ACM ICPC Cheat Sheet

University of Tehran

Faculty of Engineering

Mohsen Vakilian
m.vakilian@ece.ut.ac.ir

November 1, 2006

Contents

Chapter 1

Probable Mistakes

- Should it really be a `map` or a `multimap`?
- less than operator should impose a total ordering for the `map` to work.
- Resolve all the compiler warnings. Pay attention to type cast conversions in STL `size()`.
- A two dimensional array cannot be initialized by `fill()`.
- The predicate passed to the `set` template must have a default constructor.
- $EPS \geq 1e - 14$
- Be careful about the range of arrays. [] is your enemy!

Chapter 2

Skeleton

```
//Skeleton.cpp
#include <cassert>
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <iterator>
#include <vector>
#include <list>
#include <set>
#include <multiset>
#include <map>
#include <multimap>
#include <algorithm>
#include <cmath>
#include <cctype>
#include <ctime>
#include <stack>
#include <queue>

using namespace std;

//#define NDEBUG

#ifndef NDEBUG
#define D_(args) args
#define D_P(exp) { cerr << #exp << " = [" << (exp) << "]\n" << flush; }
#else
#define D_(args)
#define D_P(exp)
#endif

FILE *fin = freopen("x.in", "r", stdin);
FILE *fout = freopen("x.out", "w", stdout);

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

Chapter 3

Java I/O

```
BufferedReader fin = new BufferedReader(new FileReader("X.in"));
PrintWriter fout = new PrintWriter("X.out");
int n = Integer.parseInt(fin.readLine());
fout.close();
System.exit(0);

//

BufferedReader fin = new BufferedReader(new InputStreamReader(System.in));
StringTokenizer toker;
String line;
while ((line = fin.readLine().trim()) != null) {
    toker = new StringTokenizer(line);
    int m = Integer.parseInt(toker.nextToken());
    int n = Integer.parseInt(toker.nextToken());
}
```

Chapter 4

Time Measurement in C++

4.1 Measuring Program Running Time in C/C++

```
int main() {
    clock_t start = clock();
    //Do something
    clock_t ends = clock();
    cout << "Running Time : " << (double) (ends - start) / CLOCKS_PER_SEC << endl;
    return 0;
}
```

4.2 Creating a Simple Timer Program

```
int main(int argc,char **argv) {
    if (argc != 2) {
        //incorrect command line-argument
        cout << "Usage : timer <program_name>" << endl;
        return 1;
    }

    //Start the clock
    cout << "Starting " << argv[1] << "..." << endl;
    clock_t start = clock();
    //Start program
    system(argv[1]);
    //Program ends and stop clock
    clock_t ends = clock();
    cout << "Running Time : " << (double) (ends - start) / CLOCKS_PER_SEC << endl;

    return 0;
}
```

Chapter 5

Number Theoretic Algorithms

5.1 The Sieve of Eratosthenes

```
#include <cmath>

using namespace std;

//The Sieve of Eratosthenes
//This code has a bug! Can you find it out?
bool *sieve(int n){
    bool *p = new bool[n + 1];

    for(int i = 0; i < n + 1; i++)
        p[i] = true;

    p[0] = false;
    p[1] = false;
    int m = (int) sqrt((double) n);

    for (int i = 2; i <= m; i++)
        if (p[i])
            for (int k = i * i; k <= n; k += i)
                p[k] = false;

    return p;
}
```

5.2 Big Integer

16-bit integer range is $[-2^{15} \mathinner{..} 2^{15}-1]$ $[-32\,768 \mathinner{..} 32\,767]$.

32-bit integer range is $[-2^{31} \mathinner{..} 2^{31}-1]$ $[2\,147\,483\,648 \mathinner{..} 2\,147\,483\,647]$. Any integer with 9 or less digits can be safely computed using this data type.

64-bit integers (`long long` data type) can cover 18 digits integers fully, plus the 19th digit partially $[-2^{63} \mathinner{..} 2^{63}-1]$ $[9\,223\,372\,036\,854\,775\,808 \mathinner{..} 9\,223\,372\,036\,854\,775\,807]$.

`http://www.comp.nus.edu.sg/~stevenha/programming/BigInteger.h`

```
/* experimental new design of a big integer class */

typedef unsigned short PlatWord;
typedef unsigned long  PlatDoubleWord;

const int PlatDigits = (8* sizeof(PlatWord));
const PlatDoubleWord WordBase = ((PlatDoubleWord)1) << PlatDigits;

class BigInteger {
public:
    inline BigInteger();
    inline bool operator==(const BigInteger &aOther) const;
```

```
    inline bool operator!=(const BigInteger &aOther) const;
    BigInteger(const string& anInteger); // Create an integer
    BigInteger(const BigInteger &aOther) : digits(aOther.digits),
       negative(aOther.negative) {}
    string ToString() const;
    inline void Add(const BigInteger& aSource);
    inline void MultiplyAdd(const BigInteger &aX, const BigInteger &aY);
    inline void Negate() { negative = !negative; };

protected:
    typedef PlatWord ElementType;      // two bytes
    typedef PlatDoubleWord DoubleElementType; // four bytes
    enum { WordBits = PlatDigits};
    inline void Normalize();
    inline void AddWord(PlatWord aTerm);
    inline void MultWord(PlatWord aFactor);
    inline void DivideWord(PlatDoubleWord aNumber, PlatDoubleWord &aCarry);

private:
    vector<ElementType> digits;
    bool negative;
};


inline BigInteger::BigInteger() : negative(false) {
  digits.push_back(0);
}

inline bool BigInteger::operator==(const BigInteger &aOther) const {
  return (negative == aOther.negative && digits == aOther.digits);
}

inline bool BigInteger::operator!=(const BigInteger &aOther) const {
  return !(*this == aOther);
}

inline void BigInteger::Normalize() {
  int nr = digits.size();
  while (nr > 1 && digits[nr - 1] == 0) {
        digits.pop_back();
        nr--;
  }
}


inline void BigInteger::AddWord(PlatWord aTerm) {
    if (digits.size() == 0)
        digits.push_back(0);
    digits.push_back(0);
    DoubleElementType carry = 0;

    {
        DoubleElementType accu;
        accu = digits[0];
        accu += aTerm;
        accu += carry;
        digits[0] = (ElementType)(accu);
        carry = (accu >> WordBits);
    }

    int i = 1;
```

```
    while (carry) {
        DoubleElementType accu;
        accu = digits[i];
        accu += carry;
        digits[i] = (ElementType)(accu);
        carry = (accu >> WordBits);
        i++;
    }

