The Hitchhiker's Guide to the Programming Contests

$\mathcal{D}$o $\mathcal{P}$anic

Nite Nimajneb

Contents

1 Dynamic Programming **1**
- 1.1 Subset Sum . . . 1
- 1.2 The Longest Increasing Subsequence . . . 1
- 1.3 Longest Common Subsequence . . . 2
  - 1.3.1 Reduce the Space to One Dimension . . . 3
- 1.4 Max Sum on a Line . . . 3
- 1.5 The Best Triangulation . . . 3
- 1.6 Counting or Optimizing Good Paths . . . 4

2 Graphs **5**
- 2.1 Breadth First Search . . . 5
- 2.2 Depth First Search . . . 6
  - 2.2.1 Topological Sorting . . . 7
  - 2.2.2 Strongly Connected Components . . . 7
  - 2.2.3 Cut Point . . . 9
- 2.3 Shortest Paths . . . 9
  - 2.3.1 Bellman-Ford . . . 9
  - 2.3.2 Floyd-Warshall . . . 10
  - 2.3.3 Dijkstra . . . 10
  - 2.3.4 The Shortest Path DAG . . . 12
- 2.4 Counting the Paths . . . 13
- 2.5 The Minimum Average Cycle . . . 14
- 2.6 The Minimum Spanning Tree . . . 15
- 2.7 Bipartite Matching . . . 17
- 2.8 Maximum Flows . . . 19
- 2.9 Minimum (Maximum) Cost Bipartite Matching . . . 21
- 2.10 Minimum Cost (Maximum Profit) Flow . . . 22

3 Numbers **22**
- 3.1 the Greatest Common Divisor . . . 23
- 3.2 Generating the Prime Table . . . 23
- 3.3 Repeated Squaring . . . 24
- 3.4 Long Integers using Character Arrays . . . 25
- 3.5 Fractional Numbers . . . 30

4 Algebra **31**
- 4.1 the Matrix Library . . . 31
- 4.2 the Method of Relaxation . . . 34

5 Geometry **35**
- 5.1 the Geometry Library . . . 35
- 5.2 Some Geometric Facts . . . 45

6 Miscellaneous **47**
- 6.1 Binary Search . . . 47
- 6.2 Range Query . . . 47
- 6.3 Set Union and Find . . . 48
- 6.4 String Matching . . . 49
- 6.5 2D Arrays, Rectangles . . . 50
  - 6.5.1 Cover the Chessboard with Rectangles . . . 50

6.5.2 2D Max Sum . . . 52
6.5.3 Max Unaffected Rectangle . . . 52

7 The Sample Programs **53**
7.1 Combinatorics . . . 53
7.2 Dynamic Programming . . . 56
7.3 Graphs . . . 59
7.3.1 BFS . . . 59
7.3.2 DFS . . . 61
7.3.3 Max Flow . . . 63
7.3.4 Min Cost Flow and Matching . . . 66
7.4 Long Integers . . . 71

1 Dynamic Programming

1.1 Subset Sum

Problem 1.1 *Given a set of n numbers a_i sum up to M, and any $K \leq M$, whether there is a subset of the numbers such that they sum up to (hit) K? We assume n might be as big as 1000, but M or K is not too big.*

From the Guide: We use a one dimensional table `m[0..M]`, `m[b]` indicate whether b can be hit.

Outline: Subset Sum

```
int m[M+10];

    for(i=0; i<M+10; i++) m[i]=0;
    m[0]=1;
    for(i=0; i<n; i++)
        for(j=M; j>=a[i]; j--)
            m[j] |= m[j-a[i]];
```

Remark. The original idea is to use a 2 dimensional array, where each column only depends on the previous column. By a programming trick we just need one column. But we need to write the j-loop in the reversed way to avoid messing things up.

There are a lot of variants of the subset sum problem.

- **Candy for two kids:** The `a[i]`'s are thought as value of candies. We want to divide the candies as evenly as possible between the two kids. Now the problem is not to hit a fixed K. We want to search a K such that it is as close to $M/2$ as possible. We may simple compute the m array, and look up which is the nearest "yes" below $M/2$.
- **Subset sum with multiple supplies:** Each a_i can be used as many times as you like in the sum, are we going to hit K? Maybe you will go a long way or a short way to see the solution. But finally the solution is just by reversing the direction of the j-loop in the subset sum program: `for(j=a[i]; j<=K; j++)`.
- **Coin change:** Now think a_i's are coins, you want to make an exact change of K. Maybe there are multiple ways you can do this, then you want to minimize (or maximize) the number of coins you use. The structure of the solution is not changed, we just need to change the meaning of the m array. Now `m[b]` is no longer 0 or 1, it is exactly the minimum number of coins we need to hit b.
- **Candy for three kids:** We want to divide the candies as evenly as possible among three kids. It is a question that what do we mean by "as evenly as possible". The answer is that one can define it in many different ways, but the structure of the solutions are almost the same: Use a two dimensional array `m[][]`, `m[b][c]` indicates whether or not we can divide the candies so that b of them goes to the first kid, c of them goes to the second (so the rest goes to the third).

1.2 The Longest Increasing Subsequence

Problem 1.2 *Given a sequence $s[0..n-1]$, a subsequence is gotten by pick any subset of the sequence, but not changing the order of them. What is the longest sequence such that each element is bigger than the previous one?*

From the Guide: 1 `m[i]` to be the length of the longest increasing subsequence q of `s[i..n-1]` such that q contains `m[i]` as its first element. To solve `m[i]`, guess what will be the next element in the best sequence.

From the Guide: 2 `m[i]` to be the length of the longest increasing subsequence q of `s[0..i]` such that q contains `m[i]` as its last element.

Both ways are good. But if they ask you to output the actual longest sequence, especially if they want the lexicographical smallest one among all the solutions, the first key is better.

Outline: $O(n^2)$ algorithm for The Longest Increasing Subsequence

```
for(i=n-1; i>=0; i--)
{
    m[i]=1;
    for(j=i+1; j<n; j++) if(a[j]>a[i])
        m[i] >?= m[j]+1;
}
ans = 0;
for(i=0; i<n; i++) ans >?= m[i];
```

There are three other versions: descending, non-descending, non-increasing. You just need to change a bit in the program.

The following improvement is really cute. It is a nice exercise to prove or to believe why it is correct.

Outline: $O(n \log n)$ algorithm for The LIS

```
set<int> st;
set<int>::iterator it;

        ...
        st.clear();
        for(i=0; i<n; i++)
        {
                st.insert(a[i]); it=st.find(a[i]);
                it++; if(it!=st.end()) st.erase(it);
        }
        cout<<st.size()<<endl;
```

1.3 Longest Common Subsequence

Problem 1.3 *Given two sequences* `s1[0..M-1]` *and* `s2[0..N-1]`*, what is the longest common subsequence of them?*

From the Guide: `m[i][j]` to be the length of the longest common subsequence of `s1[i..M-1]` and `s2[j..N-1]`. To solve `m[i][j]`, focus on the first step, if `s1[i]==s2[j]`, then we will pick them in our common sequence (why picking them is no worse than not picking, this requires a 10 seconds proof); otherwise, we must throw away at least one of them.

Outline: O(nm) algorithm for the LCS

```
for(i=M; i>=0; i--)
    for(j=N; j>=0; j--)
    {
        if(i==M || j==N) { m[i][j]=0; continue; }
```

```
            if(s1[i]==s2[j]) m[i][j] = 1+m[i+1][j+1];
            else m[i][j] = max(m[i][j+1], m[i+1][j]);
        }
    cout<<m[0][0];
```

Remark. When all the symbols in s_1 are distinct, the LCS problem can be reduced to the LIC problem. By renaming the elements of s_2, according to the order they appear in s_1, the LCS problem is the same as finding the LIS in s_2. So, when all elements in s_1 are distinct, the problem can be solved in $O((m+n)\log(m+n))$ time.

1.3.1 Reduce the Space to One Dimension

The dynamic programming is on a two dimensional table. Again, we see that any column only depends on the right neighboring column. At any moment two columns are enough. However, this is not as nice as the situation in subset sum where one column is enough.

This is a technique you may need in many situations. Here `ii = i&1` is the shadow of `i`. Every `m[x][y]` becomes `m[x&1][y]`.

Outline: O(nm) algorithm for the LCS with O(n) sapce

```
int m[2][1000]; // instead of [1000][1000]

    for(i=M; i>=0; i--)
    {
        ii = i&1;
        for(j=N; j>=0; j--)
        {
            if(i==M || j==N) { m[ii][j]=0; continue; }
            if(s1[i]==s2[j]) m[ii][j] = 1+m[1-ii][j+1];
            else m[ii][j] = max(m[ii][j+1], m[1-ii][j]);
        }
    }
    cout<<m[0][0]; // if you want m[x][y], write m[x&1][y]
```

1.4 Max Sum on a Line

Problem 1.4 *Given an array with n positive and negative numbers, find the subarray with one or more consecutive numbers where the sum of the subarray is maximum.*

It is trivial to do the problem in $O(n^3)$ time. An $O(n^2)$ algorithm is also very easy: Preprocess the information such that `s[i]` records the sum from the beginning to the i-th position. The best solution, hinted as the following, is $O(n)$.

From the Guide: Let `m[i]` be the maximum sum of any subarray that ends at the element `a[i]`. Then `m[i]` is simply `max(a[i], m[i-1]+a[i])`.

1.5 The Best Triangulation

Problem 1.5 *Given a convex polygon $v_0v_1\cdots v_{n-1}$, a triangulation is a way of picking $n-3$ non-crossing diagonals so that the polygon is divided into $n-2$ triangles.*

The number of all possible triangulations can also be computed by dynamic programming, but we know it is the classical Catalan number $\binom{2n-4}{n-2}/(n-1)$.

Our question here is, among so many triangulations, which is the best one? The word "best" can be defined in many ways. Let say best means the sum of the lengths of the diagonals we pick is as small as possible.

From the Guide: Look at the edge v_0v_{n-1}, in any triangulation it is contained in exactly one triangle. Guess what will be the other point of that triangle. Let `m[a][b]` ($a < b$) be the value of the best triangulation of the (part of) polygon $v_av_{a+1}\cdots v_bv_a$.

This time, the solution of the longer segments depends on the shorter segments.

Outline: Best Triangulation

```
m[..][..] = 0;
for(len=3; len<n; len++)
for(a=0; a+len<n; a++)
{
    b=a+len; m[a][b]=1e+10;
    for(c=a+1; c<b; c++)
    {
        double t=m[a][c]+m[c][b];
        if(c>a+1) t+=length(a to c);
        if(c<b-1) t+=length(c to b);
        m[a][b] <?= t;
    }
}
```

1.6 Counting or Optimizing Good Paths

In an $n \times m$ grid, we want to go from the left bottom corner to the upper right corner. Each time we can only take a step to the right, or a step up. The number of ways we can do this is exactly $\binom{n+m}{n}$. But what if we forbid some points on the grid? For example, if we forbid all the points above the line $y = x$. Some of the problems has answer in closed formula. But all of them can be solved quickly by dynamic programming.

Problem 1.6 *Given a directed acyclic graph, how many paths are there from u to v? What is the longest one if there are weights on the edges?*

From the Guide: Do a topological sort (Section 2.2.1) on the graph, then do a DP along the topological order.

Remark. You **do not** need to do the DFS (topological sort) and DP separately. Just do the DP in recursive manner. The situation where the recursive DP is good is exactly when the order of the DP is hidden.

Remark. The longest increasing subsequence problem is a special case of the longest path in a DAG.

The longest path in an undirected graph is not well defined, if we allow repeated vertex on a path; and it is well defined but extremely hard. A naive algorithm checks all possible (at most will be $n!$) paths. We are going to improve, roughly, $n!$ to 2^n.

Problem 1.7 (the Traveling Salesman Problem) *In a graph G, a salesman start from the vertex 0, visit all the other vertices at least once and come back to the starting vertex. What is the fastest of such a tour?*

From the Guide: `play(a, S)` ($a \in S$) solves the question of the fastest route start from a, visit each point in S at least once, and go back to 0. To solve it, we focus on what will be the next step. If that is i for some $i \in S$, then we have $d(a, i) + \text{play}(i, S \setminus \{a\})$, where $d(a, i)$ is the shortest path from a to i.

The complete code is in Problem 7.4.

Another famous problem that is very similar is called the *Hamilton cycle problem*, where we require each vertex be visited exactly once. The question is whether there exist Hamilton cycles. If yes, what is the shortest one? On general graphs, this question is slightly simpler than the TSP problem — we just need to change the $d(a, i)$ to $r(a, i)$, i.e., we use the length of an edge instead of the shortest path.

Problem 1.8 *In a 0-1 matrix, how many ways we can pick n 1's such that each row and each column contains exactly one picked element?*

This is the same problem as

Problem 1.9 *On a $n \times n$ chess board we want to put n rooks such that no two rooks can attack each other. In how many ways we can do this? Well, $n!$. But what if we forbid some squares on the board?*

From the Guide: `play(a, S)` ($|S| = a$), where S is a set of columns, solves count how ways we can put rooks in the first a rows such that the set of column occupied is exactly S.

What if we want to put k rooks instead of n rooks? Go through the problem again, the key is almost the same, but we do not require $|S| = a$.

2 Graphs

Usually n is the number of vertices, and m (although most likely only for discussion purposes) is the number of edges. We usually use `r[][]` to represent the (adjacency matrix of the) graph, it is a 0-1 matrix if the graph is not weighted and simple, it is symmetric if the graph is undirected, etc. In some algorithms, like Floyd, we start with `r[][]` as the adjacency matrix, then it evolves to the matrix represent the all-pairs distances.

2.1 Breadth First Search

Rather than vertices and edges, we call them *states* and *transitions*. The main reason is that most examples of BFS we see are on implicit graphs, where the nodes have some inner structure and the edges are generated by transition rules.

The `struct ND` contains the information about the node, as well as information in the BFS:

```
struct ND
{
    ... state // information of the state
    int d; // the depth. i.e. the distance from the start state
    int f; // parent pointer
};
```

The array `mac[]` and pointers `born` and `dead` realizes the queue for BFS. The queue is always the nodes from `mac[dead]` to `mac[born-1]`, inclusive. The predicate `visited` can be realized by a 0-1 indicator array, a map, or a set. In the first two cases we may combine `ND.d` and `visited` together.

Outline: BFS

```
ND mac[??];
int born, dead;
int visited[??];
```

```
int BFS()
{
  ND nd, nnd;
  Initialize visited to be no; born = dead = 0;
  mac[born] = start; born++; // there might be several start points
  visited[start] = yes;
  while(dead < born)
  {
    nd = mac[dead];
    if(nd.state == goal) {output; return 1;}
    for each transition nd.state --> v
        if(!visited[v])
        {
            nnd.state = v; nnd.f = dead; nnd.d = nd.d+1;
            mac[born] = v; born++; visited[v] = yes;
        }
    dead++;
  }
  output "no solution"; return 0;
}
```

2.2 Depth First Search

Outline: Basic DFS

```
void dfs(int a) // dfs on node a
{
    if(a is not a vertex at all) return;
    if(v[a]) return;
    v[a] = 1;
    for each neighbor a->i dfs(i);
}
```

`dfs(a)` will mark (visit) all the vertices that are reachable from a. In case of undirected graphs, this means the component containing a. To find the number of components in a undirected graph, as well as mark the components with a representative using `v`, we do

Outline: Number of Components of an Undirected Graph

```
int rep, v[..];

void dfs(int a) // dfs on node a
{
    if(v[a] >= 0) return;
    v[a] = rep;
    for each neighbor a->i dfs(i);
}

    ...
    int cpt=0;
    for(i=0; i<n; i++) v[i]=-1;
```

```
for(i=0; i<n; i++) if(v[i]<0)
{
    cpt++; rep=i;
    dfs(i);
}
...
```

The so called DFS tree (or DFS forest) is the diagram based on the recursive calls of the DFS. A *tree edge* $a \to b$ means `dfs(a)` is the first one to call `dfs(b)`. The children of a node is positioned from left to the right, according to the order they are called. All the other edges are dotted around the tree, they are discovered but never actually traveled. $a \to b$ can be classified as *back edge* if b is an ancestor of a in the tree; *forward edge* if b is a descendant of a; and *cross edge* otherwise. Many algorithms are depend on some important observations about the DFS tree. For example: If the graph is undirected, there will never be a cross edge in the DFS tree. If the graph is directed, there will never be cross edges from left to right.

2.2.1 Topological Sorting

Let G be a directed acyclic graph (DAG). We can always order the vertices in a line so that all the edges are going from the left to the right. This is called a *topological order*. Using a colorful DFS we can find the topological order, or report that the input graph is not a DAG. The basic observation is that if we write down the vertices in the reversed order when we finish the DFS on that node, we get a topological order; and there is a cycle if and only if we see a back edge in DFS.

Outline: Topological Sort

```
int fg, v[..];
int od[..], p; // the final order

void dfs(int a) {
    if(v[a]==1) fg = 1;
    if(v[a]) return;
    v[a] = 1; // gray
    for each neighbor a->i dfs(i);
    v[a] = 2; // black
    od[p] = a; p--;
}

    ...
    for(i=0; i<n; i++) v[i]=0; //white
    fg=0; p=n-1;
    for(i=0; i<n; i++) if(!v[i])
        dfs(i);
    if(fg) say there is a cycle;
    ...
```

2.2.2 Strongly Connected Components

Two vertices a and b are in the strongly connected component if they can be reached from each other. It is easy to see that the strongly connected components is a partition of the vertices. The super graph of G is an animal such that each vertex represents a strongly connected component in G, and $C \to D$ is there are vertices $u \in C$ and $v \in D$ such that $u \to v$ is an edge in G. It is easy to see that the super graph is a DAG.

