Task 3: `TASKSAUTHOR`

There are many programming contests in the world today. Setting a good programming contest task is not easy. One challenge is the setting up of *test data*. A good test data should be able to differentiate a code that meets the goals, from another seemingly correct code that fails in some special cases.

In this task, your role in a contest is reversed! As an experienced programmer, you are helping the Happy Programmer Contest's committee in setting up their test data. The committee has selected two graph problems with a total of 8 different subtasks, and has written a few codes that seemingly solve the graph problems. In designing a subtask, the committee has the intention that some of the codes would get all the points, whereas some would gain zero or some points. You are given all those codes in C, C++ and Pascal versions[1]. For each subtask, your job is to produce a test data X that *differentiates* two given codes, code **A** and code **B**. More specifically, the following two conditions must be met:

1. On input X, code **A** must not lead to Time Limit Exceeded (TLE).
2. On input X, code **B** must lead to TLE.

In addition, the committee prefers smaller test data, with a target of at most T integers in the test data.

The two problems selected by the committee are the Single-Source Shortest Paths (SSSP) problem, and a graph problem we called the Mystery problem. The pseudo-codes of the codes written by the committee are listed in the appendix and the C, C++ and Pascal implementations can be found in the attached zip file that accompanies task 3 in the grading server.

Subtasks

Please refer to Table 1. Each row describes a subtask. Note that subtask 1 to 6 are on the SSSP problem whereas subtask 7 and 8 are on the Mystery problem. The number of points allocated for the subtasks are listed in column S.

Subtask	Points S	Target T	Problem	Code **A**	Code **B**
1	3	107	SSSP	ModifiedDijkstra	FloydWarshall
2	7	2222	SSSP	FloydWarshall	OptimizedBellmanFord
3	8	105	SSSP	OptimizedBellmanFord	FloydWarshall
4	17	157	SSSP	FloydWarshall	ModifiedDijkstra
5	10	1016	SSSP	ModifiedDijkstra	OptimizedBellmanFord
6	19	143	SSSP	OptimizedBellmanFord	ModifiedDijkstra
7	11	3004	Mystery	Gamble1	RecursiveBacktracking
8	25	3004	Mystery	RecursiveBacktracking	Gamble2

Table 1: The 8 Subtasks.

[1]All codes implement algorithms that are permitted inside the IOI syllabus.

To get any point for a subtask, your test data X must be able to differentiate the corresponding code **A** and code **B**. In addition, the number of points you received depends on the number of signed integers in X. Suppose X contains F integers, S points are allocated to the subtask, and T is the targeted size, then the number of points awarded is calculated as follow:

$$\lfloor (S/100) \times \lfloor min(100 \times T/F, 100) \rfloor \rfloor.$$

where $\lfloor\ \rfloor$ denotes the rounding down operation. Hence, if your test data X contains not more than T integers, the full S points are awarded.

Grading

You must name each of your eight test data as `tasksauthor.outX.1` where `X` is the subtask number. Before submission, you are required to compress your test data files using `gzip`. In Unix systems, the following command produces the compressed file:

```
    tar -cvzf tasksauthor.tgz tasksauthor.out*.1
```

In Windows systems, use software like 7-Zip or Winzip to produce this tar-gzipped archive. Submit only the file `tasksauthor.tgz` to the grading server.

The grading server will unpack the compressed file. For each subtask, say subtask `X`, the grader carries out the following steps to determine the number of points to be awarded for subtask `X`:

C1. If test data `tasksauthor.outX.1` does not exist, halts and no point will be awarded.

C2. Checks the format of `tasksauthor.outX.1`.
If the input format is invalid, halts and no point will be awarded.

C3. Runs code **A** with `tasksauthor.outX.1` as the input.
If TLE is triggered, halts and no point will be awarded.

C4. Runs code **B** with `tasksauthor.outX.1` as the input.
If TLE is triggered, halts and awards a number of points calculated using the formula:

$$\lfloor (S/100) \times \lfloor min(100 \times T/F, 100) \rfloor \rfloor.$$

All the provided codes maintain a counter (the variable `counter`) that keeps track of the number of performed operations. During the execution of a code, when the value of the counter exceeds 1,000,000, then we consider the code has triggered TLE.

Problem Statement 1: Single-Source Shortest Paths (SSSP)

Given a directed weighted graph G and two vertices s and t in G, let $p(s,t)$ be the shortest path weight from the "source" s to the "destination" t. If t is not reachable from s, then $p(s,t)$ is defined to be 1,000,000,000. In this problem, the input is the graph G and a sequence of Q queries $(s_1,t_1),(s_2,t_2),\ldots,(s_Q,t_Q)$. The output is the corresponding query results $p(s_1,t_1),p(s_2,t_2),\ldots,p(s_Q,t_Q)$.

Input/Output file Format

The input file consists of two blocks. The first block describes the adjacency list of a directed weighted graph G. The second block describes shortest path queries on G.

The first block starts with an integer V in one line, which is the number of vertices in G. The vertices are labelled as $0,1,\ldots,V-1$. Then, V lines follow where each line corresponds to a vertex, starting from vertex 0. Each line starts with n_i that describes how many outgoing-edges the vertex i has. Next, n_i pairs of integers (j,w) follow where each pair corresponds to an outgoing-edge. The first integer j in a pair is the label of the vertex the edge points to, and the second integer w is the edge's weight.

The second block starts with an integer Q in one line. Next, Q lines follows. The k-th line contains two integers s_k and t_k, corresponding to the source and destination vertex respectively.