    Normalize();
}

inline void BigInteger::MultWord(PlatWord aFactor) {
    unsigned i;
    digits.push_back(0);
    DoubleElementType carry = 0;

    for (i = 0;i < digits.size(); i++) {
        DoubleElementType accu;
        accu = digits[i];
        accu *= aFactor;
        accu += carry;
        digits[i] = (ElementType)(accu);
        carry = (accu >> WordBits);
    }

    assert(carry == 0);
    Normalize();
}

inline void BigInteger::DivideWord(PlatDoubleWord aNumber, PlatDoubleWord &aCarry) {
    PlatDoubleWord carry=0;
    int i;
    int nr = digits.size();

    for (i= nr - 1;i >= 0; i--) {
        PlatDoubleWord word = ((carry*WordBase) + ((PlatDoubleWord)(digits[i])));
        PlatWord digit = (PlatWord)(word / aNumber);
        PlatWord newCarry = (PlatWord)(word % aNumber);
        digits[i] = digit;
        carry= newCarry;
    }

    //carry now is the remainder
    aCarry = carry;
    Normalize();
}


inline void BigInteger::Add(const BigInteger& aSource) {
    while (digits.size() < aSource.digits.size())
        digits.push_back(0);
    digits.push_back(0);
    unsigned i;
    DoubleElementType carry = 0;
    for (i=0;i < aSource.digits.size(); i++) {
        DoubleElementType accu;
        accu = digits[i];
        accu += aSource.digits[i];
        accu += carry;
```

```
        digits[i] = (ElementType)(accu);
        carry = (accu>>WordBits);
    }

    while (carry) {
        DoubleElementType accu;
        accu = digits[i];
        accu += carry;
        digits[i] = (ElementType)(accu);
        carry = (accu>>WordBits);
        i++;
    }
    Normalize();
}


inline void BigInteger::MultiplyAdd(const BigInteger &aX, const BigInteger &aY) {
    unsigned i,j;
    for (i = aX.digits.size() + aY.digits.size() - digits.size(); i > 0; --i)
        digits.push_back(0);

    for (i = 0;i < aX.digits.size(); i++) {
        DoubleElementType carry = 0;
        DoubleElementType factor = aX.digits[i];
        for (j = 0;j < aY.digits.size(); j++) {
            DoubleElementType accu;
            accu = digits[i+j] + ((DoubleElementType)aY.digits[j]) * factor + carry;
            digits[i+j] = (ElementType)(accu);
            carry = (accu>>WordBits);
        }

        while (carry) {
            DoubleElementType accu;
            accu = digits[i+j] + carry;
            digits[i+j] = (ElementType)(accu);
            carry = (accu>>WordBits);
            j++;
        }
        assert(carry == 0);
    }
    Normalize();
}

BigInteger::BigInteger(const string &anInteger) {
    unsigned i=0;
    negative = false;
    if (anInteger[0] == '-') {
        negative = true;
        i++;
    }

    for (; i < anInteger.size(); ++i) {
        int digit = anInteger[i]-'0';
        MultWord(10);
        AddWord(digit);
    }
}

string BigInteger::ToString() const {
    BigInteger zero;
    if (*this == zero) return "0";
```

```
    string result;
    BigInteger number(*this);

    while (number != zero) {
        PlatDoubleWord digit;
        number.DivideWord(10, digit);
        result.push_back(digit);
    }

    int i, nr = result.size();

    for (i = (nr>>1) - 1; i >= 0; --i) {
        char swp = result[i];
        result[i] = '0' + result[result.size() - i - 1];
        result[result.size()-i-1] = '0' + swp;
    }
    if (nr & 1) result[(nr >> 1)] += '0';
    if (negative) result.insert(0, "-", 0, 1);
    return result;
}

int main(int argc, char** argv) {
#define X "123456789"
#define Y "23456789"
#define Z "456789"
    BigInteger x(X);
    BigInteger y(Y);
    BigInteger z(Z);
    BigInteger result;
    result.MultiplyAdd(x,y);
    result.Add(z);
    printf("%s * %s + %s = \n\t%s\n",X,Y,Z,result.ToString().c_str());
    /* In> 123456789 * 23456789 + 456789
       Out> 2895899850647310 */

    BigInteger yacasResult("2895899850647310");
    printf("Yacas says \n\t%s\n",yacasResult.ToString().c_str());

    return 0;
}
```

Chapter 6

Generating Permutations and Combinations

6.1 Generating the next largest permutation in lexicographical order

```
Next-Permutation(a)
▷ a_1 a_2 ··· a_n is a permutation of {1, 2, ..., n} not equal to n n−1 ··· 2 1
j ← n − 1
while a_j > a_{j+1}
      do j ← j − 1
j is the largest subscript with a_j < a_{j+1}
k ← n
while a_j > a_k
      do k ← k − 1
▷ a_k is the smallest integer greater than a_j to the right of a_j.
exchange a_j ↔ a_k
r ← n
s ← j + 1
while r > s
      do exchange a_r ↔ a_s
         r ← r − 1
         s ← s + 1
▷ This puts the tail end of the permutation after the jth position in increasing order.
```

6.2 Generating the next largest bit string

```
Next-Bit-String(b)
▷ b_{n−1} b_{n−2} ... b_1 b_0 is a bit string not equal to 11...11
i ← 0
while b_i = 1
      do b_i ← 0
         i ← i + 1
b_i ← 1
```

6.3 Generating the next r-combination in lexicographical order

```
Next-r-Combination(a)
▷ {a_1, a_2, ..., a_r} is a proper subset of {1, 2, ..., n}
  not equal to {n − r + 1, ..., n} with a_1 < a_2 < ··· < a_r.
i ← r
while a_i = n − r + i
      do i ← i − 1
a_i ← a_i + 1
for j ← i + 1 to r
      do a_j ← a_i + j − i
```

Chapter 7

Quicksort and Order Statistics

QUICKSORT sorts the entire array $A[p \,..\, r]$.

```
QUICKSORT(A, p, r)
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

PARTITION rearranges the subarray $A[p \,..\, r]$ in place.

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
     do if A[j] ≤ x
           then i ← i + 1
                exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

SELECT returns the ith smallest element of the array $A[p \,..\, r]$.

```
SELECT(A, p, r, i)
if p = r
   then return A[p]
q ← PARTITION(A, p, r)
k ← q − p + 1
if i = k            ▷ the pivot value is the answer
   then return A[q]
elseif i < k
   then return SELECT(A, p, q − 1, i)
else return SELECT(A, q + 1, r, i − k)
```