If we want to do a DFS to find the strongly connected components. We would be lucky if we start from a component in the supergraph where there is no out edge. We would be very unlucky if we start from a

component that can reach a lot of other components. The SCC algorithm is basically two DFS, the first DFS do a rough topological sort to ensure the second DFS will be lucky. In the second DFS, instead of `for(i=0; i<n; i++) if(!v[i]) dfs(i);`,

we go according to the order given by the first DFS. There are more details. The complete program is

Outline: Strongly Connected Components

```
const int N=1000, M=1000; //Is N big enough?
int r[N][N]; // adjacency matrix
int order[N], v[N], top_place;
int comp[N]; // component id
int curr_comp;


void dfs(int a) {
  if (v[a]) return;
  v[a] = 1;
  for (int i = 0; i < N; i++)
    if(r[a][i]) dfs(i);
  top_place--;
  order[top_place] = a;
}


// dfs on the reversed graph
void dfs_rev(int a) {
    if(v[a]) return;
    v[a] = 1;
    comp[a] = curr_comp;
    for(int i=0; i<N; i++) if(r[i][a]) dfs_rev(i); }

void top() {
    int i;
    top_place = N;
    for(i=0; i<N; i++) v[i]=0;
    for(i=0; i<N; i++) if(!v[i]) dfs(i);
}

void scc() {
    int i;
    top();
    for(i=0; i<N; i++) v[i]=0;
    for(i=0; i<N; i++)
    {
        int j = order[i];
        if (!v[j]) {
        curr_comp = j;
        dfs_rev(j);
    }
  }
}
```

2.2.3 Cut Point

Based on the observation that there is no cross edges in the DFS tree. So, assume v is a node at level a, and T is a subtree of v, after deleting v, the only chance to save T's connection to the rest of the world is a back edge from T to somewhere higher than a. We may detect the cut-points as well as how many components are created if we cut a certain point. The complete code is in Problem 7.6.

You need to pay attention to how the cut-points are defined if the input graph is already disconnected.

2.3 Shortest Paths

2.3.1 Bellman-Ford

The Bellman-Ford algorithm solves the single source shortest path problem in time $O(nm)$.

Outline: Bellman-Ford

Assuming the start point is vertex 0.

```
int d[..]; // or double?

initialize d[.]=infinity, d[0]=0;
while(still changing)
{
    for each edge u->v
        d[v]<?=d[u]+cost(u,v);
}
```

For any node v, let rank v be the number of stops on the shortest path (if there are more than one shortest path, pick the one with least number of stops etc.). Let R be the maximum rank in the graph. (i.e., all the vertices are within level $\leq R$ neighbors of 0.) We can prove that the `while` loop will be executed at most R times, so, if you know r to be an overestimate of R, the while loop can be simply written as
`for(i=0; i<r; i++)`

Clearly, if there is no negative cycle reachable from 0, n is a overestimate of R. So in some problems (if $O(nm)$ is not too slow), we simply write
`for(i=0; i<n; i++)`

If there is a negative cycle that is reachable from 0, the shortest path is not well defined (at least for some vertex). To detect it, we just need to do the loop one more times

Outline: Bellman-Ford to detect a negative cycle from 0

```
int d[..]; // or double?

initialize d[.] = infinity, d[0] = 0;
for(i=0; i<n+1; i++) // or i<r+1
{
    change = 0;
    for each edge u->v
    if(d[v]>d[u]+cost(u,v))
    {
        change=1;
        d[v]=d[u]+cost(u,v);
    }
}
if(change) output "negative cycle reachable from 0";
```

Even if there is a negative cycle from 0, for a particular v, it is still possible that the shortest path from 0 to v is well defined. `d[v]` is not well defined (or $-\infty$) if and only if there is a negative cycle C such that $0 \to C$ *and* $C \to v$. To detect this a simple Bellman-Ford is not enough. This can be done very neatly, see the next section.

2.3.2 Floyd-Warshall

Solves the all-pairs shortest path problem in $O(n^3)$ time.

```
construct r[][] to be the weighted graph
r[i][j] = infinity if there is no i->j edge;
r[i][i] = 0 for each i;

for(k=0; k<n; k++) for(i=0; i<n; i++) for(j=0; j<n; j++)
  r[i][j]<?=r[i][k]+r[k][j];
```

At the end `r[i][j]` is the distance from i to j.

A simple cycle means a cycle without repeated vertices. The shortest path from 0 to v is not well defined if and only if

$$\text{there is a simple negative cycle } C \text{ such that 0 can reach } C \text{ and } C \text{ can reach } v. \tag{1}$$

Bellman-Ford can only detect if there is a negative cycle reached by 0, but that cycle not necessary reach v. Even we iterate many more times, we are not promised to decide if (1) happens.

There are several solutions to (1). We can either use a Bellman-Ford combined with a reversed Bellman-Ford from v, or Bellman-Ford combined with a reversed DFS. But the best solution is by Floyd. Since Floyd has the nice properties:

- After n loops, `r[i][j]<infinity` iff there is a path from i to j.
- If i is involved in a simple negative cycle, then after n loops `r[i][i]<0`. (The converse is not necessarily true.)

Based on these properties, we can solve complete the shortest path problem with possible negative cycles.

Outline: the Ultimate Shortest Path Algorithm

```
construct r[][], r[i][j] = infinity if there is no i->j edge;
r[i][i] = 0 for each i;

for(k=0; k<n; k++) for(i=0; i<n; i++) for(j=0; j<n; j++)
  r[i][j]<?=r[i][k]+r[k][j];
if(r[i][j] is infinity) return "infinity" // no path
for(k=0; k<n; k++) if(r[k][k]<0 && r[i][k]<infinity && r[k][j]<infinity)
  return "-infinity";
return r[i][j];
```

2.3.3 Dijkstra

One important restriction of this algorithm is that it only works for graphs with non-negative weights. The running time is $O(n^2)$. With a little effort it can be made into $O(m \log n)$

It starts from the source vertex, in each stage there will be one new vertex marked. All the marked vertices consists of the civilized world, and the rest of the vertices consists on the unknown world. In each

stage, we examine all the edges go from the civilized world to the unknown, to update the estimations of the distance of vertices in the unknown world. The one with the smallest estimation is the next vertex to mark.

Outline: Dijkstra in $O(n^2)$

We assume the source is s. We use `d[]` to store the estimation of distances, at the end they will be the actual distances. We use `mk[]` to see if a vertex is marked or not.

```
d[v] = infinity-1 for all v; d[s] = 0;
mk[v] = 0 for all v;
loop n times
    mx = infinity;
    for (i=0; i<n; i++) if(!mk[i] && d[i]<mx)
        { mx =d[i]; next = i; }
    mk[next] = 1;
    for (i=0; i<n; i++) if(next to i is an edge)
        d[i] <?= d[next] + cost(next to i);
```

And here is the code in $m \log n$ time.

Outline: Dijkstra in $O(m \log n)$

```
typedef weight int; // double?
int n; // n nodes
vector<int> r[..n+10]; // r[i][j]: the j-th neighbor of i
vector<weight> e[..n+10]; //e[i][j]: the length of edge i->r[i][j]
weight dist[..n+10];
int pa[..n+10];
multimap<weight, int> h;

void init()
{
    int i;
    n = ?;
    for(i=0;i<n;i++)
    {
        r[i].clear();
        e[i].clear();
    }
    // read the graph???
    for(i=0;i<n;i++) dist[i]=-1;
}

// In the tree h, <weight, int> : <candidate distance, node id>
void dijkstra(int S)
{
  weight d, tmp;
  int v, i, j;
  multimap<weight, int>::iterator it;
  h.clear();
  dist[S]=0;
  pa[S]=-1;
  h.insert(multimap<weight, int>::value_type(0, S));
```

```
  while(!h.empty())
    {
      it=h.begin();
      v=(*it).second; // the node
      d=(*it).first; // the distance
      h.erase(it);
      for(i=0;i<r[v].size();i++)
      // for each neightbor of v
      {
        tmp=d+e[v][i];
        j=r[v][i];
        if(dist[j]<0 || tmp<dist[j])
        {
          dist[j]=tmp;
          pa[j]=v;
          h.insert(multimap<weight, int>::value_type(tmp, j));
        }
      }
    }
}
```

2.3.4 The Shortest Path DAG

Let G be a graph (for simplicity, we assume there is no negative cycles in G). The shortest path DAG (rooted at 0) is a subgraph P on the same vertices, and $u \to v$ is an edge of P if and only if

$$d[v] = d[u] + \text{cost}(u, v) \tag{2}$$

i.e., one of the shortest paths from 0 to v goes through u. In this case we call u a parent of v. In all of our shortest path algorithms, after we find the actual distances, it is easy to find one or all of v's parents by (2). (Well, v might have more than 2 parents...) Many problems are based on Topological sort of a DAG plus dynamic programming. For example, there might be many shortest paths from 0 to v, if we want to count how many of them, we can do a dynamic programming on the shortest path DAG.

Everything can be reversed. Bellman-Ford and Dijkstra solves the single source shortest path problem. By modify a little bit they can solve the single destination shortest path problem. For example, if we want to find the lexicographical smallest among all the shortest paths from 0 to v, we just need to define `dd[u]` to be the distance from u to v, and

Outline: Finding the Lexicographical Smallest Shortest Path

```
    solve dd[] with any algorithm;

    int st=0;
    while(st != v)
    {
        cout<<st<<" ";
        for(i=0; i<n; i++) if(dd[st]=dd[i]+cost(st, i))
        { st=i; break; }
    }
    cout<<v<<endl;
```

2.4 Counting the Paths

Problem 2.1 *Let G be a graph (directed or undirected, weighted or un-weighted). How many paths are there from u to v that takes exactly t steps?*

From the Guide: Write down the adjacency matrix A, where A_{ij} is the number of edges from i to j. Let A^t be the usual matrix multiplication of A t times. The (u, v) entry in A^t_{uv} is exactly the answer we are after. (Exercise: Check for $t = 0, 1, 2$.) If t is big, we can do the standard repeated squaring (Section 3.3).

Let us look at the same thing from a dynamic programming view point: `A[t][i][j]` be the number of paths from i to j with exactly t steps. To compute `A[t][i][j]`, we focus on which will be the first step:

$$\texttt{A[t][i][j]} = \sum_{k=0}^{n-1} \texttt{A[1][i][k]} \cdot \texttt{A[t=1][k][j]}$$

Problem 2.2 *Let G be a graph (directed or undirected, weighted or un-weighted). Among all the paths from i to j that takes exactly k steps, which one is of the shortest length?*

Now we just need to modify the meaning of `A[t][i][j]`, and the recurrence becomes

$$\texttt{A[t][i][j]} = \min_{k=0}^{n-1}(\texttt{A[1][i][k]} + \texttt{A[t-1][k][j]})$$

If you like to think in the algebraic way, we just redefined the rule of matrix multiplication. Be aware that if you use something like 1000000000 as infinity, and there are negative edges, then you need to some tolerance about what should be considered real infinity. (For example, if all the edges are less than 1000, and $k < 2000$, then anything as large as $1000000000 - 2000000$ can be considered as infinity.)

In the outline, we also construct the actual path. `p[t][i][j]` records what is the first step on the best path from i to j in t steps.

Outline: Shortest Path with Number of Steps Given

```
int BG=1000000000;
// to avoid overflow in addition, do not use 2^31-1

    A[1][i][j] = r[i][j]; p[1][i][j]=j;
    for(t=2; t<=n; t++)
    for(i=0; i<n; i++) for(j=0; j<n; j++)
    {
        A[t][i][j]=BG; p[t][i][j]=-1;
        for(k=0; k<n; k++) if(A[1][i][k]<BG && A[t-1][k][j]<BG)
            if(A[1][i][k]+A[t-1][k][j] < A[t][i][j])
            {
                A[t][i][j] = A[1][i][k]+A[t-1][k][j];
                p[t][i][j] = k;
            }
    }
```

to output the best path from a to b with t steps:

```
void output(int a, int b, int t)
{
    while(t)
    {
```

```
        cout<<a<<" ";
        a = p[t][a][b];
        t--;
    }
    cout<<b<<endl;
}
```

2.5 The Minimum Average Cycle

Problem 2.3 *G a directed graph with weights w. Decide if G contains a cycle or not. If there are cycles, find the cycle with minimum average weight. (If C is a cycle consists of edges $e_1, \cdots, e_t$, the average weight is $(\sum_i w(e_i))/t$.)*

The solution is an application of the previous section. `A[t][x][x]` gives the best cycle of length t that involves x. The minimum average cycle is the one with minimum `A[t][x][x]/t`. To output the best cycle is just to output the path from x to x in t steps, as use outlined in the previous section.

The running time of the above algorithm is $O(n^4)$. The feeling is that we are doing all pairs shortest path. If we can reduced it to single source (or single destination) problem, maybe the running time can be reduced by a factor of n. This is the right intuition:

We add a point n to the graph, and there is an edge to n from each vertex with weight 0. (But no edge comes out of n.) Now we want to compute `A[t][i]` to be the shortest path from i to n with exactly t steps, and `p[t][i]` to be the first step on such a path.

Outline: Minimum Average Cycle

```
int BG = 1000000000;

    A[0][i] = BG for all i; A[0][n]=0;
    for(t=1; t<=n+1; t++)
        for(i=0; i<=n; i++)
        {
            A[t][i] = BG;
            for(j=0; j<=n; j++) if(r[i][j]<BG && A[t-1][j]<BG)
            if(r[i][j]+A[t-1][j] < A[t][i])
            {
                A[t][i] = r[i][j]+A[t-1][j];
                p[t][i] = j;
            }
        }
    double ans=1e+15; int st;
    for(i=0; i<n; i++) if(A[n+1][i]<BG)
    {
        double tmp = -(1e+15);
        for(t=0; t<=n; t++) if(A[t][i]<BG)
            tmp>?=1.0*(A[n+1][i]-A[t][i])/(n+1-t);
        if(tmp<ans) {ans=tmp; st=i;}
    }
    the min average is ans, and start from st.
```

To output the actual cycle we follow the p links, but this time we do not know the actual length.

```
void output(int st)
```

```
{
    int t=n+1;
    int wk=st;
    do {
        cout<<wk<<" ";
        wk=p[t][wk]; t--;
    }while(wk!=st);
    cout<<st<<endl;
}
```

2.6 The Minimum Spanning Tree

If a graph G is connected, there might be many spanning trees. We have the celebrated

Theorem 2.1 (the Matrix-Tree Theorem) *Suppose G is a graph without self-loop (there might be multiple edges between two vertices, but no edge come from a vertex to itself.). Let M be the matrix where on the diagonal we have the degrees, i.e., M_{ii} is the degree of v_i in G; and off diagonal we have, for $i \neq j$, the negative number of edges, i.e., $M_{ij} = M_{ji}$ is the negation of the number of edges between v_i and v_j. Let M' be a matrix by deleting any row and any column from M. Then the number of spanning trees of G equals the absolute value of the determinant of M'.*

The theorem gives an algorithm to compute the number of spanning trees. As a special case, we have Cayley's formula.

Theorem 2.2 (Cayley's Theorem) *The number of spanning trees of a complete graph K_n is n^{n-2}.*

Now, there are many spanning trees. (For $n \approx 16$, you need a `long long` type to hold the answer.) If the graph is weighted, we want to find the one with the minimum weight. There are two natural greedy algorithm works.

The Prim algorithm resembles Dijkstra. It starts from a particular vertex v. We always focus on the component we have for v, in each stage there will be one new vertex connected to the component. All the marked vertices consists of the civilized world, and the rest of the vertices consists on the unknown world. In each stage, we examine all the edges go from the civilized world to the unknown. The smallest such edge will be chosen, and the other end of this edge will be marked.

Outline: Prim's MST Algorithm

```
// Prim start at node a. in O(n^2).
// Return 0 if not connected
// In pa[] store the parent on the MST.

typedef weight int; //double?
const weight  maxWeight=weight(1e9) ; //big enough?
weight r[][];
weight d[];
int n, pa[];

int mst(int a)
{
    int i,j,k,mini,tj,tk;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            r[i][j]=-1;
```

```
    // Construct graph **symmetric [i][j]->[j][i]
    int col[?]; // color 0 means not in the tree
    for(i=0;i<n;i++) {col[i]=0; d[i]=maxWeight; pa[i]=-1;}
    d[a]=0;
    //col[a]=1;
    //pa[a]=-1;
    for(i=0;i<n;i++)
    {
        mini=maxWeight;
        for(j=0;j<n;j++)
            if(col[j]==0 && mini>d[j])
              { mini=d[j]; tj=j; }
        if(mini==maxWeight) return(0);
        col[tj]=1;
        for(j=0;j<n;j++) if(col[j]==0)
            if(r[tj][j]<d[j])
              {
                d[j]=r[tj][j];
                pa[j]=tj;
            }
    }
    return(1);
}
```

In Kruskal's algorithm, we start from the empty graph, viewed as n isolated components. We want to throw the edges back to connect them. The key is that we throw the edges back one by one, ordered by their weight. We use an edge if it connects two different components, otherwise we do not need it.