Any two consecutive integers in one line must be separated by at least one space. Additionally, the input satisfies the following:

1. $0 < V \leq 300$,
2. n_i is a non-negative integer $\forall i \in [0..V-1]$,
3. $0 \leq j < V$,
4. $|w| < 10^6$ where $|w|$ denotes the absolute value of w,
5. $0 \leq \sum_{i=0}^{V-1} n_i \leq 5000$,
6. $0 < Q \leq 10$,
7. $0 \leq s_k < V$, $0 \leq t_k < V$, $\forall k \in [1..Q]$, and
8. the graph G must **not** have any negative weight cycle.

Recall that the grading server will check for the above constraints in step C2.

The output file format is less relevant in this task. Anyway, the output consists of Q lines, and the k-th line contains the integer $p(s_k,t_k)$. For convenience, the provided codes will print out the value of the variable `counter` at the end of the output.

Sample Input File[2]

```
3
2 1 4 2 1
0
1 1 2
2
0 1
1 0
```

Sample Output File[3]

```
3
1000000000
The value of counter is: 5
```

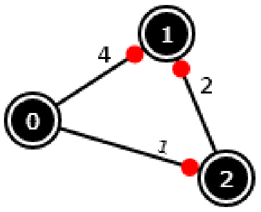

Figure 1: Directed Weighted Graph from the Sample Input File

[2]There are fifteen integers in this input file, therefore $F = 15$.

[3]The value of counter is 5 when the sample input file given above is run on ModifiedDijkstra.cpp/pas.

Problem Statement 2: Mystery

Given an undirected input graph G with V vertices and E edges, label each vertex in G with an integer $\in [0..(X-1)]$ so that no two endpoints of any edge in G has the same label. The value of X must be the lowest possible for graph G.

Input/Output file format

The input file starts with two integers V and E in one line. Then, E lines follows. Each line contains two integers a and b that denotes an *undirected* edge (a, b) in G. In addition, the input satisfies the following constraints (to be checked in step C2):

1. $70 < V < 1000$,
2. $1500 < E < 10^6$, and
3. for any edge (a, b), we have $a \neq b$, $0 \leq a < V$, $0 \leq b < V$, and it appears only once in G.

The output file starts with an integer X in one line, the smallest integer so that vertex labelling is feasible. The next line contains V integers that describe the integer label of vertex 0, vertex 1, ..., vertex $V-1$, and the last line is the value of `counter`.

Sample Input File[4]

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

Sample Output File[5]

```
3
0 1 2 1
The value of counter is: 18
```

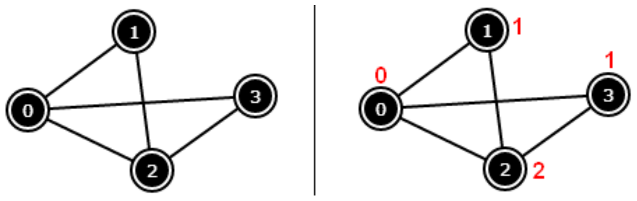

Figure 2: Left: Undirected Graph from the Sample Input File; Right: Its Labels with $X = 3$

[4]There are twelve integers in this input file, therefore $F = 12$. However, this small sample input file is only used for illustration. It is not valid as its V and E values are too small.

[5]The value of counter is 18 when the sample input file given above is run on RecursiveBacktracking.cpp/pas.

Appendix: Pseudo-codes

Here are the algorithms of the provided codes. The variable `counter` "approximates" the runtime by keeping track of some operations. Our grading server uses the C++ version of these implementation codes.

FloydWarshall.cpp/pas

```
// pre-condition: the graph is stored in an adjacency matrix M
counter = 0
for k = 0 to V-1
   for i = 0 to V-1
      for j = 0 to V-1
         increase counter by 1;
         M[i][j] = min(M[i][j], M[i][k] + M[k][j]);
for each query p(s,t)
   output M[s][t];
```

OptimizedBellmanFord.cpp/pas

```
// pre-condition: the graph is stored in an adjacency list L
counter = 0
for each query p(s,t);
   dist[s] = 0; // s is the source vertex
   loop V-1 times
      change = false;
      for each edge (u,v) in L
         increase counter by 1;
         if dist[u] + weight(u,v) < dist[v]
            dist[v] = dist[u] + weight(u,v);
            change = true;
      if change is false // this is the 'optimized' Bellman Ford
         break from the outermost loop;
   output dist[t];
```

ModifiedDijkstra.cpp/pas

```
// pre-condition: the graph is stored in an adjacency list L
counter = 0;
for each query p(s,t)
   dist[s] = 0;
   pq.push(pair(0, s)); // pq is a priority queue
   while pq is not empty
      increase counter by 1;
      (d, u) = the top element of pq;
      remove the top element from pq;
      if (d == dist[u])
         for each edge (u,v) in L
            if (dist[u] + weight(u,v) ) < dist[v]
               dist[v] = dist[u] + weight(u,v);
               insert pair (dist[v], v) into the pq;
   output dist[t];
```

Gamble1.cpp/pas

```
Sets X = V and labels vertex i in [0..V-1] with i;
Sets counter = 0; // will never get TLE
```

Gamble2.cpp/pas

```
Sets X = V and labels vertex i in [0..V-1] with i;
Sets counter = 1000001; // force this to get TLE
```

RecursiveBacktracking.cpp/pas

```
This algorithm tries X from 2 to V one by one
and stops at the first valid X.

For each X, the backtracking routine label vertex 0 with 0,
then for each vertex u that has been assigned a label,
the backtracking routine tries to assign
the smallest possible label up to label X-1 to its neighbor v,
and backtracks if necessary.

// Please check RecursiveBacktracking.cpp/pas to see
// the exact lines where the iteration counter is increased by 1
```