Chapter 8

Dynamic Programming

8.1 Longest Increasing Subsequence

```
#define MAX 400

int data[MAX][MAX];
int length[MAX];

/* length[i] is the length of the maximal increasing
subsequence of x[0]...x[n-1] that involves x[i] as its last element.
data[i][0]...data[i][length[i]-1]  is the sequence of
members of that maximal subsequence (so the last element is i).
*/

class Sequences {
public:
    /* if there are several answers of maximum length, return one whose
       last element is as small as possible */
    vector<int> LongestIncreasing(vector<int> &x) {
        int n = x.size();
        length[0] = 0;
        length[1] = 1;
        int i, j, k;

        data[0][0] = 0;
        length[0] = 1;

        for (i = 1; i < n; i++) {
            int max1 = 0;

            //Find the max length of increasing subsequences of x[0]...x[i-1] ending
                in values < x[i]
            int index1 = -1;

            for (j = 0; j < i; j++) {
                if (x[j] < x[i]) {
                    if (max1 < length[j] && x[j] < x[i]) {
                        max1 = length[j];
                        index1 = j;
                    }
                }
            }

            if (index1 == -1) {
                length[i] = 1;
                data[i][0] = x[i];
            }
```

```
            else {
                length[i] = max1 + 1;

                for (k = 0; k < max1; k++) {
                    data[i][k] = data[index1][k];
                }

                data[i][max1] = i;
            }
        }

        //Now data is complete
        int max = 0;
        int index = -1;

        for (i = 0; i < n; i++) {
            if (length[i] > max) {
                max = length[i];
                index = i;
            }
        }

        vector < int >ans;
        for (j = 0; j < length[index]; j++) {
            ans.push_back(data[index][j]);
        }

        return ans;
    }
};

int input[] = {5, 1, 6, 7, 3, 4, 5};
int main () {
    Sequences z;
    vector <int> x;

    int i, j;
    for (i = 0; i < sizeof (input) / sizeof (int); i++)
        x.push_back (input[i]);

    vector <int> ans;
    ans = z.LongestIncreasing(x);

    int n = x.size ();
    printf ("\nInput: ");

    for (i = 0; i < n; i++)
        printf ("%d ", x[i]);

    for (i = 0; i < n; i++) {
        printf ("\nending at [%d] = %d", i, x[data[i][0]]);
        for (j = 1; j < length[i]; j++)
            printf (", %d", x[data[i][j]]);
    }

    printf ("\n answer is ");
    int m = ans.size ();
    for (i = 0; i < m; i++)
        printf ("%d ", x[ans[i]]);

    return 0;
```

}

Chapter 9

Elementary Graph Algorithms

9.1 Breadth First Search

The breadth-first-search procedure BFS below assumes that the input graph $G = (V, E)$ is represented using adjacency lists.

9.1.1 Pseudocode

```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
     do color[u] ← WHITE
        d[u] ← ∞
        π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
     do u ← DEQUEUE(Q)
        for each v ∈ Adj[u]
            do if color[v] = WHITE
                  then color[v] ← GRAY
                       d[v] ← d[u] + 1
                       π[v] ← u
                       ENQUEUE(Q, v)
        color[u] ← BLACK
```

```
PRINT-PATH(G, s, v)
if v = s
   then print s
   else if π[v] = NIL
           then print "no path from" s "to" v "exists"
           else PRINT-PATH(G, s, π[v])
                print v
```

9.1.2 Analysis

Because the adjacency list of each vertex is scanned at most once, the total time spent in adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of BFS is $O(V + E)$.

9.1.3 Classifying edges by breadth-first search

A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

1. Prove that in a breadth-first search of an undirected graph, the following properties hold:

   (a) There are no back edges and no forward edges.
   (b) For each tree edge (u, v), we have $d[v] = d[u] + 1$.
   (c) For each cross edge (u, v), we have $d[v] = d[u]$ or $d[v] = d[u] + 1$.

2. Prove that in a breadth-first search of a directed graph, the following properties hold:
   (a) There are no forward edges.
   (b) For each tree edge (u, v), we have $d[v] = d[u] + 1$.
   (c) For each cross edge (u, v), we have $d[v] \leq d[u] + 1$.
   (d) For each back edge (u, v), we have $0 \leq d[v] \leq d[u]$.

9.2 Depth First Search

Vertex u is WHITE before time $d[u]$, GRAY between time $d[u]$ and time $f[u]$, and BLACK thereafter. The variable *time* is a global variable that we use for timestamping.

9.2.1 Pseudocode

```
DFS(G)
for each vertex u ∈ V[G]
      do color[u] ← WHITE
         π[u] ← NIL
dfsTime ← 0
for each vertex u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY                      ▷ White vertex u has just been discovered.
d[u] ← dfsTime ← dfsTime + 1
for each v ∈ Adj[u]                               ▷ Explore edge (u, v).
      do if color[v] = WHITE
            then π[v] ← u
                 DFS-VISIT(v)
color[u] ← BLACK                            ▷ Blacken u; it is finished.
f[u] ← dfsTime ← dfsTime + 1
```

9.2.2 Analysis

The running time of DFS is $\Theta(V + E)$.

9.2.3 Parenthesis theorem

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree, or
- the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in a depth-first tree, or

9.2.4 Classification of edges

We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on G.

1. **Tree edges** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).

2. **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.

3. **Forward edges** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.

4. **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

The DFS algorithm can be modified to classify edges as it encounters them. The key idea is that each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored (except that forward and cross edges are not distinguished):

1. WHITE indicates a tree edge,

2. GRAY indicates a back edge, and

3. BLACK indicates a forward or cross edge.

In an undirected graph, there may be some ambiguity in the type classification, since (u, v) and (v, u) are really the same edge. In such a case, the edge is classified as the first type in the classification list that applies. Equivalently, the edge is classified according to whichever of (u, v) or (v, u) is encountered first during the execution of the algorithm.

Edge (u, v) is

1. a tree edge or forward edge if and only if $d[u] < d[v] < f[v] < f[u]$,

2. back edge if and only if $d[v] < d[u] < f[u] < f[v]$, and

3. a cross edge if and only if $d[v] < f[v] < d[u] < f[u]$.

9.2.5 Theorem

In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

9.3 Topological Sort

A **topological sort** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

9.3.1 Pseudocode

Here are two possible algorithms to solve this problem:

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times f[v] for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

```
TOPOLOGICAL-SORT(G)
▷ The d array indicates a topological sorting of G.
initialize indegree[v] for all vertices
time ← 0
Q ← ∅
for i ← 1 to n
       do if indegree[v_i] = 0
             then ENQUEUE(Q, v_i)
repeat   v ← DEQUEUE(Q)
         time ← time + 1
         d[v] ← time
         for all edges (v, w)
               do indegree[w] ← indegree[w] − 1
                  if indegree[w] = 0
                     then ENQUEUE(Q, w)
   until Q = ∅
```

9.3.2 Analysis

We can perform a topological sort in time $\Theta(V + E)$.

9.4 Strongly connected components

A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other.

9.4.1 Pseudocode

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $f[u]$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices
  in order of decreasing $f[u]$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
  separate strongly connected component

9.4.2 Analysis

The above algorithm is linear ($\Theta(V + E)$-time).

9.5 Articulation points, bridges, and biconnected components

Let $G = (V, E)$ be a connected, undirected graph. An **articulation point** of G is a vertex whose removal disconnects G. A **bridge** of G is an edge whose removal disconnects G. A **biconnected component** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. We can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G.