Outline: Kruskal's MST Algorithm

```
struct eg { int x,y,d; }; // double for d?

eg e[250];
int u[250];
int f[200];

bool operator <(const eg& e1, const eg& e2)
{ return (e1.d<e2.d); }

int findg(int a)
{
    int st[200]; int pc=0;
    while(a!=f[a])
    {
        st[pc]=a;pc++;
        a=f[a];
    }
    while(pc) {pc--; f[st[pc]]=a;}
    return(a);
}

int mst()
```

```
{
    int i,x,y;
    for(i=0;i<n;i++) f[i]=i;
    int egs=0;
    for(i=0;i<m;i++)
    {
        x=findg(e[i].x);
        y=findg(e[i].y);
        if(x!=y)
        {
            u[i]=1;f[x]=y; egs++;
            //if time exceed, use a rank function
        }
        else u[i]=0;
    }
    if(egs==n-1) return(1);
    return(0);
}

... sort the edges; call mst();
```

In the program we need a nice way to bookkeeping the components. The issue is called find-union, which is the topic of Section 6.3.

2.7 Bipartite Matching

A graph G is bipartite if we can partition the vertices to red (left) and blue (right), such that each edge has end points in both part.

Theorem 2.3 *A graph G is bipartite if and only if there is no cycle in G of odd length.*

It is easy to detect if a graph G is bipartite: Run a DFS with a parameter indicating the depth, there is an odd cycle iff you find some vertex who is discovered before with the depth of a different parity.

If G is bipartite, there might be more than one way to partition the vertices into red and blue (left and right). In fact, the number of such partitions is exactly 2^t, where t is the number of connected components of G.

The next question is the *maximium bipartite matching problem.*

Problem 2.4 (Biaprtite Matching) *Suppose we are given a bipartite graph G as well as its bipartition. $G = (R, B, E)$, where R is the set of red (left, boy) vertices, B is the set of blue (right, girl) vertices, and E is the set of edges. A* matching *is a set of edges so that no vertex is paired more than once. The size of the matching is the number of pairs. We are interested in finding the maximum sized matching. i.e., we want to pair them up as many as we can.*

For bipartite graphs, we still use a 2D array `r[][]`. But now it is not necessarily square. `r[i][j]` is not the relation between v_i and v_j; now it gives information between r_i and b_j (the i-th boy and the j-th girl).

The algorithm for finding the max bipartite matching starts from the empty matching, and grow it bigger and bigger. The way we grow it is by finding an augmenting path. An *augmenting path* for a matching M is a path of the form $r_0 b_1 r_1 b_2 r_2 \cdots b_k r_k b_{k+1}$, such that all the edges on the path are in $E(G)$, all the edges $b_i r_i$ is in the matching M, and neither r_0 nor $bk + 1$ is matched in M. If we have such a path, we can switch the pairing to get one more pair in our matching. The nice theorem says that the converse is also true.

Theorem 2.4 *A matching M is of maximum size iff there is no augmenting path for M.*

The program below finds the maximum matching, moreover, it gets the max matching with lexicographical smallest set for the left part if not all the left part can be satisfied.

Outline: Bipartite Graph Maximum Matching

In the variables below, we assume 110 is the upper bound for both left set and right set. The sizes may vary.

```
int r[110][110];
int N,M;
int v[110];
int m[110], m1[110];
```

To use the code, you need to set up N to be the number of vertices in the left set, M to be the number of vertices in the right set, and `r[i][j]` to be 1 if there is an edge from the i-th left vertex to the j-th right vertex. Then, call `bipMatch()`, it will return the size of the max matching. And after the call the array element `m[i]` indicates the partner of the i-th left vertex (-1 means it is not matched to anyone).

```
int dfs(int a)
{
        if(a<0) return(1);
        if(v[a]) return(0);
        v[a]=1;
        int i;
        for(i=0;i<M;i++) if(r[a][i]) //* see remark
        {
                if(dfs(m1[i]))
                {
                        m[a]=i;m1[i]=a;
                        return(1);
                }
        }
        return(0);
}

int dfsExp(int a) {
        int i;
        for(i=0;i<N;i++) v[i]=0;
        return dfs(a);
}

int bipMatch()
{
        int i;
        int ans=0;
        for(i=0;i<N;i++) m[i]=-1;
        for(i=0;i<M;i++) m1[i]=-1;
        for(i=0;i<N;i++) if(m[i]<0) ans+=dfsExp(i);
        return(ans);
}
```

Remark. The line with a `*` loops over all the possible partners of the a-th left vertex. We have some freedom here. For example, if in some applications `double r[][]` represents the distances, and a match is

possible if the distance is no bigger than a threshold D, then the line becomes

```
for(i=0;i<M;i++) if(r[a][i]<D+EPS) // for some small error EPS
```

In some applications we need linked list representation of the graph, since the graph is sparse and we could not afford a matrix, then we do not use `r[][]`, instead define something like `vector<int> r1[1010]`. And the loop becomes

```
for(j=0, int l=r1[a].size();j<l;j++)
{
        i=r1[a][j];
        ...
}
```

Matching can be defined on general graphs: A set of edges such that no vertex is touched (paired) more than once. In a general graph, the maximum matching is still computable in polynomial time, but it is much more complicated than the bipartite case.

A set S of vertices is called a *vertex cover* for G if each edge of G contains at least one end point in S. Consider the vertices and edges are the map of a city, you want to pick several intersections to put police so that every road is guarded. The minimum vertex cover is the least number of polices you need to put in order to guard all the roads. It is easy to see the minimum vertex cover number is at least the maximum matching number, since no vertex can guard two edges in a matching. In general, these two numbers are not equal. And the vertex cover number is likely to be very hard – the problem is NP-complete. Nevertheless, we have a nice theorem in the bipartite case.

Theorem 2.5 *In a bipartite graph G, the vertex cover number is the same as maximum matching number.*

2.8 Maximum Flows

Problem 2.5 (Max Flow) *Given a graph G, a source s and a destination t, and on each edge e, there is a capacity $c(e) \geq 0$, which is the maximum amount of substance we can transfer along that edge. (We work with directed graphs, so $c(u, v)$ is not necessarily the same as $c(v, u)$.) What is the maximum amount of substance we can transfer from s to f through the whole graph?*

Problem 2.6 (Min Cut) *Given a graph G, a source s and a destination t, and on each edge e, there is a cost $c(e) \geq 0$. We want to delete some edges so that t is disconnected from s. (i.e., there is no directed path from s to t; but there might be path from t to s). What is the minimum cost of such a cut?*

In a weighted graph G with weights $c(e)$, we view c as capacity in the max flow problem, and view c as the cost in the min cut problem. It is easy to see that any flow cannot transfer more than the cost of any cut. So max flow is at most the min cut. The magic theorem says that they are actually equal.

Theorem 2.6 (max flow min cut) *In any weighted graph, max flow = min cut.*

As a convention, we focus on the problem of finding the max flow (the amount as well as the actually flow). We will find the cut that achieves the min cut as a by-product of our max flow algorithms.

Before we describe the algorithms, we show some applications of max flow. First, we have the following theorem that allows us to work on discrete units instead of continuous substance.

Theorem 2.7 *If G is a graph where all the capacities are integers, then among all the possible max flows, there is one where the amount of substance flowing on each edge is integer.*

In fact all our algorithm will give such an integer flow if the input capacities are integers.

Application. If we have a bipartite graph $G = (A, B, E)$, think of A is the vertices we draw on the left, B on the right side. We add a leftmost point s and rightmost point t, and add edges between s and each point in A, and t to each point in B. (We may direct each edge from left to the right, or just allow both directions.) And we think each edge has capacity 1, then the max matching problem becomes a special case of the max flow problem.

Application. If we have a graph G and two vertices s and t. We want to know at most how many edge-disjoint paths (meaning no edge can be used on two paths, but a vertex may be used several times) from s to t we can find. A nice theorem in graph theorem tells us that this is the same as the question of at least how many edges we need to cut off so that s and t are disconnected. This is nothing but the min cut problem where we think each edge has both direction with capacity 1. So we can solve this by a max flow.

Application. How about the restrictions (the capacities) are on the vertices instead of edges? Aha, here is the beautiful construction: We draw the graph, and draw it again on a slightly higher level, say, second floor. Now, each node v has two copies, one on each floor. We call the one on the first floor v_{in} (the entrance of v), and the one on the second floor v_{out} (the exit). Now we redraw the edges, if there was an edge $u \to v$, we add an edge $u_{out} \to v_{in}$, with capacity big enough, say, bigger than the max capacity on any vertex in the original graph. At last, for each vertex v in the old graph, we add in the new graph an edge $v_{in} \to v_{out}$ with capacity exactly the same as the capacity on v in the old graph. By doing this, we simulated the flow problem with vertex capacities with a usual one where the capacities are on edges. The answer we want is the max flow from s_{in} to t_{out}.

Application. Given a graph G and two vertices s and t, we want to know at most how many vertex-disjoint paths from s to t we can find. This is just a special case of the previous application, where we think each vertex except s and t has capacity 1. See Problem 7.7 for an example.

Outline: the Edmonds-Karp Algorithm for Max Flow

The basic algorithm for max flow is simple. We star from the empty flow, the residue capacity is $c(e) - f(e)$, meaning how many capacity left on an edge. (Note that if we flow along an edge $u \to v$ with flow x, the capacity $c(v, u)$ is also changed, it is increased by x. We may think a flow of x from u to v is also a flow of $-x$ from v to u. So, even originally the capacity of (v, u) is 0, it may become positive in our algorithm.)

At any stage we have a flow f, the residue network is the graph consists of all the edges with positive residue capacity. A path P from s to t in the residue network is call an augmenting path for f. Clearly, we can transfer x more substance from s to t along the augmenting path, where x is the minimum residue capacity on the path P. (Finding a path is easy, use DFS, BFS, or even Bellman-Ford.)

We do this again and again, until the residue network becomes disconnected. (t is not reachable from s.) There is a theorem assures us that the flow we found is actually the max flow. And let A be the vertices that are reached in the final residue network, B be the rest of the world, all the edges from A to B in the original graph gives a min cut.

In the worst case, this algorithm is slow — it's even not promised to run in polynomial time. But if we always use BFS in finding the augmenting path, the algorithm can be proved in polynomial time.

We do not provide the code for this algorithm. But we do provide it under another name. A more general algorithm solves something more: the min cost max flow (Section 2.10).

The algorithm we are going to use is more sophisticated. We sketch the outline below, the complete code is identical to the solution to Problem 7.7.

Outline: the Goldberg-Tarjan Preflow-Push-Relabel Algorithm for Max Flow

Instead of sending some extra flow all the way to t each time, in this algorithm we send as much as possible from s, those excessive substance that cannot reach t will flow back. Such a flow, where the capacities are

obeyed, but not necessary all the substance are reaching t, is called a *preflow*. The initial preflow is

$$f(e) = \begin{cases} c(e) & \text{if } e \text{ goes out of } s \\ 0 & \text{otherwise} \end{cases}$$

Given any preflow, the excessiveness of a vertex, $ex(v)$, is define to be the amount of substance flow into v minus the amount flow out of v. If all the substance reach t, (i.e., the preflow is actually a flow) then any v other than s, t has excessiveness 0. We call a vertex *active* if it has positive excessiveness.

We will keep the label function ϕ so that $\phi(s) = n$, $\phi(t) = 0$, and $\phi(u) \leq \phi(v)+1$ for any edge $u \to v$. We initialize the labels $\phi(s) = n$, and $\phi(v) = 0$ for any other v. At any stage, an edge $u \to v$ is called *admissible* if $\phi(u) = \phi(v) + 1$.

```
void relabel(v){
    phi[v] := min(phi[w]+1), where w runs over all edges
    v->w in the residue network
}

void push(v, w){
    x := min (ex[v], c[v][w]); // note, c is the residue capacity
    f(v, w) += x; // push x substance from v to w
    f(w, v) -= x;
}

...
    f(u, v) = 0 for all edges;
    f(s, v) = c(s,v) for all v;
    phi[v] = 0 for all v; phi[s] = n;
    while (there is active vertex)
        pick an active vertex v
    if no edge v->w is admissible then relabel(v);
    else pick an admissible edge v->w and push(v,w);
...
```

At the end, the max flow is the sum of all flow out of s. To find the min cut, we may do a simple DFS on the residue network. (See the discussion in the Edmonds-Karp algorithm.)

2.9 Minimum (Maximum) Cost Bipartite Matching

Problem 2.7 *In a bipartite graph there might be multiple answers to the max matching problem. If there is a cost for each edge, what is the min(max) cost matching?*

We may assume the input graph is complete bipartite graph. (If there is no edge between A_i and B_j, we add one with very big cost.) In a lot of applications, the right part has at least as many points as the left part, so each point from the left part can be matched. We denote the partner of A_i by $B_{m[i]}$.

Outline: Cycle Canceling Algorithm for Minimum Bipartite Matching

We start with an arbitrary matching where the points left part are all matched, and repeat improving it, until there is no improvement, and we can prove at that time the matching is the min cost one.

To do the improvement, we draw an "improvement graph". For any u and v, we put weight x on the edge from u to v, if we give the current partner of v to u, we will gain cost x. i.e. $x = c(v, m[v]) - c(u, m[v])$. Now it is clear a negative cycle in the improvement graph means if we cyclicly switch the partners along that cycle, we will have a better matching.

The complete code is in the first solution to Problem 7.8.

2.10 Minimum Cost (Maximum Profit) Flow

Problem 2.8 (Min Cost (Max Profit) Flow) *Given a graph G, each edge e has a capacity* `c(e)`*, as well as a unit cost (profit)* `Cost[e]`*, and there are two vertices s and t. Among all the maximum flows from s to t, which one costs the least (biggest)? The cost of an edge is computed by the unit cost times the number of units we transfer along that edge, i.e.* `f(e)*Cost[e]`*.*

Remark. . We usually assume the costs are non-negative in finding the min cost flow. The real trouble occurs when there is a negative cycle, in this case we can always find cheaper and cheaper (negative cost) flows. The same trouble comes if we have positive cost cycles in the max cost flow problem.

Application. A slightly general question. If there are some set A of vertices with supplies $S(v), v \in A$, and another set B with demands $D(w), w \in B$, and $\sum_{v\in A} S(A) = \sum_{w\in B} D(w)$. We are interested in whether we can transfer the substances so that all the supplies are out, all the demands are satisfied, and no other points has any substance. If this is possible, there might be multiple ways to do it, what is cost for the least cost (most profit) way?

To solve this question, we simple add two vertices s and t. Add edges from s to each one $v \in A$ with capacity $S(v)$ and cost 0, and each one $w \in B$ to t with capacity $D(w)$ and cost 0. Then we solve the min(max) cost flow problem on the new graph. The original job is possible if and only if the amount of max flow is $\sum_{v\in A} S(A)$, and if it is possible, it gives the min(max) cost way to transfer the substance.

Application. (Min / Max cost matching in bipartite graph revisited) This is very similar to the previous application. Add s and t. Edge s to the left part vertices with capacity 1 and cost 0, and the right part vertices to t with capacity 1 and cost 0. The second solution to Problem 7.8 illustrates this.

Outline: Successive Shortest Path Algorithm

The complete code is in the second solution to Problem 7.8. (The part before `main()`. In `main()` function we illustrate what should be initialized.) The outline is surprisingly short:

```
start from the empty flow
while (1)
{
    find a shortest path P from s to t in the residue network
    if (there is no such path) break;
    let x be the minimum residue capacity of edges on P
    augment f along the path P with amount x
}
```

Well, we need to say what do we mean by "shortest path", since we have costs and capacities. Clearly, the natural guess will be the costs, and this is a correct guess.

We can use any shortest path algorithm to find P. Especially we may use Bellman-Ford, if there are negative costs.

Remark. As you see, this algorithm solves something much more than the max flow problem with almost no extra work than a max flow program.

3 Numbers

We list some basic Facts.

- If p is a prime and $a \not\equiv 0 \bmod p$, $a^{p-1} \equiv 1 \bmod p$; if $a^{(p-1)/2} \equiv 1 \bmod p$ then there exist b such that $b^2 \equiv a \bmod p$.

- Let n be a positive integer greater than 1 and let its unique prime factorization be $p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$ where $e_i > 0$ and p_i is prime for all i. Then the Euler Φ function

$$\Phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \dots (1 - \frac{1}{p_k})$$

  describes the number of positive integers co-prime to n in $[1..n]$. As a special case, $\Phi(p) = p - 1$ for prime p. The number of divisors of n is $\prod_i (e_i + 1)$.

- Euler's Theorem, which extends Fermat's Little Theorem: If $(a, n) = 1$, $a^{\Phi(n)} \equiv 1 \bmod p$.

3.1 the Greatest Common Divisor

Outline: GCD and extended GCD

```
int gcd(int a, int b) // need long long?
{
    if(a==0) return(b);
    return(gcd(b%a, a));
}

int extGcd(int a, int b, int& x, int& y)
// returns d=gcd(a,b), and give one pair x,y such that ax+by=d
{
    if(a==0)
    {
        x=0;y=1;
        return(b);
    }
    int a1, b1, c, x1, y1, rst;
    a1=b%a; b1=a; c=b/a;
    rst=extGcd(a1, b1, x1, y1);
    x=y1-c*x1; y=x1;
    return(rst);
}
```

3.2 Generating the Prime Table

Here is the code of the sieve of Erastothenes for generating the prime table less than 1000000. The number 1000000 can be adjusted to any number you like, but by the space restriction it may never be as big as 10^8.

