left(\frac{a+\sqrt{a^2-x^2}}{x}\right)$

9.13. **Table of trigonometric integrals.** A list of common and not-so-common trigonometric integrals:

$\int \tan x dx$	$-\ln\lvert\cos x\rvert$
$\int \cot x dx$	$\ln\lvert\sin x\rvert$
$\int \sec x dx$	$\ln\lvert\sec x+\tan x\rvert$
$\int \csc x dx$	$\ln\lvert\csc x-\cot x\rvert$
$\int \sec^2 x dx$	$\tan x$
$\int \csc^2 x dx$	$\cot x$
$\int \sin^n x dx$	$\frac{-\sin^{n-1}x\cos x}{n}+\frac{n-1}{n}\int\sin^{n-2}x dx$
$\int \cos^n x dx$	$\frac{\cos^{n-1}x\sin x}{n}+\frac{n-1}{n}\int\cos^{n-2}x dx$
$\int \arcsin x dx$	$x\arcsin x+\sqrt{1-x^2}$
$\int \arccos x dx$	$x\arccos x-\sqrt{1-x^2}$
$\int \arctan x dx$	$x\arctan x-\frac{1}{2}\ln\lvert1-x^2\rvert$

General strategy hints:

(1) Everyone should have read every problem by the end of the second hour of competition.
(2) **In the last hour, only one question should be attempted at once.**
(3) Always use vectors instead of arrays in the single dimensional case. Use vectors instead of arrays in the bidimensional case if the outermost dimension is not too big. This allows for better debugging.

When thinking (before coding):

(1) **Be organized!**
(2) Don't touch the computer unless the solution is done, including implementation details: avoid thinking too much at the computer. **Time spent on paper detailing a solution is time well spent.**
(3) A corollary to the above: **Don't touch the computer if you doubt your idea**.

In case you have no solution ideas:

(1) Talk with member team about one idea for solve. Case he already had read the problem.

In case of Wrong Answer:

(1) Make sure the algorithm is correct (i.e. **sketch a proof of correctness**) as soon as possible. Take your time and check it carefully. **If you're sure the idea is correct, make sure you don't second-guess it** when checking for bugs, even if an implementation bug is nowhere to be found.
(2) If the code uses vectors (c.f. general strategy hints), add #define _GLIBCXX_DEBUG at the very beginning of the source code and submit it again. A runtime error means out-of-bounds array access or other STL misuse.
(3) Debug on paper, and don't go back to the computer for every bug you find: **check the whole solution at least once more after finding each new bug.**
(4) Think of tricky test cases.
(5) Read the problem again. For every constraint found, check the printed source code.
(6) If you can't find any bug in five minutes, **go to the bathroom**. If you still can't find the bug, **go to another problem and come back to the wrong solution later**. Debugging a single program for too long leads to finding a lot of false bugs and makes it easy to overlook simple mistakes.

In case of Runtime Error:

(1) Check divisions and modulo operations.
(2) Check array indices (both in declarations and in accesses).
(3) Check for infinite recursion.

ACM ICPC Team Reference - Contents

Team Exceptions
Universidade Federal do Espírito Santo

Contents