9.5.1 Articulation points

1. Prove that the root of G_π is an articulation point of G if and only if it has at least two children in G_π.
2. Let v be a nonroot vertex of G_π. Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.
3. Let
$$low[v] = \min \begin{cases} d[v], \\ d[w] : (u, w) \text{ is a back edge for } u = v \text{ or some descendant } u \text{ of } v. \end{cases}$$
4. Show how to compute $low[v]$ for all vertices $v \in V$ in $O(E)$ time.
5. Show how to compute all articulation points in $O(E)$ time.

```
#include <vector>
#include <algorithm>

using namespace std;

#define INF 1000000000
#define MAX 1000
#define NOPAR -1
#define WHITE 0
#define GRAY 1
#define BLACK 2

int g[MAX][MAX];
int n; //number of vertices. (Note: vertices are numbered from 1 to n but this can be
    easily changed by changing loops)
int color[MAX], p[MAX], d[MAX], f[MAX];
```

```
int dfsTime;

//These are used to delete and restore vertices
bool deleted[MAX];
int backup[MAX];

int low[MAX], nchilds[MAX], iscut[MAX];

void init() {
    int i, j;
    for (i = 0; i < MAX; ++i)
        for (j = 0; j < MAX; ++j)
            g[i][j] = 0;
    n = 0;
    fill(deleted, deleted + MAX, true);
}

void readNext() {
    /*
    Remember to
    1. Set 'n' (the number of vertices)
    2. Keep the symmetry of the adjacency matrix
    3. Set the deleted[] if it's needed
    */

    //...
}

//Returns the adjacents of u
vector<int> adjs(int u) {
    vector<int> a;
    for (int v = 1; v <= n; ++v) {
        if (deleted[v]) continue;
        if (g[u][v] >= 1)
            a.push_back(v);
    }

    return a;
}

void delVertex(int u) {
    for (int v = 1; v <= n; ++v)
        backup[v] = g[u][v];

    for (int v = 1; v <= n; ++v)
        g[u][v] = g[v][u] = 0;

    deleted[u] = true;
}

void restoreVertex(int u) {
    for (int v = 1; v <= n; ++v)
        g[u][v] = g[v][u] = backup[v];

    deleted[u] = false;
}

void dfsVisit(int u) {
    color[u] = GRAY;
    d[u] = ++dfsTime;
```

```
    low[u] = d[u]; //low[u] keeps the d of the lowest back edge going out from u or
        any of its descendants.

    vector<int> a = adjs(u);

    for (int i = 0; i < (int) a.size(); ++i) {
        int v = a[i];
        if (deleted[v]) continue;
        if (color[v] == WHITE) {
            p[v] = u;
            dfsVisit(v);
            low[u] = min(low[u], low[v]);
        }
        else if (color[v] == GRAY) { //Check if a lower back edge is coming out of u
            low[u] = min(low[u], d[v]);
        }
    }

    color[u] = BLACK;
    f[u] = ++dfsTime;
}

//Returns the number of connected components.
int dfs() {
    fill(color, color + MAX, WHITE);
    fill(p, p + MAX, NOPAR); //the parent array
    fill(d, d + MAX, 0); //the start time array
    fill(f, f + MAX, 0); //the finish time array
    fill(low, low + MAX, INF);

    dfsTime = 0;

    int comps = 0; //number of connected components

    for (int v = 1; v <= n; ++v) {
        if (deleted[v]) continue;
        if (color[v] == WHITE) {
            dfsVisit(v);
            ++comps;
        }
    }

    return comps;
}

//nchild[u] indicates the number of children of u in the DFS tree.
void calc_nchilds() {
    fill(nchilds, nchilds + MAX, 0);

    for (int v = 1; v <= n; ++v) {
        if (deleted[v]) continue;
        if (p[v] != NOPAR)
            ++nchilds[p[v]];
    }
}

//Sets iscut[] for each vertex.
void findCutVertices() {
    fill(iscut, iscut + MAX, false);

    dfs();
```

```
    calc_nchilds();

    //v is an articulation point iff v has a child s such that there is no back edge
        from s or any descendant of s to a proper ancestor of v.
    for (int v = 1; v <= n; ++v) {
        if (p[v] == NOPAR) continue;

        if (low[v] >= d[p[v]])
            iscut[p[v]] = true;
    }

    //The root of the DFS tree is an articulation point iff it has at least two
        children.
    for (int v = 1; v <= n; ++v) {
        if (deleted[v]) continue;
        if (p[v] != NOPAR) continue;

        if (nchilds[v] >= 2)
            iscut[v] = true;
        else
            iscut[v] = false;
    }
}
```

9.5.2 Bridges and biconnected components

1. Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G.
2. Show how to compute all the bridges of G in $O(E)$ time.
3. Prove that the biconnected components of G partition the nonbridge edges of G.
4. Give an $O(E)$-time algorithm to label each edge e of G with a positive integer $bcc[e]$ such that $bcc[e] = bcc[e']$ if and only if e and e' are in the same biconnected component.

   Solution. At first, we remove all bridges, then each call to DFS-Visit returns a biconnected component.

9.6 All-Pairs Shortest Paths

9.6.1 Floyd-Warshall

Floyd-Warshall(W)

$n \leftarrow rows[W]$
$D \leftarrow W$
for $k \leftarrow 1$ **to** n
    do for $i \leftarrow 1$ **to** n
        do for $j \leftarrow 1$ **to** n
            do $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$
return D

9.7 Eulerian Tours

An **eulerian trail** in a digraph (or graph) is a trail containing all edges. An **eulerian circuit** is a closed trail containing all edges. A digraph is **eulerian** if and only if it has an eulerian circuit.

Find-Eulerian-Circuit(G)

$\triangleright$ *circuit* is a global array.
circuit-pos $\leftarrow 0$
pick an arbitrary vertex u.
Find-Circuit(u)

```
Find-Eulerian-Tour(G)
  ▷ circuit is a global array.
  circuit-pos ← 0
  Pick an odd degree vertex u.
  Find-Circuit(u)
```

```
Find-Circuit(v)
  ▷ nextnode and visited are local arrays.
  ▷ The path will be found in reverse order.
  while v has neighbors
        do Pick an arbitrary neighbor node w of node v
           Delete-Edges(v, w)
           Find-Circuit(w)
  circuit[circuit-pos] ← v
  circuit-pos ← circuit-pos +1
```

9.7.1 Analysis

Both of these algorithms run in $O(E + V)$ time, if the graph is represented by adjacency list. With larger graphs, there is a danger of overflowing the run-time stack, so we might have to use our own stack.

9.7.2 Nonrecursive Eulerian Tour

```
Find-Circuit(v)
  Push(S, v)
  while S ≠ ∅
        do while Top(S) has neighbors
                 do Pick an arbitrary neighbor node w of node Top(S)
                    Delete-Edges(Top(S), w)
                    Push(S, w)
           while Top(S) has no neighbors
                 do Push(circuit, Pop(S))
```

Chapter 10

Maximum Bipartite Matching

A **matching** in a graph G is a set of non-loop edges with no shared endpoints. The vertices incident to the edges of a matching M are **saturated** by M; the others are **unsaturated** (we say M-*saturated* and M-*unsaturated*).