Outline: Prime Table Generator

```
    int i,j;
    for(i=0;i<1000000;i++) pr[i]=1;
    pr[0]=pr[1]=0;
    for(i=2;i<1000000;i++) if(pr[i])
        for(j=i+i;j<=1000000;j+=i) pr[j]=0;
```

If you want to compress the prime numbers, i.e, not a 0-1 array of indicators, but actually a list of all the primes less than 1000000, you may do

Outline: Compressed Prime Table Generator

```
int pr[1000010], p[100010], cnt=0;
// pr is the indicator, p is the compressed table,
// usually 1/10 of pr is enough.

    ...
    int i,j;
    for(i=0;i<1000000;i++) pr[i]=1;
    pr[0]=pr[1]=0;
    for(i=2;i<1000000;i++) if(pr[i])
    {
        p[cnt]=i; cnt++;
        for(j=i+i;j<=1000000;j+=i) pr[j]=0;
    }
```

In the code above, `pr[]` and `p[]` can be the same array, if we will never need the 0-1 indicator later in the program.

If we are only interested in whether a single number a is prime, we just need to test whether a is divisible by b for any $b^2 \leq a$; or we just need to test if a is divisible by b for any prime number b where $b^2 \leq a$. Given a pre-computed prime table, the complexity of this job for $a = 10^9$ is roughly 10^4. Another faster way to test the prime numbers will be presented in the next section.

3.3 Repeated Squaring

To compute a^b when b is big. One remark: in this case you will never get the exact answer unless you use long integer with char arrays, because a^b is really a huge number. However, in a lot of applications, you just need know $a^b \% n$. For example, just want to know the last 5 digits of a^b.

The idea is, in the beginning, there is a. After one step, we can get a^2, then we get a^4, a^8, and so on. In the second stage, we we these blocks to get a^b. This algorithm involves the binary representation of b as well as repeated squaring of a.

Another way to look at this: To compute a^b. If $b = 2k + 1$, then we need $(a^2)^k \cdot a$; if $b = 2k$, we need $(a^2)^k$. Both cases are of recursive nature.

Outline: Repeated Squaring to get $a^b \bmod T$

```
int pwab5(long long a, int b) {
    long long r=1;
    while(b)
    {
        if(b%2) r=(r*a)%T;
        a=(a*a)%T;
        b/=2;
    }
    return (int)r;
}
```

Note that you need to be careful when do you need `long long`, when do you need `unsigned long long`, and when even these are not enough.

Below is a fast program to test if a number up to $2^{32} - 1$ is prime. It is based on the Rabin-Miller Strong Pseudoprime Test. In fact, if we replace the set $\{2, 7, 61\}$ by the set of first 8 primes $\{2, 3, 5, 7, 11, 13, 17, 19\}$, we have a prime number test for all numbers up to $3 \cdot 10^{14}$. The time complexity is roughly 200 steps in the worst case, and less than 100 steps on average.

Outline: Prime Test for Big Numbers

```
int suspect(long long b, int t, long long u, long long n)
{
    long long prod=1;
    while(u)
    {
        if(u&1) prod=((prod*b)%n);
        b=((b*b)%n);
        u/=2;
    }
    if(prod == 1) return 1;

    for(int i = 1; i <= t; i++)
    {
        if(prod == n-1) return 1;
        prod = (prod * prod) % n;
    }
    return 0;
}

int isprime(unsigned int n)
{
    long long k = n - 1;
    int t = 0;
    while(!(k%2)) { t++; k/=2; }
    if(n>2 && n%2==0) return 0;
    if(n>3 && n%3==0) return 0;
    if(n>5 && n%5==0) return 0;
    if(n>7 && n%7==0) return 0;
    if(suspect(61, t, k, n) && suspect(7, t, k, n) && suspect(2, t, k, n))
        return 1;
    return 0;
}
```

3.4 Long Integers using Character Arrays

The way we represent a number is from the least significant digit up. It is reverse to what we see. So you need the `rev()` function during the input/output.

We do not implement long integers with signs. You need to be careful about this when doing substraction.

Simply by changing the 10 to any other number B, we represent the numbers in base B. Sure, you need to do some extra work when output the answer if $B > 10$. See Problem 7.9 for an example.

Outline: Long Integers

```
void simplify(char v[]) {
    int i;
    i=strlen(v)-1;
    while(i>0 && v[i]=='0') {v[i]='\0';i--;}
}

void add(char v[], int q) {
    int c=q;
    int i,d;
```

```
    for(i=0;v[i];i++)
    {
        d=((v[i]-'0')+c);
        c=d/10;d%=10;
        v[i]='0'+d;
    }
    while(c)
    {
        v[i]='0'+(c%10);
        c/=10;i++;
    }
    v[i]='\0';
}

void multi(char v[], int q) {
    int c=0;
    int i,d;
    for(i=0;v[i];i++)
    {
        d=((v[i]-'0')*q+c);
        c=d/10;d%=10;
        v[i]='0'+d;
    }
    while(c)
    {
        v[i]='0'+(c%10);
        c/=10;i++;
    }
    v[i]='\0';
}

int divi(char v[], int q)
// returns the reminder
{
    int i,l=strlen(v);
    int c=0,d;
    for(i=l-1;i>=0;i--)
    {
        d=c*10+(v[i]-'0');
        c=d%q; d/=q; v[i]='0'+d;
    }
    i=l-1;
    while(i>0 && v[i]=='0') i--;
    v[i+1]='\0';
    return c;
}

void add(char v1[], char v2[])
// v1 = v1+v2;
{
        int i,d,c=0;
```

```
        int l1=strlen(v1);
        int l2=strlen(v2);
        for(i=l1;i<l2;i++) v1[i]='0';
        for(i=l2;i<l1;i++) v2[i]='0';
        for(i=0;i<l1||i<l2;i++)
        {
                d=(v1[i]-'0')+(v2[i]-'0')+c;
                c=d/10;d%=10;
                v1[i]='0'+d;
        }
        while(c)
        {
                v1[i]='0'+(c%10);
                c/=10;i++;
        }
        v1[i]='\0';
        v2[l2]='\0';
}

void subs(char v1[], char v2[])
// v1=v1-v2;
{
        int i,d,c=0;
        int l1=strlen(v1);
        int l2=strlen(v2);
        for(i=l2;i<l1;i++) v2[i]='0';
        for(i=0;i<l1;i++)
        {
                d=(v1[i]-'0'-c)-(v2[i]-'0');
                if(d<0) {d+=10; c=1;} else c=0;
                v1[i]='0'+d;
        }
        v2[l2]='\0';
    i=l1-1;
    while(i>0 && v1[i]=='0') i--;
    v1[i+1]='\0';
}

//return the sign of v1-v2
int comp(char v1[], char v2[]) {
    int i;
    int l1=strlen(v1);
    int l2=strlen(v2);
    if(l1>l2) return(1);
    if(l1<l2) return(-1);
    for(i=l1-1;i>=0;i--)
    {
        if(v1[i]>v2[i]) return(1);
        if(v1[i]<v2[i]) return(-1);
    }
    return(0);
```

```
}

char tmp[10000]; char tmp1[10000]; char tmpd[10][10000]; char
bs[10000];

void multi(char v1[], char v2[])
// v1=v1*v2;
{
    bs[0]='\0';
    int l2=strlen(v2);
    int i;
    strcpy(tmpd[0],"0");
    for(i=1;i<10;i++)
    {
        strcpy(tmpd[i],tmpd[i-1]);
        add(tmpd[i],v1);
    }
    strcpy(v1,"0");
    for(i=0;i<l2;i++)
    {
        strcpy(tmp,bs); bs[i]='0';bs[i+1]='\0';
        strcat(tmp,tmpd[v2[i]-'0']);
        add(v1,tmp);
    }
}

void multi(char v1[], char v2[], char v3[])
//make sure v1 is not v3
// v3=v1*v2;
{
    bs[0]='\0';
    int l2=strlen(v2);
    int i;
    strcpy(tmpd[0],"0");
    for(i=1;i<10;i++)
    {
        strcpy(tmpd[i],tmpd[i-1]);
        add(tmpd[i],v1);
    }
    strcpy(v3,"0");
    for(i=0;i<l2;i++)
    {
        strcpy(tmp,bs); bs[i]='0';bs[i+1]='\0';
        strcat(tmp,tmpd[v2[i]-'0']);
        add(v3,tmp);
    }
}

void divi(char v1[], char v2[], char v3[], char v4[])
//v1/v2=v3...v4
// make sure v3, v4 are different from v1, v2
```

```
{
    int i;
    if(strcmp(v2, "1")==0)
    {
        strcpy(v3, v1);
        strcpy(v4, "0");
        return;
    }
    if(strcmp(v1, "0")==0)
    {
        strcpy(v3, "0");
        strcpy(v4, "0");
        return;
    }

    for(i=0;v1[i];i++) v3[i]='0';
    v3[i]='\0';

    int ff=1;
    int l=i;
    for(i=l-1;i>=0;i--)
    {
        while(1)
        {
            if(v3[i]=='9') break;
            v3[i]++;
            multi(v3, v2, v4);
            if(comp(v4, v1)>0)
            {
                v3[i]--;
                break;
            }
            ff=0;
        }
        if(ff && i) v3[i]='\0';
        //simplify(v3);
    }
    multi(v2, v3, tmp1);
    strcpy(v4, v1);
    subs(v4, tmp1);
}

void showBigint(char v[]) {
    simplify(v);
    int l=strlen(v);
    int i;
    for(i=l-1;i>=0;i--) cout<<v[i];
}

void rev(char v[]) {
    int l=strlen(v);
```

```
    int i; char cc;
    for(i=0;i<l-1-i;i++)
    {
        cc=v[i];v[i]=v[l-1-i];v[l-i-1]=cc;
    }
}
```

3.5 Fractional Numbers

We write a fractional number in the form a/b. In the standard form, $\gcd(a, b) = 1$ and if a/b is negative, we put the sign on a. These are the jobs of `adjFr` function.

Outline: Fractional Numbers

```
struct fr {
    int a,b; // long long? in that case also change gcd to long long
    // fractional number a/b
};

int gcd(int a, int b) {
    if(a==0) return(b);
    return(gcd(b%a, a));
}

void adjFr(fr &v)
// change v to the reduced form,
// so that gcd(a,b)=1 and b>0
{
    int tmp=1;
    if(v.b==0) {v.a=0; return;}
    if(v.b*v.a<0) tmp=-1;
    if(v.a<0) v.a=-v.a;
    if(v.b<0) v.b=-v.b;
    int g=gcd(v.a,v.b);
    v.a/=g;v.b/=g;
    v.a*=tmp;
}

void addFr(fr v1, fr v2, fr &v)
// make sure v does not equal v1
{
    v.b=v1.b*v2.b;
    v.a=v1.a*v2.b+v1.b*v2.a;
    adjFr(v);
}

void SubFr(fr v1, fr v2, fr &v)
// v can't be v1
{
    v.b=v1.b*v2.b;
    v.a=v1.a*v2.b-v1.b*v2.a;
```

```
    adjFr(v);
}

void MultiFr(fr v1, fr v2, fr &v)
// v can't be v1
{
    v.b=v1.b*v2.b;
    v.a=v1.a*v2.a;
    adjFr(v);
}

void DivFr(fr v1, fr v2, fr &v)
// v can't be v1
{
    v.b=v1.b*v2.a;
    v.a=v1.a*v2.b;
    if(v.b!=0) adjFr(v); else v.a=0;
}
```

4 Algebra

4.1 the Matrix Library

To invert a matrix M, we write down the identity matrix I to the right of M to make a $n \times 2n$ matrix, then do a Gauss elimination to transform the left half to be I, the right half will be M^{-1}. The idea is that the operations in the Gauss elimination are row reductions, the whole effect M' is the multiplication of the row reduction matrices. If M' transform M to I, then M' is M^{-1}, and it transform I to itself. The similar idea is used to solve the equation $Mx = b$ (in any field).

Remark. Our matrix library is based on the real field. If you want to deal with the finite field F_p where p is a prime number, you need to change several places: `LD` becomes `int`; after every addition and multiplication you should mod p. After every substraction you should do `x=(x%p+p)%p` to avoid negative results. Every division by r becomes the multiplication by r^{-1} in the field. Use Robert's Theorem:

Theorem 4.1 (Robert Renaud) *In the field F_p where p is a prime, if $a \neq 0$, the inverse of a is a^{p-2} mod p.*

Thus, we can use the repeated squaring (Section 3.3) to compute (or precompute) a^{-1}.

In the library we solve the equation $Ax = b$ only when A is a invertible square matrix. If it is not invertible, there might be no solution or infinitely many solutions, but we did not handle these cases in our code.

Remark. The three lines in the `solve()` function is a nice place to see what kind of operations we can do with our library.

```
typedef long double LD;

LD EPS = 1e-8;

struct MATRIX
{
    int n,m;
```

```
vector< vector<LD> > a;

void resize(int x, int y, LD v=0.0)
{
    n=x; m=y;
    a.resize(n);
    for(int i=0; i<n; i++) a[i].resize(m, v);
}
LD Gauss()
// Row elimination based on the first n columns
// if the first n columns is not invertible, kill yourself
// otherwise, return the determinant of the first n columns
{
    int i,j,k;
    LD det=1.0, r;
    for(i=0;i<n;i++)
    {
        for(j=i, k=-1; j<n; j++) if(fabs(a[j][i])>EPS)
            { k=j; j=n+1; }
        if(k<0) { n=0; return 0.0; }
        if(k != i) { swap(a[i], a[k]); det=-det; }
        r=a[i][i]; det*=r;
        for(j=i; j<m; j++) a[i][j]/=r;
        for(j=i+1; j<n; j++)
        {
            r=a[j][i];
            for(k=i; k<m; k++) a[j][k]-=a[i][k]*r;
        }
    }
    for(i=n-2; i>=0; i--)
    for(j=i+1; j<n; j++)
    {
        r=a[i][j];
        for(k=j; k<m; k++) a[i][k]-=r*a[j][k];
    }
    return det;
}
int inverse()
// assume n=m. returns 0 if not invertible
{
    int i, j, ii;
    MATRIX T; T.resize(n, 2*n);
    for(i=0;i<n;i++) for(j=0;j<n;j++) T.a[i][j]=a[i][j];
    for(i=0;i<n;i++) T.a[i][i+n]=1.0;
    T.Gauss();
    if(T.n==0) return 0;
    for(i=0;i<n;i++) for(j=0;j<n;j++) a[i][j]=T.a[i][j+n];
    return 1;
}
vector<LD> operator*(vector<LD> v)
// assume v is of size m
```

```
    {
        vector<LD> rv(n, 0.0);
        int i,j;
        for(i=0;i<n;i++)
            for(j=0;j<m;j++)
                rv[i]+=a[i][j]*v[j];
        return rv;
    }
    MATRIX operator*(MATRIX M1)
    {
        MATRIX R;
        R.resize(n, M1.m);
        int i,j,k;
        for(i=0;i<n;i++)
            for(j=0;j<M1.m;j++)
                for(k=0;k<m;k++) R.a[i][j]+=a[i][k]*M1.a[k][j];
        return R;
    }
    void show()
    {
        int i,j;
        for(i=0;i<n;i++)
        {
            for(j=0;j<m;j++) printf("%15.10f ", (double)a[i][j]);
            printf("\n");
        }
        printf("end of the show \n");
    }
};

LD det(MATRIX &M)
// compute the determinant of M
{
    MATRIX M1=M;
    LD r=M1.Gauss();
    if(M1.n==0) return 0.0;
    return r;
}

vector<LD> solve(MATRIX& M, vector<LD> v)
// return the vector x such that Mx = v; x is empty if M is not invertible
{
    vector<LD> x;
    MATRIX M1=M;
    if(!M1.inverse()) return x;
    return M1*v;
}

void show(vector<LD> v)
{
     int i;
```

```
    for(i=0;i<v.size();i++) printf("%15.10f ", (double)v[i]);
    printf("\n");
}
```

4.2 the Method of Relaxation

The method of relaxation is a very elegant way to solve some system of linear equations. In some problems, we have a graph G, and there is an unknown variable x_i on each vertex v_i. Besides, for each x_i we have a linear equation in the form $x_i = \sum_j r_{ij}x_j$ telling us if all the (neighboring values) x_j are known, how shall we compute x_i. We view this equation as a rule on the vertex i. The set of rules is nothing but a system of linear equations. In some problems it is clear (maybe by the law of the nature) there should be a unique solution to the system. We give some examples:

- **Electric Network:** The graph is an electric network with resistors, the voltage on v_0 is 0, and the voltage on v_{n-1} is 1. What is the voltage on each vertex.
- **The drunkard's walk on a line:** There are n stations on a line, the leftmost one, 0, is the home, the rightmost one, $n-1$, is the jail. The drunkard's walk stops either he hits the jail or home. At any other position, he move to the left or to the right with equal probability 1/2. Given any start position i, let x_i be the probability that he hits the jail first. Then the set of rules is: $p_0 = 0$, $p_{n-1} = 1$, and $p_i = \frac{1}{2}p_{i-1} + \frac{1}{2}p_{i+1}$ for all other i. (Focus on what happens on the first move.) Similarly, let E_i be the expected number of steps he needs to take before he stops, how to solve E_i? Now the rules becomes $E_0 = E_{n-1} = 0$, and $E_i = \frac{1}{2}(E_{i-1}+1) + \frac{1}{2}(E_{i+1}+1) = 1 + \frac{1}{2}(E_{i-1} + E_{i+1})$ for any other i.
- More general, the **drunkard's walk on a graph**: Now he walks on a graph. v_0 is still the home, and v_{n-1} the jail. From each point, in one step, he will randomly choose one of the neighbors in the graph. Formally, for $0 < i < n-1$, if v_i has t neighbors $u_1, \cdots, u_t$, then the rule on vertex i is

$$p_i = \frac{1}{t}\sum_j p_{u_j}, \text{ and } E_i = 1 + \frac{1}{t}\sum_j E_{u_j}$$

  .
- Even more general, the **random walk on a graph**: Now the person random choose a neighbor, but not with equal probability. There is a fixed set of probabilities r_{ij}, where r_{ij} is the probability that, if the person is at v_i, he will choose v_j for the next step. So, the reasonable condition is that for each i, the probability going out of it sum up to 1. The rules are not any more complicated than the previous problem.