Given a matching M in an X, Y-bigraph G, we search for M-augmenting paths from each M-unsaturated vertex in X. We need only search from vertices in X, because every augmenting path has odd length and thus has ends in both X and Y. We will search from the unsaturated vertices in X simultaneously.

10.1 Pseudocode

Input: An X, Y-bigraph G, a matching M in G, and the set U of M-unsaturated vertices in X.

Idea: Explore M-alternating paths from U, letting $S \subseteq X$ and $T \subseteq Y$ be the sets of vertices reached. $Mark$ vertices of S that have been explored for path extensions. As a vertex is reached, record the vertex from, which it is reached.

Augmenting-Path(G, M, U)

$S \leftarrow U$
$T \leftarrow \varnothing$
while S has some unmarked vertices
    do Select an unmarked vertex $x \in S$.
        for each $y \in N(x)$ such that $xy \notin M$
            do if y is unsaturated
                then return an M-augmenting path from U to y.
                else $\triangleright$ y is matched to some $w \in X$ by M.
                    Include y in T (reached from x).
                    Include w in S (reached from y).
        Mark x.
$\triangleright$ Announce $T \cup (X - S)$ as a minimum cover and M as a maximum matching.
return M

10.2 Analysis

Let G be an X, Y-bigraph with n vertices and m edges. If the time for one edge exploration is bounded by a constant, then this algorithm to find a maximum matching runs in time $O(mn)$.

10.3 Implementation in C++

```cpp
typedef set<int> Set;
typedef Set::iterator SetIter;

int n;
vector<bool> mark;
vector<int> match, parent;
Set S, T;

class IsUnmarked {
```

```
public:
    bool operator()(int x) {
        return !mark[x];
    }
};

int findAugment() {
    while (true) {
        SetIter it = find_if(S.begin(), S.end(), IsUnmarked());
        if (it == S.end()) return -1;
        int x = *it;
#ifndef NDEBUG
        printf("\nx=%d\n", x);
#endif

        vector<int> adj;
        getAdj(x, adj);

        for (int i = 0; i < adj.size(); ++i) {
            int y = adj[i];

            if (match[x] == y) continue;
#ifndef NDEBUG
            printf("y=%d\n", y);
#endif
            if (match[y] == -1)  {
                parent[y] = x;
                return y;
            }


            int w = match[y];
#ifndef NDEBUG
            printf("w=%d\n", w);
#endif
            if (mark[w])
                continue;

            parent[y] = x;
            parent[w] = y;

            T.insert(y);
            S.insert(w);
        }

        mark[x] = true;
    }
}

bool augment() {
#ifndef NDEBUG
    printf("\naugment()\n");
#endif
    init();
    int v = findAugment();
    if (v == -1) return false;
    while (v != -1) {
        match[v] = parent[v];
        match[parent[v]] = v;
#ifndef NDEBUG
        printf("%d & %d\n", v, parent[v]);
```

```
#endif

        v = parent[parent[v]];
    };

    return true;
}

void init() {
    T.clear();
    fill(parent.begin(), parent.end(), -1);
    fill(mark.begin(), mark.end(), false);

    S.clear();

    Fill S by U (the set of M-unsaturated vertices in X).
}

void solve() {
    mark.assign(n, false);
    match.assign(n, -1);
    parent.assign(n, -1);

    while (augment())
        ;

    printOutput();
}
```

Chapter 11

String Matching

Given two strings P, T, We want to check whether P is a substring of T.

11.1 Naive String Matching Algorithm

As the name suggest, this algorithm search checks for all position i whether T is a substring of P. It uses a loop that checks the condition $P[0\,..\,m-1] = T[i\,..\,i+m-1]$. Let $n := length[T]$ and $m := length[P]$ so the complexity of this algorithm is $O((n-m+1)\times m)$.

```
/*
    naiveStringMatch(T, P)
    - match whether P is substring of T.
    - return the starting index of the first occurence of P in T.
*/
int naiveStringMatch(const string &T, const string &P) {
    int i, j, n, m;
    bool found;

    n = T.size();
    m = P.size();

    for (i = 0; i <= n - m; i++) {
        found = true;
        for (j = 0; j < m; j++)
            if (P[j] != T[i + j]) {
                found = false;
                break;
            }
        if (found) return i;
    }
    return -1;
}
```

11.2 Rabin-Karp String Matching Algorithm

```
/* Rabin-Karp String Matching Algorithm
   rabinKarpStringMatch(T,P,d,q)
   - Assume T and P consist only a..z and A..Z
   - radix d, prime q
   - match whether P is a substring of T
   - return the starting index of the first occurence of P in T
   - n = length[T]
   - m = length[P]

     Worst Case : O((n - m + 1) * m)
     Best Case : O(n + m)
```

```
*/

#define tonum(c) (c >= 'A' && c <= 'Z' ? c - 'A' : c - 'a' + 26)

/* return a^p mod m */
int mod(int a, int p, int m) {
      if (p == 0) return 1;
      int sqr = mod(a, p/2, m) % m;
      sqr = (sqr * sqr) % m;
      if (p & 1) return ((a % m) * sqr) % m;
      else return sqr;
}

int rabinKarpStringMatch(const string &T, const string &P, int d, int q) {
      int i, j, p, t, n, m, h;
      bool found;

      n = T.size();
      m = P.size();

      h = mod(d, m - 1, q);
      p = t = 0;

      for (i = 0; i < m; i++) {
            p = (d * p + tonum(P[i])) % q;
            t = (d * t + tonum(T[i])) % q;
      }

      for (i = 0; i <= n - m; i++) {
            if (p == t) {
                  found = true;
                  for (j = 0; j < m; j++)
                        if (P[j] != T[i + j]) {
                              found = false;
                              break;
                        }
                  if (found) return i;
            } else {
                  t = (d * (t - ((tonum(T[i]) * h) % q)) + tonum(T[i + m])) % q;
            }
      }
      return -1;
}
```

11.3 The Knuth-Morris-Pratt Algorithm

The running time of `computeNext()` is $\Theta(m)$ and the matching time of `go()` is $\Theta(n)$.

Given a pattern $P[1\,..\,m]$, the ***prefix function*** for pattern P is the function $next : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$ such that $next[state] = max\{k : k < state \text{ and } P_k \sqsupset P_q\}$.

By Keeping the *state*, subsequent calls to `go` can be thought of as matching against a single long text.

```
P = "␣" + P; //next[] is an array of size one more than that of P's.
computeNext(P); //It should be computed just once before go()
int finalState = go(T, P, 0);

int go(string &T, string &p, int state) {
     for (int i = 0; i < t.size(); ++i) {
          while (state > 0 && P[state + 1] != T[i])
               state = next[state];
          if (P[state + 1] == T[i])
               ++state;
          if (state == p.size()) {
```

```
            //Pattern occurs here with shift i - m.
            state = next[state];
        }
    }

    return state;
}

void computeNext(string &P) {
    next[1] = 0;
    int k = 0;
    for (int state = 2; state <= p.size(); ++state)
        while (k > 0 && P[k + 1] != P[state])
            k = next[k];
        if (P[k + 1] == P[state])
            ++k;
        next[state] = k;
}
```

Chapter 12

Geometric Algorithms

12.1 Trigonometric Functions

tan

```
double tan(double x);
```

Description

This function computes the tangent of x. (which should be given in radians).

Return Value

The tangent of x. If the absolute value of x is finite but greater than or equal to 2^{63}, the return value is 0 (since for arguments that large each bit of the mantissa is more than π). If the value of x is infinite or `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

atan2

```
double atan2(double y, double x);
```

Description

This function computes the angle, in the range $[-\pi \mathinner{..} \pi]$ radians, whose tangent is y/x. In other words, it computes the angle, in radians, of the vector (x, y) with respect to the $+x$ axis, reckoning the counterclockwise direction as positive, and returning the value in the range $[-\pi \mathinner{..} \pi]$.

Return Value

The arc tangent, in radians, of y/x. π is returned if x is negative and y is a negative zero, -0.0. π is returned, if x is negative, and y is a positive zero, $+0.0$.

If either x or y is infinite, `atan2` returns, respectively, π with the sign of y or zero, and `errno` is left unchanged. However, if *both* arguments are infinite, the return value is `NaN` and `errno` is set to `EDOM`.

A `NaN` is returned, and `errno` is set to `EDOM`, if either x and y are both zero, or if either one of the arguments is a `NaN`.

```
#include <iostream>
#include <cmath>
#include <list>
#include <cassert>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

#define RIGHT 1
#define ON 0
```

```
#define LEFT -1
#define EPS 1e-10

#define CPR const Point &
#define CP3R const Point3D &
#define CLR const Line &
#define CL3R const Line3D &

bool eq(double d1, double d2) {
    return fabs(d1 - d2) <= EPS;
}

bool lt(double d1, double d2) {
    return d1 <= d2 - EPS;
}

bool gt(double d1, double d2) {
    return d1 >= d2 + EPS;
}

bool lteq(double d1, double d2) {
    return d1 <= d2 + EPS;
}

bool gteq(double d1, double d2) {
    return d1 >= d2 - EPS;
}

struct Point {
    double x, y;

    Point() {}
    Point(double x, double y) : x(x), y(y) {}

    bool operator==(CPR p) {
        return (eq(p.x, x) && eq(p.y, y));
    }

    bool operator!=(CPR p) {
        return !(*this == p);
    }

    Point operator-(CPR p2) const {
        return Point(p2.x - x, p2.y - y);
    }

    Point operator+(CPR p2) const {
        return Point(p2.x + x, p2.y + y);
    }

    friend ostream &operator<<(ostream &output, CPR p);
};

ostream &operator<<(ostream &output, CPR p) {
    output << "(" << p.x << ", " << p.y << ")";
    return output;
}

Point operator*(double scalar, CPR p) {
    return Point(scalar * p.x, scalar * p.y);
}
```

```
struct Line {
    Point p1, p2;
    double x, y;

    Line() {}
    Line(CPR p1, CPR p2): p1(p1), p2(p2), x(p2.x - p1.x), y(p2.y - p1.y) {}

    bool operator==(CLR l) {
        return (p1 == l.p1 && p2 == l.p2);
    }

    bool operator!=(CLR l) {
        return !(*this == l);
    }

    Line operator-(CLR l2) const {
        return Line(p1 - l2.p1, p2 - l2.p2);
    }

    Line operator+(CLR l2) const {
        return Line(p1 + l2.p1, p2 + l2.p2);
    }

    friend ostream &operator<<(ostream &output, CLR l);
};

ostream &operator<<(ostream &output, CLR l) {
    output << l.p1 << "->" << l.p2;
    return output;
}

struct Point3D {
    double x, y, z;

    Point3D() {}
    Point3D(double x, double y, double z) : x(x), y(y), z(z) {}

    Point3D operator-(CP3R p2) const {
        return Point3D(p2.x - x, p2.y - y, p2.z - z);
    }

    Point3D operator+(CP3R p2) const {
        return Point3D(p2.x + x, p2.y + y, p2.z + z);
    }
};

struct Line3D {
    Point3D p1, p2;
    double x, y, z;

    Line3D() {}
    Line3D(CP3R p1, CP3R p2) : p1(p1), p2(p2), x(p2.x - p1.x), y(p2.y - p1.y), z(p2.z
        - p1.z) {}

    Line3D operator-(CL3R l2) const {
        return Line3D(p1 - l2.p1, p2 - l2.p2);
    }

    Line3D operator+(CL3R l2) const {
        return Line3D(p1 + l2.p1, p2 + l2.p2);
```

```
    }
};

struct Polygon {
    list<Point> vertices;
//    vector<Point> vertices;

    typedef list<Point>::iterator Iterator;
//    typedef vector<Point>::iterator Iterator;

    int size() { return (int) vertices.size(); }

    Iterator next(Iterator iter) {
        assert(!vertices.empty());

        if (iter == vertices.end())
            return vertices.begin();
        ++iter;
        if (iter == vertices.end())
            return vertices.begin();
        return iter;
    }

    Iterator last() {
        assert(!vertices.empty());

        Iterator lastIter = vertices.end();
        --lastIter;
        return lastIter;
    }

    Iterator prev(Iterator iter) {
        assert(!vertices.empty());

        if (iter == vertices.begin())
            return last();
        --iter;
        return iter;
    }
/*
    CPR operator[](int index) {
        return vertices[index];
    }
*/

    friend ostream &operator<<(ostream &output, Polygon &g);
};

ostream &operator<<(ostream &output, Polygon &g) {
    string comma = "";
    for (Polygon::Iterator iter = g.vertices.begin(); iter != g.vertices.end();
       ++iter) {
        output << comma << *iter;
        comma = ", ";
    }

    return output;
}

Line3D crossProduct(CL3R l1, CL3R l2) {
    return Line3D(Point3D(0, 0, 0), Point3D(l1.y * l2.z - l1.z * l2.y, l1.z * l2.x -
```

```
        l1.x * l2.z, l1.x * l2.y - l1.y * l2.x));
}

double zCrossProduct(CLR l1, CLR l2) {
    return l1.x * l2.y - l1.y * l2.x;
}

double dotProduct(CLR l1, CLR l2) {
    return l1.x * l2.x + l1.y * l2.y;
}

Line scalarProduct(CLR l1, double s) {
    return Line(s * l1.p1, s * l1.p2);
}
```