Remark. I guess you guessed that the first problem and the last problem are actually the same: From any vertex, the resistors adjacent to that vertex gives a set of probabilities (big resistors means harder to travel via that edge). Let p_i be the probability that we start from v_i, do the random walk on the graph, we hit v_0 before we hit v_{n-1}. p_i is exactly the same as the voltage on vertex i. This is the beautiful relation between random walks and electric networks.

Now comes the same idea occurred in Bellman-Ford and Floyd. This general idea is called relaxation: We start from an arbitrary solution, and based on the rules we do iterations, after each round the solution improves. And a nice thing here is that they improves fast, very quickly our solution converges to the real solution.

Here is the simple solution to these problems:

Outline: Solving Linear Equations by Relaxation

We have n variables x_i, $0 \le i < n$; and a rule for each variable: $x_i = \sum_j r_{ij}x_j + c_i$. The r's and c's are fixed. We want to solve the x's.

```
    //start from any initial values.
    x[i]=0.0 for all i;
    repeat enough times (*)
        for each rule on i
            x[i] = sum(r[i][j]*x[j])+c[i];
```

One word about the (*). You may check in each repetition what is the maximum amount of change in x_i's. Stop when the maximum change is negligible (say, less than 10^{-9}). For most problems, the method converges to the solution very quickly. You may simply say "repeat 1000 times" or so.

5 Geometry

Here is our huge piece of planar geometry library. We list the whole code first, then make some remarks.

5.1 the Geometry Library

```
#define MAX_SIZE 1000

const double PI = 2.0*acos(0.0);
const double EPS = 1e-9; //too small/big?????

struct PT
{
    double x,y;

    double length() {return sqrt(x*x+y*y);}

    int normalize()
    // normalize the vector to unit length; return -1 if the vector is 0
    {
        double l = length();
        if(fabs(l)<EPS) return -1;
        x/=l; y/=l;
        return 0;
    }
    PT operator-(PT a)
    {
        PT r;
        r.x=x-a.x; r.y=y-a.y;
        return r;
    }
    PT operator+(PT a)
    {
        PT r;
        r.x=x+a.x; r.y=y+a.y;
        return r;
    }
    PT operator*(double sc)
    {
        PT r;
        r.x=x*sc; r.y=y*sc;
```

```
        return r;
    }
};

bool operator<(const PT& a,const PT& b)
{
    if(fabs(a.x-b.x)<EPS) return a.y<b.y;
    return a.x<b.x;
}

double dist(PT& a, PT& b)
  // the distance between two points
{
    return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

double dot(PT& a, PT& b)
  // the inner product of two vectors
{
    return(a.x*b.x+a.y*b.y);
}

// ================================================================
// The Convex Hull
// ================================================================


int sideSign(PT& p1,PT& p2,PT& p3)
  // which side is p3 to the line p1->p2? returns: 1 left, 0 on, -1 right
{
    double sg = (p1.x-p3.x)*(p2.y-p3.y)-(p1.y - p3.y)*(p2.x-p3.x);
    if(fabs(sg)<EPS) return 0;
    if(sg>0)return 1;
    return -1;
}


bool better(PT& p1,PT& p2,PT& p3)
  // used by convec hull: from p3, if p1 is better than p2
{
    double sg = (p1.y - p3.y)*(p2.x-p3.x)-(p1.x-p3.x)*(p2.y-p3.y);
    //watch range of the numbers
    if(fabs(sg)<EPS)
    {
            if(dist(p3,p1)>dist(p3,p2))return true;
            else return false;
    }
    if(sg<0) return true;
    return false;
}
```

```
//convex hull in n*n
void vex(vector<PT>& vin,vector<PT>& vout)
{
    vout.clear();
    int n=vin.size();
    int st=0;
    int i;
    for(i=1;i<n;i++) if(vin[i]<vin[st]) st=i;
    vector<int> used;
    // used[i] is the index of the i-th point on the hull
    used.push_back(st);
    int idx=st; int next;
    do{
        next=0;
        for(i=1;i<n;i++)
            if(better(vin[i],vin[next],vin[idx]))next=i;
        idx=next;
        used.push_back(idx);
    }while(idx!=st);
    for(i=0;i+1<used.size();i++) vout.push_back(vin[used[i]]);
}

//convex hull nlogn
void vex2(vector<PT> vin,vector<PT>& vout)
  // vin is not pass by reference, since we will rotate it
{
    vout.clear();
    int n=vin.size();
    sort(vin.begin(),vin.end());
    PT stk[MAX_SIZE];
    int pstk, i;
    // hopefully more than 2 points
    stk[0] = vin[0];
    stk[1] = vin[1];
    pstk = 2;
    for(i=2; i<n; i++)
    {
        if(dist(vin[i], vin[i-1])<EPS) continue;
        while(pstk > 1 && better(vin[i], stk[pstk-1], stk[pstk-2]))
                pstk--;
        stk[pstk] = vin[i];
        pstk++;
    }

    for(i=0; i<pstk; i++) vout.push_back(stk[i]);

    // turn 180 degree
    for(i=0; i<n; i++)
    {
            vin[i].y = -vin[i].y;
            vin[i].x = -vin[i].x;
```

```
    }

    sort(vin.begin(), vin.end());

    stk[0] = vin[0];
    stk[1] = vin[1];
    pstk = 2;

    for(i=2; i<n; i++)
    {
        if(dist(vin[i], vin[i-1])<EPS) continue;
        while(pstk > 1 && better(vin[i], stk[pstk-1], stk[pstk-2]))
            pstk--;
        stk[pstk] = vin[i];
        pstk++;
    }

    for(i=1; i<pstk-1; i++)
    {
        stk[i].x= -stk[i].x; // don't forget rotate 180 d back.
        stk[i].y= -stk[i].y;
        vout.push_back(stk[i]);
    }
}

int isConvex(vector<PT>& v)
// test whether a simple polygon is convex
// return 0 if not convex, 1 if strictly convex,
// 2 if convex but there are points unnecesary
// this function does not work if the polycon is self intersecting
// in that case, compute the convex hull of v, and see if both have the same area
{
    int i,j,k;
    int c1=0; int c2=0; int c0=0;
    int n=v.size();
    for(i=0;i<n;i++)
    {
        j=(i+1)%n;
        k=(j+1)%n;
        int s=sideSign(v[i], v[j], v[k]);
        if(s==0) c0++;
        if(s>0) c1++;
        if(s<0) c2++;
    }
    if(c1 && c2) return 0;
    if(c0) return 2;
    return 1;
}
```

```
// ================================================================
// Areas
// ================================================================

double trap(PT a, PT b)
{
    return (0.5*(b.x - a.x)*(b.y + a.y));
}

double area(vector<PT> &vin)
  // Area of a simple polygon, not neccessary convex
{
    int n = vin.size();
    double ret = 0.0;
    for(int i = 0; i < n; i++)
            ret += trap(vin[i], vin[(i+1)%n]);
        return fabs(ret);
}

double peri(vector<PT> &vin)
  // Perimeter of a simple polygon, not neccessary convex
{
    int n = vin.size();
    double ret = 0.0;
    for(int i = 0; i < n; i++)
            ret += dist(vin[i], vin[(i+1)%n]);
    return ret;
}

double triarea(PT a, PT b, PT c)
{
    return fabs(trap(a,b)+trap(b,c)+trap(c,a));
}

double height(PT a, PT b, PT c)
  // height from a to the line bc
{
    double s3 = dist(c, b);
    double ar=triarea(a,b,c);
    return(2.0*ar/s3);
}


// ======================================================
// Points and Lines
// ======================================================

int intersection( PT p1, PT p2, PT p3, PT p4, PT &r )
  // two lines given by p1->p2, p3->p4 r is the intersection point
  // return -1 if two lines are parallel
{
```

```
    double d = (p4.y - p3.y)*(p2.x-p1.x) - (p4.x - p3.x)*(p2.y - p1.y);

    if( fabs( d ) < EPS ) return -1;
        // might need to do something special!!!

        double ua, ub;
        ua = (p4.x - p3.x)*(p1.y-p3.y) - (p4.y-p3.y)*(p1.x-p3.x);
        ua /= d;
        //  ub = (p2.x - p1.x)*(p1.y-p3.y) - (p2.y-p1.y)*(p1.x-p3.x);
        //ub /= d;
        r = p1 + (p2-p1)*ua;
        return 0;
}

void closestpt( PT p1, PT p2, PT p3, PT &r )
  // the closest point on the line p1->p2 to p3
{
    if( fabs( triarea( p1, p2, p3 ) ) < EPS )
    { r = p3; return; }
    PT v = p2-p1;
    v.normalize();
    double pr; // inner product
    pr = (p3.y-p1.y)*v.y + (p3.x-p1.x)*v.x;
    r = p1+v*pr;
}

int hcenter( PT p1, PT p2, PT p3, PT& r )
{
    // point generated by altitudes
    if( triarea( p1, p2, p3 ) < EPS ) return -1;
    PT a1, a2;
    closestpt( p2, p3, p1, a1 );
    closestpt( p1, p3, p2, a2 );
    intersection( p1, a1, p2, a2, r );
    return 0;
}

int center( PT p1, PT p2, PT p3, PT& r )
{
    // point generated by circumscribed circle
    if( triarea( p1, p2, p3 ) < EPS ) return -1;
    PT a1, a2, b1, b2;
    a1 = (p2+p3)*0.5;
    a2 = (p1+p3)*0.5;
    b1.x = a1.x - (p3.y-p2.y);
    b1.y = a1.y + (p3.x-p2.x);
    b2.x = a2.x - (p3.y-p1.y);
    b2.y = a2.y + (p3.x-p1.x);
    intersection( a1, b1, a2, b2, r );
    return 0;
}
```

```
int bcenter( PT p1, PT p2, PT p3, PT& r )
{
    // angle bisection
    if( triarea( p1, p2, p3 ) < EPS ) return -1;
    double s1, s2, s3;
    s1 = dist( p2, p3 );
    s2 = dist( p1, p3 );
    s3 = dist( p1, p2 );

    double rt = s2/(s2+s3);
    PT a1,a2;
    a1 = p2*rt+p3*(1.0-rt);
    rt = s1/(s1+s3);
    a2 = p1*rt+p3*(1.0-rt);
    intersection( a1,p1, a2,p2, r );
    return 0;
}


// ================================================
// Angles
// ================================================

double angle(PT& p1, PT& p2, PT& p3)
  // angle from p1->p2 to p1->p3, returns -PI to PI
{
    PT va = p2-p1;
    va.normalize();
    PT vb; vb.x=-va.y; vb.y=va.x;
    PT v = p3-p1;
    double x,y;
    x=dot(v, va);
    y=dot(v, vb);
    return(atan2(y,x));
}

double angle(double a, double b, double c)
  // in a triangle with sides a,b,c, the angle between b and c
  // we do not check if a,b,c is a triangle here
{
    double cs=(b*b+c*c-a*a)/(2.0*b*c);
    return(acos(cs));
}

void rotate(PT p0, PT p1, double a, PT& r)
  // rotate p1 around p0 clockwise, by angle a
  // don't pass by reference for p1, so r and p1 can be the same
{
    p1 = p1-p0;
    r.x = cos(a)*p1.x-sin(a)*p1.y;
```

```
    r.y = sin(a)*p1.x+cos(a)*p1.y;
    r = r+p0;
}

void reflect(PT& p1, PT& p2, PT p3, PT& r)
  // p1->p2 line, reflect p3 to get r.
{
    if(dist(p1, p3)<EPS) {r=p3; return;}
    double a=angle(p1, p2, p3);
    r=p3;
    rotate(p1, r, -2.0*a, r);
}


// ================================================
// points, lines, and circles
// ================================================

int pAndSeg(PT& p1, PT& p2, PT& p)
  // the relation of the point p and the segment p1->p2.
  // 1 if point is on the segment; 0 if not on the line; -1 if on the line but not on the segment
{
    double s=triarea(p, p1, p2);
    if(s>EPS) return(0);
    double sg=(p.x-p1.x)*(p.x-p2.x);
    if(sg>EPS) return(-1);
    sg=(p.y-p1.y)*(p.y-p2.y);
    if(sg>EPS) return(-1);
    return(1);
}

int lineAndCircle(PT& oo, double r, PT& p1, PT& p2, PT& r1, PT& r2)
  // returns -1 if there is no intersection
  // returns 1 if there is only one intersection
{
    PT m;
    closestpt(p1,p2,oo,m);
    PT v = p2-p1;
    v.normalize();

    double r0=dist(oo, m);
    if(r0>r+EPS) return -1;
    if(fabs(r0-r)<EPS)
    {
        r1=r2=m;
        return 1;
        }
    double dd = sqrt(r*r-r0*r0);
    r1 = m-v*dd; r2 = m+v*dd;
    return 0;
}
```

```
int CAndC(PT o1, double r1, PT o2, double r2, PT& q1, PT& q2)
  // intersection of two circles
  // -1 if no intersection or infinite intersection
  // 1 if only one point
{
    double r=dist(o1,o2);
    if(r1<r2) { swap(o1,o2); swap(r1,r2); }
    if(r<EPS) return(-1);
    if(r>r1+r2+EPS) return(-1);
    if(r<r1-r2-EPS) return(-1);
    PT v = o2-o1; v.normalize();
    q1 = o1+v*r1;
    if(fabs(r-r1-r2)<EPS || fabs(r+r2-r1)<EPS)
    { q2=q1; return(1); }
    double a=angle(r2, r, r1);
    q2=q1;
    rotate(o1, q1, a, q1);
    rotate(o1, q2, -a, q2);
    return 0;
}

int pAndPoly(vector<PT> pv, PT p)
  // the relation of the point and the simple polygon
  // 1 if p is in pv; 0 outside; -1 on the polygon
{
    int i, j;
    int n=pv.size();
    pv.push_back(pv[0]);
    for(i=0;i<n;i++)
            if(pAndSeg(pv[i], pv[i+1], p)==1) return(-1);
    for(i=0;i<n;i++)
            pv[i] = pv[i]-p;
        p.x=p.y=0.0;
    double a, y;
    while(1)
        {
            a=(double)rand()/10000.00;
            j=0;
            for(i=0;i<n;i++)
        {
            rotate(p, pv[i], a, pv[i]);
            if(fabs(pv[i].x)<EPS) j=1;
        }
            if(j==0)
        {
            pv[n]=pv[0];
            j=0;
            for(i=0;i<n;i++) if(pv[i].x*pv[i+1].x < -EPS)
                {
                    y=pv[i+1].y-pv[i+1].x*(pv[i].y-pv[i+1].y)/(pv[i].x-pv[i+1].x);
```

```
                    if(y>0) j++;
                }
            return(j%2);
        }
        }
    return 1;
}


void show(PT& p)
{
    cout<<"("<<p.x<<", "<<p.y<<")"<<endl;
}

void show(vector<PT>& p)
{
    int i,n=p.size();
    for(i=0;i<n;i++) show(p[i]);
    cout<<":)"<<endl;
}

void cutPoly(vector<PT>& pol, PT& p1, PT& p2, vector<PT>& pol1, vector<PT>& pol2)
// cut the convex polygon pol along line p1->p2;
// pol1 are the resulting polygon on the left side, pol2 on the right.
{
    vector<PT> pp,pn;
    pp.clear(); pn.clear();
    int i, sg, n=pol.size();
    PT q1,q2,r;
    for(i=0;i<n;i++)
    {
        q1=pol[i]; q2=pol[(i+1)%n];
        sg=sideSign(p1, p2, q1);
        if(sg>=0) pp.push_back(q1);
        if(sg<=0) pn.push_back(q1);
        if(intersection(p1, p2, q1, q2,r)>=0)
        {
            if(pAndSeg(q1, q2, r)==1)
            {
                pp.push_back(r);
                pn.push_back(r);
            }
        }
    }
    pol1.clear(); pol2.clear();
    if(pp.size()>2) vex2(pp, pol1);
    if(pn.size()>2) vex2(pn, pol2);
    //show(pol1);
    //show(pol2);
}
```

```
// ==========================================================================
// UVA 137, the intersection of two CONVEX polygons.
// ==========================================================================
// return 1 if the intersection is empty.


int PInterP(vector<PT>& p1, vector<PT>& p2, vector<PT>& p3)
{
    vector<PT> pts;
    PT pp;
    pts.clear();
    int m=p1.size();
    int n=p2.size();
    int i, j;
    for(i=0;i<m;i++)
        if(pAndPoly(p2, p1[i])!=0) pts.push_back(p1[i]);
    for(i=0;i<n;i++)
        if(pAndPoly(p1, p2[i])!=0) pts.push_back(p2[i]);
    if(m>1 && n>1)
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            if(intersection(p1[i], p1[(i+1)%m], p2[j], p2[(j+1)%n], pp)==0)
            {
                //cout<<i<<" "<<j<<" -> "<<pp.x<<" "<<pp.y<<endl;
                if(pAndSeg(p1[i], p1[(i+1)%m], pp)!=1) continue;
                if(pAndSeg(p2[j], p2[(j+1)%n], pp)!=1) continue;
                pts.push_back(pp);
            }
    if(pts.size()<=1)
    {
        p3.resize(1);
        p3[0].x=p3[0].y=0.0;
        return(1);
    }
    //show(pts);
    vex2(pts, p3); // or vex
    return(0);
}
```

We think of the `struct PT` as both point and vector. Thus, `PT(3,5)` can be interpreted both as the point $(3, 5)$, or the vector from the origin to $(3, 5)$. Especially, when we use the member functions of `PT` (`normalize(), +, -, *`), we think it as a vector.