12.2 Distances

```
double dist2(CPR p1, CPR p2) {
    return (p2.x - p1.x) * (p2.x - p1.x) + (p2.y - p1.y) * (p2.y - p1.y);
}

double dist(CPR p1, CPR p2) {
    return sqrt(dist2(p1, p2));
}

double length2(CLR l) {
    return dist2(l.p1, l.p2);
}

double length(CLR l) {
    return dist(l.p1, l.p2);
}
```

Distance of the point P from the vector $\overrightarrow{P_1P_2}$ is

$$PH = \frac{|\overrightarrow{P_1P} \times \overrightarrow{P_1P_2}|}{|\overrightarrow{P_1P_2}|}$$

```
double pointLineDist(CPR p, CLR l) {
    Line AP(l.p1, p);
    return fabs(zCrossProduct(AP, l)) / length(l);
}

double pointSegDist(CPR p, CLR l);

//Return the position of the point relative to the line.
int relativePos(CPR p, CLR l) {
    double cz = zCrossProduct(l, Line(l.p1, p));
    if (eq(cz, 0.0))
        return ON;
    else if (gt(cz, 0))
        return LEFT;
    else
        return RIGHT;
}

//Do we make a left or right turn at p1?
int turnDir(CPR p0, CPR p1, CPR p2) {
    double z = zCrossProduct(Line(p0, p2), Line(p0, p1));
```

```
    if (lt(z, 0))
        return LEFT;
    else if (gt(z, 0))
        return RIGHT;
    else
        return ON;
}

bool pointOnSeg(CPR p, CLR l) {
    return (relativePos(p, l) == ON && eq(length(l), dist(p, l.p1) + dist(p, l.p2)));
}
```

12.3 Intersection

```
bool linesIntersect(CLR l1, CLR l2) {
    double z = zCrossProduct(l1, l2);

    if (!eq(z, 0))
        return true;
    else //are the same
        return (relativePos(l1.p1, l2) == ON);
}

bool segsIntersect(CLR l1, CLR l2) {
    int r1 = relativePos(l1.p1, l2);
    int r2 = relativePos(l1.p2, l2);
    int r3 = relativePos(l2.p1, l1);
    int r4 = relativePos(l2.p2, l1);

    if (r1 == ON || r2 == ON || r3 == ON || r4 == ON)
        return true;
    return (r1 != r2 && r3 != r4);
}

bool segLineIntersect(CLR seg, CLR l) {
    int r1 = relativePos(seg.p1, l);
    int r2 = relativePos(seg.p2, l);

    return (r1 * r2 <= 0);
}
```

12.3.1 Point of Intersection of Two Lines

For the lines AB and CD in two dimensions, the most straight forward way to calculate the intersection of them is to solve the system of two equations and two unknowns:

$$A_x + (B_x - A_x)i = C_x + (D_x - C_x)j$$

$$A_y + (B_y - A_y)i = C_y + (D_y - C_y)j$$

The point of intersection is:

$$(A_x + (B_x - A_x)i,\ A_y + (B_y - A_y)i)$$

In three dimensions, solve following system of equations, where i and j are the unknowns:

$$A_x + (B_x - A_x)i = C_x + (D_x - C_x)j$$

$$A_y + (B_y - A_y)i = C_y + (D_y - C_y)j$$

$$A_z + (B_z - A_z)i = C_z + (D_z - C_z)j$$

If this system has a solution $(i,\ j)$, where $0 \leq i \leq 1$ and $0 \leq j \leq 1$, then the line segments intersect at:

$$(A_x + (B_x - A_x)i,\ A_y + (B_y - A_y)i,\ A_z + (B_z - A_z)i)$$

```
Point intersectLines(CLR l1, CLR l2) {
    double m = l1.x, n = l1.y, o = l2.x, p = l2.y;
    Point A = l1.p1, B = l1.p2, C = l2.p1, D = l2.p2;

    assert(!eq(n * o, m * p));
    double j = (n * A.x - m * A.y - n * C.x + m * C.y) / (n * o - m * p);
    double i = -1.0;

    if (!eq(m, 0.0))
        i = (C.x + o * j - A.x) / m;
    else {
        assert(!eq(n, 0.0));
        i = (C.y + p * j - A.y) / n;
    }

    return Point(A.x + l1.x * i, A.y + l1.y * i);
}

//l1: segment, l2: line
Point intersectSegLine(CLR l1, CLR l2) {
    double m = l1.x, n = l1.y, o = l2.x, p = l2.y;
    Point A = l1.p1, B = l1.p2, C = l2.p1, D = l2.p2;

    assert(!eq(n * o, m * p));
    double j = (n * A.x - m * A.y - n * C.x + m * C.y) / (n * o - m * p);
    double i = -1.0;

    if (!eq(m, 0.0))
        i = (C.x + o * j - A.x) / m;
    else {
        assert(!eq(n, 0.0));
        i = (C.y + p * j - A.y) / n;
    }

    assert(lteq(0, i) && lteq(i, 1));

    return Point(A.x + l1.x * i, A.y + l1.y * i);
}

Point intersectSegs(CLR l1, CLR l2) {
    double m = l1.x, n = l1.y, o = l2.x, p = l2.y;
    Point A = l1.p1, B = l1.p2, C = l2.p1, D = l2.p2;

    assert(!eq(n * o, m * p));
    double j = (n * A.x - m * A.y - n * C.x + m * C.y) / (n * o - m * p);
    double i = -1.0;

    if (!eq(m, 0.0))
        i = (C.x + o * j - A.x) / m;
    else {
        assert(!eq(n, 0.0));
        i = (C.y + p * j - A.y) / n;
    }

    assert(lteq(0, i) && lteq(i, 1));
    assert(lteq(0, j) && lteq(j, 1));

    return Point(A.x + l1.x * i, A.y + l1.y * i);
}
```

12.4 Projection

$\mathbf{a}'$ is the projection of $\mathbf{a}$ onto $\mathbf{b}$.

$$\mathbf{a}' = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}} \mathbf{b} \tag{12.1}$$

```
//Return the project of the given point on the given vector.
Point project(CPR p, CLR l) {
    Line AP(l.p1, p);
    Line vj = scalarProduct(l, dotProduct(l, AP) / length2(l));
    return Point(vj.p2 - vj.p1 + l.p1);
}
```

12.5 Reflection

$\mathbf{a}''$ is the reflection of $\mathbf{a}$ relative to $\mathbf{b}$.

$$\mathbf{a}'' = 2\mathbf{a}' - \mathbf{a} = 2\frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}} \mathbf{b} - \mathbf{a} \tag{12.2}$$

```
//Return the reflect of the given point relative to the given vector.
Point reflect(CPR p, CLR l) {
    Line AP(l.p1, p);
    Line vj(l.p1, project(p, l)); //the project of p on l.
    Line vr = scalarProduct(vj, 2); //the reflect of p relative to l.
    vr = vr - l;
    return Point(vr.p2 - vr.p1 + l.p1);
}
```