Extra care is always needed when we deal with precision problems and special cases in geometry. There are many nice little tricks to deal with these, which are beyond the scope of this notes.

5.2 Some Geometric Facts

The following list are derived from the Taylor series.

- $\sin x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{(2n-1)!} x^{2n-1}$
- $\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$

- $e = 2.71828182845904523536028747135266249775 7$
  $= \lim_{x\to\infty}(1+\frac{1}{x})^x$
  $= \sum_{k=0}^{\infty}\frac{1}{k!}$
- $\pi = 3.14159265358979323846264338327950288419$
  $= 4\times\sum_{k=1}^{\infty}\frac{(-1)^{k+1}}{2k-1}$
  $= 4\times(1-\frac{1}{3}+\frac{1}{5}-\ldots)$

There is a nice one-one correspondence between the point (a, b) and the complex number $(a + ib)$. A complex number can also be written as (r, α), where $r = \sqrt{a^2+b^2}$ is the length, and $\alpha =$ `atan2(b, a)` is the counter-clockwise angle from the x-axis to the vector. Conversely, if we know (r, α), the point is given by $(r\cos(\alpha), ir\sin(\alpha))$. The product of two complex numbers (r_1, α_1) and (r_2, α_2) is $(r_1r_2, \alpha_1 + \alpha_2)$. Based on these facts, we derive

- $\sin(\alpha+\beta) = \sin\alpha\cos\beta + \sin\beta\cos\alpha$
- $\sin(\alpha-\beta) = \sin\alpha\cos\beta - \sin\beta\cos\alpha$
- $\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$
- $\cos(\alpha-\beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$
- $\sin(n\theta) = 2\cos\theta\sin[(n-1)\theta] - \sin[(n-2)\theta]$
- $\cos(n\theta) = 2\cos\theta\cos[(n-1)\theta] - \cos[(n-2)\theta]$

Given any two independent vectors f_1 and f_2 as basis, we have a coordinate system. If f_1 and f_2 are orthogonal unit vectors, we say that they form a orthonormal basis.

For any two vectors u and v, their inner product $(u, v) = |u||v|\cos(\alpha)$, where α is the angle between them. If v is a unit vector, then (u, v) is the projection of u on the v direction.

So, if $(e1, e2)$ is an orthonormal basis, the coordinates of a vector v is just (u, e_1) for the first coordinate, (u, e_2) for the second. This is also true in higher dimensions. In general, if the basis (f_1, f_2) is not orthonomal and we want to get the coordinate (a, b) of v in the new system, we just need to solve a simple equation $v = af_1 + bf_2$, here a and b are unknowns, and we have two equations,

$$v.x = af_1.x + bf_2.x,$$

$$v.y = af_1.y + bf_2.y.$$

Certainly this has no solution iff (f_1, f_2) is not a basis, i.e., they are of the same or opposite direction.

Let ABC be a triangle, denote the opposite edges by a, b, and c, respectively, and denote the angles by α, β, γ, respectively. We have

- $\alpha+\beta+\gamma = \pi$.
- $a = b$ iff $\alpha = \beta$; $a < b$ iff $\alpha < \beta$
- $a^2 = b^2 + c^2 - 2bc\cos\alpha$
- $\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma} = 2R$, where R is the radius of the circumcircle.
- $S = \frac{1}{2}ab\sin\gamma = \frac{abc}{4R} = \frac{1}{2}(a+b+c)r$, where S is the area of the triangle, R is the radius of the circumcircle, r is the radius of the incircle.

From the fact $a^2 = b^2 + c^2 - 2bc\cos\alpha$, we can derive the following fact: Let a and b be the length of sides of a parallelogram, x and y be the length of its two diagonals, then $2(a^2 + b^2) = x^2 + y^2$.

Our function `pAndPoly` decides whether a point is inside a simple polygon. If we know the polygon is convex, then there is an easier program: For each edge, P and the end points of that edge form a triangle. We add the areas together. P is inside the triangle iff the sum of the areas equals the area of the polygon. Otherwise, the sum of the areas will be strictly bigger.

6 Miscellaneous

6.1 Binary Search

Binary search is done on a monotone array or a monotone function on a range of numbers. The first thing and most important thing in writing the binary search is to write a good plan, or so called loop invariant.

In the example below, we assume there exists a number x between 0 and $n-1$, such that any number less than x does not have certain property P, and any other number has property P. We want to binary search the number x.

Outline: Binary Search on Integers

```
int binarySearch()
    // want: P(lo) is always false, and P(hi) always true
    int lo, hi, mi; lo=0; hi=n-1;
    while(lo+1<hi)
    {
        mi=(lo+hi)/2;
        if(P(mi)) hi=mi; else lo=mi;
    }
    return hi;
}
```

See Problem 7.2 for an nice example of binary search on integers.

In the example below, we assume there exists a real number x between a and b (e.g., 0 and 10^10), such that we can always tell if a number is smaller than x or not. We want to binary search the number x.

Outline: Binary Search on Reals

```
int binarySearch()
    // want: lo is always <x, and hi always >=x
    double lo, hi, mi; lo=a; hi=b;
    while(lo+EPS<hi)
    // you need to decide what is the right EPS, 1e-9?
    {
        mi=(lo+hi)/2.0;
        if(mi is too small) lo=mi; else hi=mi;
    }
    return hi;
}
```

When playing with big numbers with char arrays, some times we use 10-nary search. For example, if we have a big integer v in a char array, we want to find the integer part of $\sqrt{v}$. A binary search would involve long integer division, which is a little painful. The 10-nary search is a little slower, but with a succinct program.

6.2 Range Query

Given a (big) array `r[0..n-1]`, and a lot of queries of certain type. We may want to pre-process the data so that each query can be performed fast. In this section, we use $T(f, g)$ to denote the running time for an algorithm is $O(f(n))$ for pre-processing, and $O(g(n))$ for each query.

If the queries are of type `getsum(a, b)`, which asks the sum of all the elements between a and b, inclusive, we have a $T(n, 1)$ algorithm: Compute `s[i]` to be the sum from $r[0]$ to $r[i-1]$, inclusive, then `getsum(a,b)` simply returns `s[a+1]-s[b]`.

For the queries of the form `getmax(a,b)` asks the maximum elements between `r[a]` and `r[b]`, inclusive, the task is little more hard. The idea is always to get the max in some big ranges, so in the queries we may try to use these big ranges to compute fast. One simple algorithm is $T(n, \sqrt{n})$: Break the n numbers into $\sqrt{n}$ regions, each of size $\sqrt{n}$. Compute the champion for each region. For each query `getmax(a,b)`, we go from a towards right to the nearest station (at most $\sqrt{n}$ steps), then go by at most $\sqrt{n}$ stations (big regions) to the nearest station before b, and from there go to b. Below we give a nice $T(n, \log n)$ algorithm. [1] We pre-process in $\log n$ levels. On level i, all the blocks are of size 2^i, and each starting point is divisible by 2^i.

Outline: Range Max 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));
    }
}
```

6.3 Set Union and Find

In the beginning we have a collection of objects, each one is a singleton set on its own. From time to time, we want to perform the operations:
`FIND(x)` asks which set x belongs to, we require the program output a unique representative of the set that containing x.
`UNION(x,y)` to combine the two sets containing x and y together, if they were in different sets.

A naive solution is to represent the sets by trees. the representative of a set is the root of the tree. To find the root for a set we just need to follow the parent links. To union two sets, we just need to set the parent link of one root to the other root.

This implementation could be slow. But if we add the following two improvements, the running time will be very fast.
path compression: Every time we find r to be the root of x by follow the parent link up along a path P, we change all the nodes on P to be the direct child of r.
union by rank: There is a rank for each root, initially all 0. every time we want to combine two roots, we make the one with smaller rank the child to the one with bigger rank. If they are of the same rank, we pick any one to be the new root, but then we increase its rank by 1.

Outline: Set Find with Compression - Union by Rank

[1] With some extra work, this can be done in $T(n, 1)$, which was invented by our native Rutgers people Martin Farach-Colton.

```
int pa[30000], rk[30000], tCnt[30000]; // is it big enough?
// parent, rank, and number of nodes in the subtree (if it is the root)


void init(int n)
{
  int i;
  for(i=0;i<n;i++)
    {
      pa[i]=i; tCnt[i]=1; rk[i]=0;
    }
}

int find(int x)
{
  int a=x;
  while(pa[a]!=a) a=pa[a];
  int b=x,c;
  while(pa[b]!=a)
    {
      c=pa[b];
      pa[b]=a;
      b=c;
    }

  return(a);
}


void fUnion(int x, int y)
{
  int ax=find(x);
  int ay=find(y);
  if(ax==ay) return;
  if(rk[ay]>rk[ax]) swap(ax,ay);
  if(rk[ax]==rk[ay]) rk[ax]++;
  pa[ay]=ax;
  tCnt[ax]+=tCnt[ay];
}
```

6.4 String Matching

Given a long text `text` and a relatively short pattern `pat`, we want to know if `pat` occurs as a substring of `text`. The KMP algorithm first build an automaton (a profile) `f` for `pat`, then we can find `pat` in any text quickly.

Outline: the KMP String Matching Algorithm

```
char text[100000], pat[100];
int f[100];
/* kmpsetup: setup back tracking links, pat is the pattern,
   back tracing links are recorded in f. */
```

```
void kmpsetup (char *pat, int* f) {
  int i, k, len = strlen(pat);
  for (f[0] = -1, i = 1; i < len; i++) {
    k = f[i-1];
    while (k >= 0)
      if (pat[k] == pat[i-1]) break; else k = f[k];
    f[i] = k + 1;
  }
}
/* kmpscan: find substring pat in string text using
   back tracing link recorded in f. */
int kmpscan (char *pat, char *text, int *f) {
  int i, k, ret = -1, len = strlen(pat);
  for (i = k = 0; text[i];) {
    if (k == -1) { i++; k = 0; }
    else if (text[i] == pat[k]) {
      i++; k++;
      if (k >= len) { ret = i - len; break; }
    } else k = f[k];
  }
  return ret;
}
int main (void)
{
  int r;
  scanf ("%s %s", pat, text);
  kmpsetup (pat, f);
  r = kmpscan (pat, text, f);
  if (r == -1)
    printf ("Not Found\n");
  else
    printf ("substring starts at %d\n", r);
}
```

6.5 2D Arrays, Rectangles

Here we discuss several problems with rectangles. We start with the simplest one, then we introduce a general technique that makes the problem size smaller when the information is "sparse". And we further discuss some more problems.

6.5.1 Cover the Chessboard with Rectangles

Problem 6.1 *On a chessboard of size $N \times M$, we put several rectangles, they may intersect in various patterns. What is the total area covered by all these rectangles? How many connected components are there of all the squares covered by the rectangles? etc.*

This kind of problems are very easy when n and m are small (say, 100). One imagines that the original chessboard is a sheet of clean (white) paper, and those squares covered by some rectangle will become gray. When a rectangle comes, one just simulate the job of "cover" the chessboard by setting all the squares in the region to gray.

Let N and M be the size of the chessboard, and R be the number of rectangles. It becomes a problem when the size of the chessboard is very large. (e.g., it is bad enough if one merely needs to initialize a

2d array of size 10000×10000.) However, a basic observation is that the key factor here is R. We think the (huge) chess board as just the piece of paper, and draw lines according to the edges of the rectangles. Therefore, the R rectangles only introduces $2R$ horizontal lines and $2R$ vertical lines. These lines introduce a chessboard of size at most $2R \times 2R$, although each square is no longer a unit square, but a rectangle. For simplicity of the program, we usually work with a $4R \times 4R$ chessboard, where we do not differentiate the numbers (coordinates) introduced by the horizontal lines and vertical lines. We call these numbers the *critical numbers*.

The above technique works in general when we have problems in two dimensional plane or big arrays where the critical events are sparse.

Here are the variables we are going to use:

```
struct REC
{
    int x,y,xx,yy; // rectangle from x->xx, y->yy
    void input() {cin>>x>>y>>xx>>yy;}
};

REC rec[101];
map<int, int> mp;
vector<int> vt;
int bd[500][500];
```

`rec[]` are the rectangles; `bd[][]` is the chessboard we will play the simulation, `vt[]` is the vector of all the critical numbers in increasing order (there might be repeated critical numbers, it do not matter in general — by doing that we just introduced some empty rectangles); and `mp` maps a critical number to its position in `vt`, i.e., given a coordinate, tell us which row (column) on the chessboard does that coordinate corresponds to.

We first read all the rectangles and record the critical numbers.

```
for(i=0;i<R;i++) rec[i].input();
vt.clear();
for(i=0;i<R;i++)
{
    vt.push_back(rec[i].x); vt.push_back(rec[i].y);
    vt.push_back(rec[i].xx); vt.push_back(rec[i].yy);
}
```

Then we put a bounding box and sort the coordinates. And build the map from the coordinates to indices.

```
vt.push_back(-100000000);
vt.push_back(100000000);
sort(vt.begin(), vt.end());
mp.clear();
for(int c=0; c<vt.size(); c++)
    mp[vt[c]]=c;
```

Now we can play the simulation on the board — initialize the board and put the rectangles. Note that `bd[i][j]` records the status of the square from `(vt[i], vt[j])` to `(vt[i+1], vt[j+1])`.

```
for(i=0;i<500;i++) for(j=0;j<500;j++) bd[i][j]=0;
for(i=0;i<n;i++)
{
```

```
        for(x=mp[rec[i].x]; x<mp[rec[i].xx]; x++)
        for(y=mp[rec[i].y]; y<mp[rec[i].yy]; y++)
            bd[x][y]=1;
    }
```

6.5.2 2D Max Sum

Problem 6.2 *Given a two dimensional array* `a[][]`*, find the subarray (a rectangle of consecutive rows and columns) where the sum of the subarray is maximized.*

This is the 2D version of the problem in Section 1.4. The bad algorithm runs in $O(n^6)$ time by (1) pick all the subarrays ($O(n^4)$) and (2) sum the numbers inside each array ($O(n^2)$). By a preprocessing and inclusion/exclusion, we can reduce it to $O(n^4)$, where each step (2) only takes $O(1)$ time. (Let `s[i][j]` be the sum of all the numbers from the upper left corner to the position (i, j).)

In Section 1.4 we see how to solve the one dimensional max sum problem in $O(n)$ time. We borrow that to get an $O(n^3)$ time algorithm to solve Problem 6.2.

Outline: 2D Max Sum

(a). Preprocess the sum of vertical segments, `s[i][j][k]` to be the sum of all the numbers on the k-th column, from the i-th row to the j-th row. Notice that this takes $O()n^3$ time if you use something like `s[i][j][k]=s[i][j-1][k]+a[j][k]`.
(b) Now, the task reduces to the 1D max sum. We may fix any possible top edge and bottom edge i and j ($O(n^2)$ choices), try to find what is the rectangle with i as the top row, j as the bottom row. This is just the 1D max sum problem with elements `s[i][j][0..M-1]`.

Remark. In the problem about this kind, you always need to be careful about the boundary conditions, and issues like whether the coordinates should be off by 1. There are many possible solutions. A good practice is that always expand a 0 row on the top and a 0 column at the left; and always write down the meaning of your arrays (especially whether the coordinates are inclusive or exclusive) before you start the code.

6.5.3 Max Unaffected Rectangle

Problem 6.3 *Given a 0-1 matrix,* 0 *means forbidden or damaged. What is the maximum (area) rectangle we can find (with all 1's)?*

Having the previous section in your mind, the problem can be solved in a similar manner in $O(n^3)$ time, except both step (a) and (b) are simpler. In (a), `s[i][j][k]` becomes the indicator whether all the numbers on the k-th column, from the i-th row to the j-th row are 1. If you want to save space, you can just use `s[i][k]` to record the first 0 on the k-th column above the i-th row. (This can be processed in $O(n^2)$ time.) In (b), once we fix i and j, it becomes a problem of finding the longest consecutive 1's in an array of 0's and 1's.

From the Guide: In case your $O(n^3)$ algorithm runs beyond the lifetime of the universe, and you finished your lunch, and none of your teammates looks more clever than you...

Again, we use `s[i][k]` to record the first 0 on the k-th column above the i-th row. Now, for any fixed sea level i, `s[i][0..M-1]` gives a skyline of Manhattan. `s[i][k]` is the height of the building of width 1 at the k-th position. Now we have M buildings, thus $M+1$ boundaries. Initially each building k has boundary k and $k+1$. The we sort the buildings, from the highest to the lowest, and union them in that order.

Let `f[i]` be the *friend* of boundary i. It records the other end of the piece so far connected to i. (i.e., if i is the left boundary of a piece, then `f[i]` is the right boundary.) In the beginning, no piece is added, all `f[i]=i`.

From the tallest building to the shortest building we add them. When we add a building k, we connect the piece P_1 and P_2, where P_1 has k as the right bound, and P_2 has $k+1$ as its left bound. The new piece is from the left bound of P_1 to the right bound of P_2.

```
l = f[k]; r = f[k+1];
f[l] = r; f[r] = l;
```

Since we add the buildings from the higher to the lower ones, we can conclude that the all 1 rectangle which include (i, k) on the top edge can go as far as l to the left, and r to the right. So the all 1 rectangle we have in this case is of size $(r - l) \times s[i][k]$.

7 The Sample Programs

7.1 Combinatorics

Problem 7.1 (the Famous Joseph Problem) *[UVA 10940] In the Joseph Problem, n person are standing on a circle. We start from some people, every m-th person is going to be executed and only the life of the last remaining person will be saved. Which position will be the surviving position? There are many Joseph problems on UVA. The following is a nice one, which is Joseph's problem with $m = 2$ in disguise.*

Given is an ordered deck of n cards numbered 1 to n with card 1 at the top and card n at the bottom. The following operation is performed as long as there are at least two cards in the deck: Throw away the top card and move the card that is now on the top of the deck to the bottom of the deck. Your task is to find the last, remaining card.

Each line of input (except the last) contains a positive number $n \leq 500000$. The last line contains 0 and this line should not be processed. For each number from input produce one line of output giving the last remaining card. Input will not contain more than 500000 lines.

Sample Input:

```
7
19
10
6
0
```

Sample Output:

```
6
6
4
4
```

Remark. The idea is to simulate the Joseph game fast. In the first pass, all the odd numbers are killed, we are left with a similar problem with size about half of the original size. We can solve the smaller problem, and see how to get our answer based on the answer to the smaller problem.

In the program, `c[n]` is the survivor if there are n person, and we start the game from the first person (kill the first person); `d[n]` is the survivor if there are n person, and we start the game from the second person. Actually one array is enough, but then we need to do some shift back and forth, which is not so necessary when $m = 2$.

Sample Solution:

```
int c[500010], d[500010]; int n;

void play() {
    int i;
    c[1]=d[1]=1;
    c[2]=2; d[2]=1;
    for(i=3; i<500010; i++)
    {
        if(i%2)
        {
            c[i]=d[i/2]*2;
            d[i]=c[i/2+1]*2-1;
        }
        else
        {
            c[i]=c[i/2]*2;
            d[i]=d[i/2]*2-1;
        }
    }
}

int main() {
    play();
    while(cin>>n && n) cout<<c[n]<<endl;
}
```

Problem 7.2 (Basic Numbers) *Yijian finished his study in the primary school at the age of thirteen. At that time he learnt addition, multiplication and division. And he was introduced to the concept of prime numbers. Like most of us, he likes mathematics, he was so glad that, after one day's happy work, he was able to generate the first thirteen prime numbers - a great achievement for a thirteen-year-old boy. However, unlike most of us, he is not very good at math. Any thing beyond those primes appears too complicated for Yijian. He never thought about whether there are infinitely many primes; he is satisfied with the ability that, given any number, he can tell whether or not it is divisible by any of the first thirteen primes.*

The first thirteen prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, and 41; their product is 304250263527210. A number called basic if it is divisible by at least one of the first thirteen primes. Thus, the first number that is not basic is 1, and the second is 43. Yijian write all the basic numbers in ascending order in a (very long) list. Your task is to find out, given k, what is the k-th element in the list.

The input consists of up to 500 test cases. In each case there is a single number k. For each test case, output the k-th basic number on a single line. You may assume that the answer is never bigger than 304250263527210.

Sample Input:

```
2 3 8
2
3 42
```

Sample Output:

```
3
4
```

```
9
3
4
44
```

Remark. Yijian, my dear friend, is the first person who ever beat me seriously in any mathematical competition. Clearly anything said about his math ability in the statement is just my opinion.

Remark. The solution combines the binary search and inclusion-exclusion principle. Given a rank k, it is hard to find the k-th answer. But the inverse is somehow easy: Given a number, we can tell how many basic numbers are below it by inclusion-exclusion.

Sample Solution:

```
long long m[10000];
int sg[10000];
// m[Set]: the product of that set of primes, where the first 13 primes are labeled 0..12
// sg[Set]: 1 or -1, the parity of the set; will be used in Inclusion-Exclusion
int p[13]={2,3,5,7,11,13,17,19,23,29,31,37,41};

void init()
{
    int s, i;
    for(s=1;s<(1<<13);s++)
    {
        m[s]=1; sg[s]=-1;
        for(i=0;i<13;i++) if(s&(1<<i))
        {
            sg[s]=-sg[s];
            m[s]=m[s]*p[i];
        }
    }
    //cout<<m[(1<<13)-1]<<endl;
}

// play(wk) tells the number of basic numbers in the range 1..wk
// It looks like the inverse to the function we want to compute
long long play(long long wk)
{
    long long pt, cnt=0;
    int s;
    for(s=1;s<(1<<13);s++)
    {
        pt=wk/m[s];
        cnt=cnt+pt*sg[s];
    }
    return(cnt);
}

int main()
{
    init();
```

```
    long long k,lo,hi,mi,k0;
    while(cin>>k)
    {
        // Do a binary search on play() we get the answer.
        lo=1; hi=m[(1<<13)-1]+1;
        // assertion: from 1 to lo there will be less than k
        // from 1 to hi there will be no less than k
        while(lo+1<hi)
        {
            mi=(lo+hi)/2;
            k0=play(mi);
            if(k0<k) lo=mi; else hi=mi;
        }
        cout<<hi<<endl;
    }
}
```

7.2 Dynamic Programming

Problem 7.3 (Monotone Traveling Salesman in the Plane) *[Southeastern European Regional Programming Contest 2005] Given points in the plane, and assume there are no two points with the same x-coordinate. A Monotone tour is in two stages, in the first stage we start from the left most point, always go from the left to the right until we reach the rightmost point; and in the second stage we always go to the left until we come back to the leftmost point. The tour is required to visit each point exactly once.*

Sample Input:

```
3
1 1
2 3
3 1
4
1 1
2 3
3 1
4 2
```

Sample Output:

```
6.47
7.89
```

Remark. The problem `(a,b)` is defined for all $a \leq b$, to find the shortest (a, b)-thread, which means two paths, both starting from the leftmost point 0 and both goes from the left to the right. One of the paths ends at a, the other at b, and such that each point between 0 and b belongs to at least one of these paths. Note that we actually consider all the tours where each point is visited at least once, but in the optimal solution certainly no point will be visited twice.

Sample Solution:

```
struct PT {
```

```
    double x,y;
};

double dist(PT& p1, PT& p2) {
    return(sqrt(1.0*(p1.x-p2.x)*(p1.x-p2.x)+1.0*(p1.y-p2.y)*(p1.y-p2.y)));
}

vector<PT> pt; int n; double m[2005][2005];

double play(int a, int b) //a<=b
{
    double& r=m[a][b];
    int i;
    if(m[a][b]>-0.5) return m[a][b];
    r=1e+10;
    if(b>a+1)
    {
        r=play(a, b-1)+dist(pt[b], pt[b-1]);
        return r;
    }
    for(i=0;i<=a;i++)
    {
        r<?=play(i,a)+dist(pt[b], pt[i]);
    }
    return r;
}

int main() {
    int i,j;
    while(cin>>n)
    {
        pt.resize(n);
        for(i=0;i<n;i++) cin>>pt[i].x>>pt[i].y;
        for(i=0;i<n;i++) for(j=0;j<n;j++) m[i][j]=-1.0;
        m[0][0]=0.0;
        for(i=0;i<n;i++)
            for(j=i;j<n;j++)
                play(i,j);
        printf("%.2f\n", m[n-1][n-1]);
    }
    return 0;
}
```

Problem 7.4 (the Traveling Salesman on a Small Graph) *[UVA 10944] So as Ryan and Larry decided that they don't really taste so good, they realized that there are some nuts located in certain places of the island.. and they love them! Since they're lazy, but greedy, they want to know the shortest tour that they can use to gather every single nut!*

Input: You'll be given x, and y, both less than 20, followed by x lines of y characters each as a map of the area, consisting sorely of "`.`", "`#`", and "`L`". Larry and Ryan are currently located in "`L`", and the nuts are

represented by "#". They can travel in all 8 adjacent direction in one step. See below for an example. There will be at most 15 places where there are nuts, and "L" will only appear once.

Output: On each line, output the minimum amount of steps starting from "L", gather all the nuts, and back to "L".

Sample Input:

```
5 5
L....
#....
#....
.....
#....
```

Sample Output:

```
8
```

Remark. The problem will be more interesting if you are not allowed to touch a nut twice. In this case, the `r[s][i]` cannot be pre-computed, it also depends on the set S. We would need a shortest path problem nested in the DP.

Sample Solution:

```
int N,M;
char bd[50][50];
int x[20],y[20],r[20][20];
int n;
int m[1<<17][17];

int abs(int a)
{
    if(a<0) return -a;
    return a;
}

int play(int S, int s)
{
    if(S==(1<<s)) return r[s][0];
    int& v=m[S][s];
    if(v>=0) return v;
    v=100000000;
    for(int i=0;i<n;i++) if(i!=s) if(S&(1<<i))
        v=min(v,r[s][i]+play(S-(1<<s), i));
    return v;
}

int main()
{
    int i,j;
    while(cin>>N>>M)
```

```
    {
        for(i=0;i<N;i++) cin>>bd[i];
        n=1;
        for(i=0;i<N;i++) for(j=0;j<M;j++)
        {
            if(bd[i][j]=='L') {x[0]=i; y[0]=j;}
            if(bd[i][j]=='#') {x[n]=i; y[n]=j; n++;}
        }
        for(i=0;i<n;i++) for(j=0;j<n;j++)
            r[i][j]=max(abs(x[i]-x[j]), abs(y[i]-y[j]));
        memset(m, -1, sizeof(m));
        cout<<play((1<<n)-1, 0)<<endl;
    }
    return 0;
}
```

7.3 Graphs

7.3.1 BFS

Problem 7.5 (Dungeon Master) *[UVA 532] Find the shortest path in a 3D maze from 'S' to 'E'. It is clear from the sample*

Sample Input:

```
3 4 5
S....
.###.
.##..
###.#

#####
#####
##.##
##...

#####
#####
#.###
####E

1 3 3
S##
#E#
###

0 0 0
```

Sample Output:

```
Escaped in 11 minute(s).
Trapped!
```

Sample Solution:

```
int R, L, C;
char dun[40][40][40];
int d[40][40][40];
int dead, born;
int arrx[64000], arry[64000], arrz[64000];

void play(int x,int y,int z, int depth) {
  if(x<0 || y<0 ||z<0 || x>=R || y>=L||z>=C) return;
  if(dun[x][y][z] == '#') return;
  if(d[x][y][z] != -1) return;
  arrx[born]=x; arry[born]=y; arrz[born]=z;
  born++;
  d[x][y][z]=depth+1;
}


void bfs() {
  int x,y,z,dd;
  born=dead=0;
  memset(d, -1, sizeof(d));
  for(x=0; x<R; x++) for(y=0; y<L; y++) for(z=0; z<C; z++)
    if(dun[x][y][z] == 'S')
      {
        arrx[born]=x; arry[born]=y; arrz[born]=z;
        born++;
        d[x][y][z] = 0;
      }
  while(born>dead)
    {
      x=arrx[dead]; y=arry[dead]; z=arrz[dead]; dead++;
      dd=d[x][y][z];
      if(dun[x][y][z] == 'E')
      {
        cout<<"Escaped in "<<dd<<" minute(s)."<<endl;
        return;
      }
      play(x+1,y, z, dd); play(x-1, y, z, dd);
      play(x,y+1, z, dd); play(x, y-1, z, dd);
      play(x,y, z+1, dd); play(x, y, z-1, dd);
    }
  cout<<"Trapped!"<<endl;
}

int main() {
  while(cin>>R>>L>>C && R)
    {
      for(int i=0; i<R; i++)
        for(int j=0; j<L; j++)
            cin>>dun[i][j];
      bfs();
```

```
    }
}
```

7.3.2 DFS

Problem 7.6 (Cut Point) *[Greater New York ACM/ICPC 2000] We omit the problem statement. The task is clear from the sample.*

Sample Input:

```
1 2  5 4  3 1  3 2  3 4  3 5 0
1 2  2 3  3 4  4 5  5 1  0
1 2  2 3  3 4  4 6  6 3  2 5  5 1 0
0

Network #1
  SPF node 3 leaves 2 subnets

Network #2
  No SPF nodes

Network #3
  SPF node 2 leaves 2 subnets
  SPF node 3 leaves 2 subnets
```

Sample Solution:

```
// Finding the cut points in an undirected graph
// Assuming the original graph is connected;
// otherwise every point is a cut point if n>2

set<int> eg[1010]; // adjacency list (set)
int st; // starting node - the indices are not neccessary from 1 to n.
int v[1010], hh[1010], qq[1010];
// v[]: visited mark for dfs, 0: not visited; 1: gray; 2: finished
// hh[]: height in the dfs tree
// qq[a]: number of components after deleting a

int input() {
    int i,j;
    for(i=0;i<1010;i++) { eg[i].clear(); v[i]=0; qq[i]=0; }
    st=0;
    while(1)
    {
        cin>>i;
        if(i==0) break;
        cin>>j;
        st=i;
        if(i==j) continue;
        eg[i].insert(j);
        eg[j].insert(i);
```

```
    }
    return(st);
}

int dfs(int a, int h0)
// returns the highest level that any node in the
// subtree a can reach.
{
    v[a]=1; hh[a]=h0;
    set<int>::iterator it;
    int h, i, ret=h0, cp=0;
    for(it=eg[a].begin(); it!=eg[a].end(); it++)
    {
        i=(*it);
        if(!v[i])
        {
            h=dfs(i, h0+1);
            ret=min(ret, h);
            if(h>=h0) cp++;
            // the subtree i cannot reach anywhere else
            // note: there is no cross edge in undirected dfs tree
        }
        else if(v[i]==1)
        {
            ret=min(ret, hh[i]);
        }
    }
    if(h0>0) cp++;
    // if it's not the root, count the part that contains a's ancestors.
    qq[a]=cp;
    v[a]=2;
    return(ret);
}

int main() {
    int cs=0;
    int i,j;
    while(input())
    {
        dfs(st, 0);
        if(cs) cout<<endl;
        cs++;
        cout<<"Network #"<<cs<<endl;
        j=0;
        for(i=1;i<1010;i++)
        if(qq[i]>1)
        {
            cout<<"  SPF node "<<i<<" leaves "<<qq[i]<<" subnets"<<endl;
            j=1;
        }
        if(j==0) cout<<"  No SPF nodes"<<endl;
```

```
    }
    return(0);
}
```

7.3.3 Max Flow

Problem 7.7 (Crime Wave) *[UVA 563] Given an s by a grid, view as s streets and a avenues; and given b crossings where there is a robber. Is that possible to find b routes for the robbers to escape out of the grid such that no two routes touch each other?*

Input: The first line of the input contains the number of problems p to be solved.

The first line of every problem contains the number s of streets ($1 \leq s \leq 50$), followed by the number a of avenues ($1 \leq a \leq 50$), followed by the number b ($b \geq 1$) of robbers.

Then b lines follow, each containing the location of a robber in the form of two numbers x (the number of the street) and y (the number of the avenue).

Output: The output file consists of p lines. Each line contains the text `possible` or `not possible`. If it is possible to plan non-crossing get-away routes, this line should contain the word: possible. If this is not possible, the line should contain the words not possible.

Remark. . In the program we define `aj` as the adjacency list and `pc[i]` is the counter (size) of `aj[i]`. We know in this problem the maximum degree (not in the grid graph, but the actual graph we run the max flow) is at most 200. If in a problem you do not know any good upper bound of degrees, you may just change `aj` to be of type `vector<int>`.

You always need to estimate the size of the array you need in a problem. Here all the 5010 or 6000 are upper bounds of the number of vertices. (Again, not on the grid, but the two level graph you will run max flow.)

Sample Solution:

Some **remark** on how to use the max flow code:
1. Use `init(a)` to set up an empty graph of size a. The global variable `N` will be assigned a. For each edge from x to y with capacity c, use `addEdge(x, y, c)`. Always use 0 to be the source and $N-1$ to be the destination. After the set up, call `maxflow`. It returns the amount of max flow. The actual flow can be found in all the `eg[i].f` ($0 \leq i < N$) where `eg[i].f>0`.
2. You need to worry about the size of the arrays, but I think our definition here is big enough for most applications. However, we assume here the biggest degree for each vertex is 200. If you cannot upper bound this number, but still know max flow is the right solution, you can change `aj[][]` to be `vector<int> aj[5010]`, and `pc[]` disappears, it becomes `aj[].size()`; the line in the `init()` function becomes

```
for(i=0;i<N;i++) aj[i].clear();
```

3. If the capacity of the edges are not integers, you need to change some integer variables to `double` (or `long long`). The key is to change `f, c` in the definition of `struct edge` to be double, and the function `double maxflow()`. Then, any variable operated with them should be changed type. These include `ex[]`, `gg` in `push()`, `ans` in `maxflow()`, and `c` in `addEdge()`. Again, never compare two `double` numbers with `==` or `<`. You need to change the code accordingly.

Now, the program with the max flow.