12.6 Area of polygon

The area of a polygon with vertices $(x_1, y_1), \ldots, (x_n, y_n)$ is equal to:

$$\frac{1}{2} \left\| \begin{array}{cccc} x_1 & x_2 & \ldots & x_n \\ y_1 & y_2 & \ldots & y_n \end{array} \right\|$$

where the determinate is defined to be similar to the 2 by 2 determinant:

$$x_1 y_2 + x_2 y_3 + \cdots + x_n y_1 - y_1 x_2 - y_2 x_3 - \cdots - y_n x_1$$

From here one can deduce that n given lines are collinear if the area of their polygon is zero.

```
double area(Polygon &g) {
    double a = 0.0;
    for (Polygon::Iterator iter = g.vertices.begin(); iter != g.vertices.end();
       ++iter)
        a += iter->x * g.next(iter)->y;
    for (Polygon::Iterator iter = g.vertices.begin(); iter != g.vertices.end();
       ++iter)
        a -= iter->y * g.next(iter)->x;
    return fabs(a) / 2;
}
```

12.7 Transformations

12.7.1 Rotation

Use $X' = R_\theta X$ to rotate the point X around the origin by an angle of θ, where

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

12.7.2 Line Reflection

Reflection matrix relative to the x-axis is $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. Reflection matrix relative to the y-axis is $\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$. Reflection matrix relative to the line $y = x$ is $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Reflection matrix relative to the line $y = -x$ is $\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$. Reflection matrix relative to the line $y = mx$ is $S = \begin{pmatrix} \cos 2\alpha & \sin 2\alpha \\ \sin 2\alpha & -\cos 2\alpha \end{pmatrix} = \frac{1}{1+m^2} \begin{pmatrix} 1-m^2 & 2m \\ 2m & m^2-1 \end{pmatrix}$.

12.7.3 Scale

$$K = \begin{pmatrix} k & 0 \\ 0 & k \end{pmatrix}$$

12.7.4 Perpendicular Projection

$$H = \frac{1}{2}(S+I) = \begin{pmatrix} \cos^2\theta & \sin\theta\cos\theta \\ \sin\theta\cos\theta & \sin^2\theta \end{pmatrix}$$

12.7.5 Amood Monasef

```
Line amoodMonasef(CLR l) {
    Point mid(0.5 * (l.p1.x + l.p2.x), 0.5 * (l.p1.y + l.p2.y));
    Point p;

    if (eq(l.y, 0))
        p = Point(mid.x, mid.y + 1);
    else if (eq(l.x, 0))
        p = Point(mid.x + 1, mid.y);
    else {
        double m = l.y / l.x;
        m = -1 / m;
        p = Point(mid.x + 1, mid.y + m);
    }

    return Line(mid, p);
}
```

12.8 Point in Triangle

To check if a point A is in a triangle, find another point B which is within the triangle (the average of the three vertices works well). Then, check if the point A is on the same side of the three lines defined by the edges of the triangle as B.

12.9 Convex Hull (Graham's scan)

```
struct P0Finder {
    bool operator()(CPR p1, CPR p2) {
        if (lt(p1.y, p2.y))
            return true;
        else if (eq(p1.y, p2.y))
            if (lt(p1.x, p2.x))
                return true;
        return false;
    }
};

struct CVSorter {
    Point p0;

    CVSorter(CPR p0) : p0(p0) {}
```

```
    bool operator()(CPR p1, CPR p2) {
        int relPos = relativePos(p1, Line(p0, p2));

        if (relPos == RIGHT) return true;
        if (relPos == ON && lt(dist2(p0, p1), dist2(p0, p2)))
            return true;
        return false;
    }
};

//One or Two points are not assumed to construct a convex hull. They should be
    considered separately if needed.
Polygon convexHull(vector<Point> &points) {
    assert(points.size() >= 3);
    vector<Point>::iterator p0Iter = min_element(points.begin(), points.end(),
       P0Finder());
    assert(p0Iter != points.end());
    Point p0 = *p0Iter;
    points.erase(p0Iter);
    sort(points.begin(), points.end(), CVSorter(p0));

    Polygon g;
    g.vertices.push_back(p0);
    g.vertices.push_back(points[0]);
    g.vertices.push_back(points[1]);

    for (int i = 2; i < (int) points.size(); ++i) {
        while (true) {
            if (g.size() < 2) break;
            Polygon::Iterator topIter = g.last(), nextToTopIter = g.prev(topIter);

            if (turnDir(*nextToTopIter, *topIter, points[i]) != LEFT) {
                if (g.size() > 0)
                    g.vertices.pop_back();
            }
            else
                break;
        }

        g.vertices.push_back(points[i]);
    }

    return g;
}
```

Chapter 13

Libraries

13.1 STL

=	==	!=	<	adjacent_find
advance	assign	back	base	begin
binary_search	capacity	clear	copy	copy_backward
count	count_if	empty	end	equal
equal_range	erase	fill	fill_n	find
find_end	find_first	find_first_of	find_if	find_last
for_each	front	generate	generate_n	includes
inplace_merge	insert	iter_swap	key_comp	lexicographical_compare
lower_bound	make_heap	make_pair	max	max_element
max_size	merge	min	min_element	mismatch
next_permutation	nth_element	partial_sort	partial_sort_copy	partition
pop	pop_back	pop_front	pop_heap	prev_permutation
push_back	push_front	push_heap	random_shuffle	rbegin
remove	remove_copy	remove_copy_if	remove_if	rend
replace	replace_copy	replace_copy_if	replace_if	resize
reverse	reverse_copy	rotate	rotate_copy	seacrh_n
search	set_difference	set_intersection	set_symmetric_difference	set_union
size	sort	sort_heap	splice	stable_partition
stable_sort	swap	swap_ranges	top	transform
unique	unique_copy	upper_bound	value_comp	

13.1.1 bitset

[]	&=	|=	^=	<<=	>>=	~	&	|	^	<<	>>
any	count	flip	none	reset	set	size	test	to_ulong			

13.1.2 string

+=	append	atoi	compare	erase
find	find_first_not_of	find_first_of	find_last_not_of	find_last_of
insert	push_back	replace	rfind	substr

13.2 Character Classification

isalnum	isalpha	iscntrl	isdigit	isgraph	islower	isprint	ispunct	isspace	isupper
isxdigit	tolower	toupper							

13.3 Streams

!	>>	<<	bad	clear	eof	fail	gcount	get	getline
good	ignore	put	rdstate	read	setstate	width	write		

13.3.1 ios-base

adjustfield	basefield	boolalpha	copyfmt	dec	fill	fixed
flags	floatfield	hex	internal	left	noboolalpha	noshowbase
noshowpoint	noshowpos	noskipws	nouppercase	oct	resetiosflags	right
scientific	setbase	setf	setiosflags	setprecision	showbase	showpoint
showpos	skipws	unitbuf	unsetf	uppercase		

13.3.2 File Streams

app	ate	binary	close	get	in	is_open	open
out	putback	readsome	seekg	tellg	trunc	unget	