```
// Crime Wave --- an example of max flow [preflow-push-label].
int N,M; // number of verts and edges
```

```
struct edge { int x,y, f,c, rev; };

edge eg[500000];
int aj[5010][200]; int pc[5010];
int phi[5010]; int ex[5010];
int mac[6000]; int ac[5010];
int dead, born;

void push(int a)
{
        int x=eg[a].x; int y=eg[a].y; int gg=ex[x];
        if(gg>eg[a].c) gg=eg[a].c;
        eg[a].f+=gg; eg[a].c-=gg;
        int k=eg[a].rev;
        eg[k].f-=gg; eg[k].c+=gg;
        ex[x]-=gg; ex[y]+=gg;
        if(ex[x]==0) {dead=(dead+1)%6000; ac[x]=0;}
        if(y && y<N-1 && ac[y]==0) {mac[born]=y; ac[y]=1; born=(born+1)%6000;}
}

int maxflow()
{
        int i,j,k,t1,t2,t3;
        //for(i=0;i<M;i++) eg[i].f=0;
        for(i=1;i<N;i++) { ex[i]=0;     ac[i]=0; }
        ex[0]=1000000000;
        dead=born=0;
        for(i=0, j=pc[0];i<j;i++)
                push(aj[0][i]);
        phi[0]=N;
        for(i=1;i<N;i++) phi[i]=0;
        while(dead!=born)
        {
                i=mac[dead];
                t2=100000000;
                for(t1=pc[i], j=0; j<t1; j++)
                {
                        k=aj[i][j];
                        if(eg[k].c==0) continue;
                        t3=phi[eg[k].y]+1;
                        if(t3<t2) t2=t3;
                        if(phi[i]==phi[eg[k].y]+1)
                        {
                                push(k);
                                j=t1+10;
                        }
                }
                if(j<t1+5) phi[i]=t2;
        }
        int ans=0;
        for(i=0, j=pc[0];i<j;i++)
```

```
        {
                k=aj[0][i];
                ans+=eg[k].f;
        }
        //cout<<ans<<endl;
        return(ans);
}

void init(int a)
{
        int i;
        N=a;
        for(i=0;i<N;i++) pc[i]=0;
        M=0;
}

void addEdge(int x, int y, int c)
{
        eg[M].x=x; eg[M].y=y; eg[M].c=c; eg[M].f=0;
        eg[M].rev=M+1; eg[M+1].rev=M;
        eg[M+1].x=y; eg[M+1].y=x; eg[M+1].c=0;
        eg[M+1].f=0;
        aj[x][pc[x]]=M; pc[x]++;
        aj[y][pc[y]]=M+1; pc[y]++;
        M+=2;
}


int n,m;
int B;
int oPt(int a, int b){ return(2*(a*m+b)+1); }
int iPt(int a, int b){ return(2*(a*m+b)+2); }

int main()
{
                int i,j,k;
                int q; cin>>q;
                while(q)
                {
                        q--;
                        cin>>n>>m;
                        init(2*m*n+2);
                        for(i=0;i<n;i++)
                                for(j=0;j<m;j++)
                                {
                                        k=oPt(i,j);
                                        addEdge(iPt(i,j),k,1);
                                        if(i==0) addEdge(k,N-1,1);
                                        else addEdge(k,iPt(i-1,j),1);
                                        if(i==n-1) addEdge(k,N-1,1);
                                        else addEdge(k,iPt(i+1,j),1);
```

```
                                        if(j==0) addEdge(k,N-1,1);
                                        else addEdge(k,iPt(i,j-1),1);
                                        if(j==m-1) addEdge(k,N-1,1);
                                        else addEdge(k,iPt(i,j+1),1);
                                }
                        cin>>B;
                        for(k=0;k<B;k++)
                        {
                                cin>>i>>j;
                                i--;j--;
                                if(B<=200) addEdge(0,iPt(i,j),1);
                        }
                        if(B>200) cout<<"not possible";
                        else if(maxflow()==B) cout<<"possible";
                        else cout<<"not possible";
                        cout<<endl;
                }
                return(0);
}
```

7.3.4 Min Cost Flow and Matching

Here is a problem I made some 8 years ago for the SJTU team. And I submitted it to ICPC regionals in 2004, it was used by several regions in USA. We have two solutions here, one use the cycle canceling, and the other is the standard successive shortest path algorithm for min cost flows.

Problem 7.8 (Going Home) *On a grid map there are n little men and n houses. In each unit time, every little man can move one unit step, either horizontally, or vertically, to an adjacent point. For each little man, you need to pay $1 travel fee for every step he moves, until he enters a house. The task is complicated with the restriction that each house can accommodate only one little man.*

Your task is to compute the minimum amount of money you need to pay in order to send these n little men into those n different houses. The input is a map of the scenario, a `'.'` *means an empty space, an* `'H'` *represents there is a house on that point, and am* `'m'` *indicates there is a little man on that point.*

You can think of each point on the grid map is a quite large square, so it can hold n little men at the same time; also, it is okay if a little man steps on a grid with a house without entering that house.

Sample Input:

```
2 2
.m
H.
5 5
HH..m
.....
.....
.....
mm..H
7 8
...H....
...H....
...H....
mmmHmmmm
```

```
...H....
...H....
...H....
```

Sample Output:

```
2
10
28
```

In the ACM/ICPC world final 2005, the legendary Robert Renaud solve a hard problem, part of the program uses the min cost flow code in the notes. The note, prepared by the legendary Ben Etin, is actually the max cost flow. Robert spent a long time and discovered this. The difference between max/min cost is just changing one '<' to '>' in the code. Be aware! I believe this note is correct, and the necessary change is commented in the code.

Sample Solution:

Some **remark** on how to use the min cost matching code:
1. Set `N` to be the number of vertices in both side, and `Cost[i][j]` to be the cost (or profit) from the i-th left vertex to the j-th right vertex. Then call `bestMacthing`. After the call, the best matching is stored in `mc[]`.
2. If the costs are real numbers, you need to change the variables `Cost[][]`, `dt`, and `ret` to `double`, as well as the return types of all the functions. Again, change all the comparisons of real numbers with ϵ tolerance.

```
// min cost flow -- cycle canceling

int N; // Number of houses / persons.
int Cost[110][110]; // Cost[i][j]: The distance between house i and person j
int cc[110];
int cccnt; // cc[] record the augumenting path, cccnt tells the length of cc[].
int mc[110]; // Current matching. House i matches to person mc[i].

int augCycle()
{
    // Find a positive cycle in the Delta graph.
    int i,j,k;
    int dt[110][110]; // Delta: Edge weights in the Augumenting Graph
    int fs[110][110]; // Parent link for the biggest paths
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
        {
            dt[i][j]=Cost[i][mc[i]]-Cost[i][mc[j]];
            fs[i][j]=j;
        }
    // Floyd
    for(k=0;k<N;k++)
        for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            if(dt[i][k]+dt[k][j]>dt[i][j])
            // !!!! change to < if want max cost matching
            {
                dt[i][j]=dt[i][k]+dt[k][j];
```

```
                fs[i][j]=fs[i][k];
                if(i==j)
                {
                    cccnt=0;
                    do
                    {
                        cc[cccnt]=i; cccnt++;
                        i=fs[i][j];
                    }while(i!=j);
                    return(1);
                }
            }
    return(0);
}

int bestMatching()
{
    int i,j;
    for(i=0;i<N;i++) mc[i]=i;
    while(augCycle())
    {
        j=mc[cc[0]];
        for(i=0;i<cccnt-1;i++)
            mc[cc[i]]=mc[cc[i+1]];
        mc[cc[i]]=j;
    }
    int ret=0;
    for(i=0;i<N;i++) ret+=Cost[i][mc[i]];
    return(ret);
}

char Map[40][40];

int main()
{
    int n,m,i,j,x,y,t1,t2;
    while(cin>>n>>m)
    {
        for(i=0; i<n; i++) cin>>Map[i];
        N = 0;
        for(i=0; i<n; i++)
            for(j=0; j<m; j++)
                if(Map[i][j] == 'H') N++;
        t1 = -1;
        for(i=0; i<n; i++)
            for(j=0; j<m; j++)
                if(Map[i][j] == 'H')
                {
                    t1++; t2 = -1;
                    for(x=0; x<n; x++)
                        for(y=0; y<m; y++)
```

```
                            if(Map[x][y] == 'm')
                            {
                                t2++;
                                Cost[t1][t2] = abs(i - x) + abs(j - y);
                            }
                }
        cout<<bestMatching()<<endl;
    }
    return(0);
}
```

Sample Solution:

Some **remark** on how to use the min cost matching code:
1. First, use `init(a)` to set `N` to be the number of points a and initialize the empty graph. Always use source as 0, and destination as $N-1$. For an edge $x \to y$ with capacity u and cost t, use `addEdge(x,y,u,t);`. After the setup, call `minCostFlow();`. It returns the minimum cost max flow. If you want to find out the actual amount of the max flow, sum up all the `f[0][i]` where `f[0][i]>0`. The actual flow is in the `f[][]` array.
2. The `long long` type is not needed in this problem. When the cost and capacities are not integers, you should change them to `double`.

```
// min cost flow -- successive shortest path with bellman-ford

long long f[210][210], c[210][210];
int N,M;
int ex[50010],ey[50010];
long long w[50010];
long long dist[210], aug[210];
int pa[210];
long long infty=(long long)(1e+18);


void init(int a)
{
    N=a; M=0;
    int i,j;
    for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        f[i][j]=c[i][j]=0;
}

long long bellmanFord()
{
    //is it possible neg cycle? No.. pf- shortest paths
    int i,j,k,fg;
    long long ww;
    for(i=0;i<N;i++) dist[i]=aug[i]=infty;
    dist[0]=0;
    do
    {
```

```
        fg=0;
        for(k=0;k<M;k++)
        {
            i=ex[k]; j=ey[k];
            if(f[i][j]<c[i][j])
            {
                if(f[i][j]<0) ww=-w[k]; else ww=w[k];
                if(dist[j]>dist[i]+ww)
                {
                        dist[j]=dist[i]+ww;
                        aug[j]=min(aug[i], c[i][j]-f[i][j]);
                        pa[j]=i;
                        fg=1;
                }
            }
        }
    }while(fg);
    if(dist[N-1]>=infty/2) return(0);
    int wk=N-1,wk1;
    while(wk)
    {
        wk1=pa[wk];
        f[wk1][wk]+=aug[N-1];
        f[wk][wk1]-=aug[N-1];
        wk=wk1;
    }
    return(dist[N-1]*aug[N-1]);
}

long long minCostFlow()
{
    long long ret=0;
    long long tt;
    while(1)
    {
        tt=bellmanFord();
        if(tt==0) break;
        ret+=tt;
    }
    return(ret);
}

void addEdge(int x, int y, int capa, int cost)
{
     c[x][y]+=capa;
     ex[M]=x; ey[M]=y; w[M]=cost;
     M++;
}

char Map[40][40];
```

```
int main()
{
    int n,m,i,j,x,y,t,t1,t2,cst;
    while(cin>>n>>m)
    {
        for(i=0; i<n; i++) cin>>Map[i];
        t = 0;
        for(i=0; i<n; i++)
            for(j=0; j<m; j++)
                if(Map[i][j] == 'H') t++;
        init(2*t+2);
        t1 = 0;
        for(i=0; i<n; i++)
            for(j=0; j<m; j++)
                if(Map[i][j] == 'H')
                {
                    t1++; t2 = t;
                    for(x=0; x<n; x++)
                        for(y=0; y<m; y++)
                            if(Map[x][y] == 'm')
                            {
                                t2++;
                                cst = abs(i - x) + abs(j - y);
                                addEdge(t1,t2,1,cst);
                                addEdge(t2,t1,1,cst);
                            }
                }
        for(i=1; i<=t; i++)
        {
            addEdge(0,i,1,0);
            addEdge(i,0,1,0);
        }
        for(i=t+1; i<=2*t; i++)
        {
            addEdge(2*t+1,i,1,0);
            addEdge(i,2*t+1,1,0);
        }
        cout<<minCostFlow()<<endl;
    }
    return(0);
}
```

7.4 Long Integers

Problem 7.9 (Number Base Conversion) *[Greater New York ACM/ICPC 2002]*

Write a program to convert numbers in one base to numbers in a second base. There are 62 different digits: `{ 0-9,A-Z,a-z }`.

Input: The first line of input contains a single positive integer. This is the number of lines that follow. Each of the following lines will have a (decimal) input base followed by a (decimal) output base followed by

a number expressed in the input base. Both the input base and the output base will be in the range from 2 to 62. That is (in decimal) $A = 10$, $B = 11$, $\cdots$, $Z = 35$, $a = 36$, $b = 37$, $\cdots$, $z = 61$ (0-9 have their usual meanings).

Sample Input:

```
3
62 2 abcdefghiz 10 16 12345678901234567890123456789012345678901234567890 16
35 3A0C92075C0DBF3B8ACBC5F96CE3F0AD2
```

Sample Output:

```
62 abcdefghiz
2 10111000001000101111100100101100111111001001100011010010001

10 12345678901234567890123456789012345678901234567890
16 3A0C92075C0DBF3B8ACBC5F96CE3F0AD2

16 3A0C92075C0DBF3B8ACBC5F96CE3F0AD2
35 333YMHOUE8JPLT7OX6K9FYCQ8A
```

Sample Solution:

```
int a,b; char sa[10000]; char sb[10000];

void rev(char s[]) {
    int l=strlen(s);
    for(int i=0; i<l-1-i; i++) swap(s[i],s[l-1-i]);
}

void multi(char s[], int k) {
    int i, c=0, d;
    for(i=0;s[i];i++)
    {
        d=(s[i]-'0')*k+c;
        c=d/b; d%=b;
        s[i]='0'+d;
    }
    while(c)
    {
        s[i]='0'+(c%b); i++;
        c/=b;
    }
    s[i]='\0';
}

void add(char s[], int k) {
    int i, c=k, d;
    for(i=0;s[i];i++)
    {
        d=(s[i]-'0')+c;
        c=d/b; d%=b;
```

```
        s[i]='0'+d;
    }
    while(c)
    {
        s[i]='0'+(c%b); i++;
        c/=b;
    }
    s[i]='\0';
}

void trans(char s[]) {
    int i;
    for(i=0;s[i];i++)
    {
        char& c=s[i];
        if(c>='A' && c<='Z') c='0'+10+(c-'A');
        if(c>='a' && c<='z') c='0'+36+(c-'a');
    }
}

void itrans(char s[]) {
    int i;
    for(i=0;s[i];i++)
    {
        char& c=s[i]; int d=c-'0';
        if(d>=10 && d<=35) c='A'+(d-10);
        if(d>=36) c='a'+(d-36);
    }
}

int main() {
    int q; cin>>q;
    int i,j;
    while(q)
    {
        q--;
        cin>>a>>b>>sa; sb[0]='0'; sb[1]='\0';
        cout<<a<<" "<<sa<<endl;
        trans(sa);
        for(i=0;sa[i];i++)
        {
            multi(sb, a);
            add(sb, sa[i]-'0');
        }
        rev(sb);
        itrans(sb);
        cout<<b<<" "<<sb<<endl;
        cout<<endl;
    }
    return 0;
}
```

Index

augmenting path, 17

Bellman-Ford, 9, 22
best triangulation, 3
BFS, 5, 20, 59
binary search, 47, 55
bipartite graph, 17

Cao, Yijian, 54
Cayley's Formula, 15
connected components, 6
    strongly, 7
cut point, 9, 61
cycle canceling, 21, 67

determinant, 31
DFS, 6, 8, 9, 18, 61
    tree, 7
Dijkstra, 10, 15

edge-disjoint paths, 20
Edmonds-Karp, 20
electric network, 34
Euler's Φ function, 23
Euler's Theorem, 23

Farach-Colton, Martin, 48
Fermat's Little Theorem, 22
find and union, 17, 48
Floyd-Warshall, 10

greatest common divisor, 23
    extended, 23

Hamilton cycle, 5

inclusion-exclusion, 55

Joseph problem, 53

KMP, 49
Kruskal, 16

longest common subsequence, 2
    in linear space, 3
longest increasing subsequence, 1, 4

matching, 17, 19
    bipartite, 17
    maximum, 19
Matrix-Tree Theorem, 15
max bipartite matching, 17, 18, 20
    lexicographically smallest, 18
max flow, 19, 63
    integer, 19
    with capacity on vertices, 20
Max Flow Min Cut Theorem, 19
max sum
    one dimensional, 3
    two dimensional, 52
method of relaxation, 34
min cost bipartite matching, 21, 22, 66
min cost flow, 22, 66
min cut, 19, 21
minimum average cycle, 14
minimum spanning tree, 15

negative cycle, 9, 10

preflow-push-relabel, 20
Prim, 15
prime test, 23, 25

random walk, 34
range query, 47
rectangles in the plane, 50
Renaud, Robert, 31, 67
repeated squaring, 24, 31
residue network, 20

shortest path, 9, 10
    DAG, 12
    lexicographically smallest, 12
    within certain number of steps, 13
sieve of Erastothenes, 23
string matching, 49
subset sum, 1
successive shortest path, 22, 69
system of linear equations, 31, 34

topological sort, 4, 7
traveling salesman, 4, 57
    monotone, Euclidean, 56
trigonometric functions, 45

vertex cover, 19
    minimum, 19
vertex-disjoint paths, 20