МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

«Санкт-Петербургский национальный исследовательский университет

информационных технологий, механики и оптики»

**ФАКУЛЬТЕТ СИСТЕМ УПРАВЛЕНИЯ И РОБОТОТЕХНИКИ**

**ЛАБОРАТОРНАЯ РАБОТА**

**(1450, 1521, 1160)**

Выполнили:

Хюинь Тан Куонг (336231)

Преподаватель:

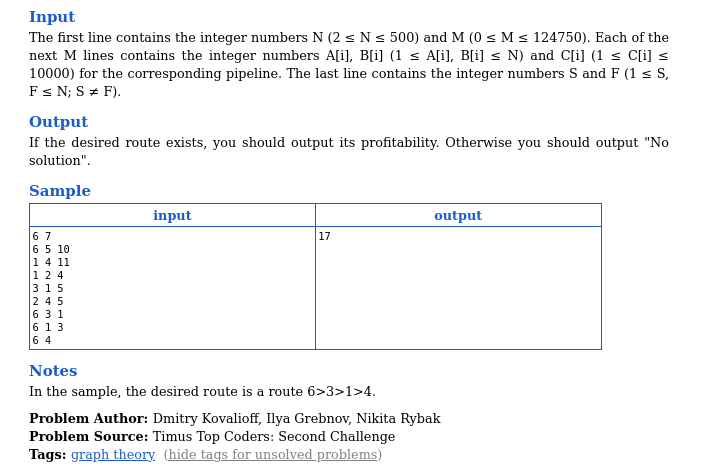
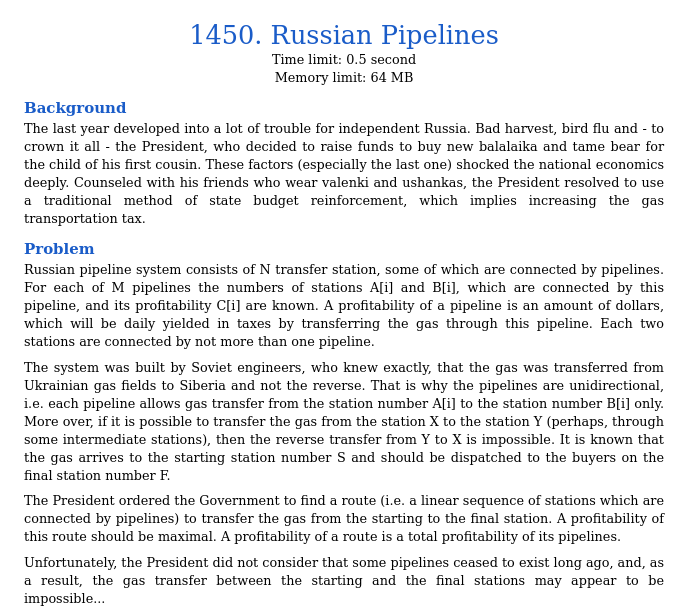
Tropchenko Andrey Aleksandrovich



Санкт-Петербург, 2023

# 1450. Russian Pipelines

## Problem description



## Timus system acceptance:



## Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <inttypes.h>

#define maxN 501

#define negative\_infinity -1000000000000

int graph[maxN][maxN];

unsigned int n, m;

int64\_t d[maxN];

void print\_graph(int n) {

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

for (int j = 1; j <= n; j++) {

printf("%d ", graph[i][j]);

}

printf("\n");

}

}

void Ford\_Bellman() {

int stop;

do

{

stop = 1;

for (unsigned int u = 1; u <= n; u++) {

for (unsigned int v = 1; v <= n; v++) {

if (graph[u][v] != 0) {

if (d[v] < d[u] + graph[u][v]) {

d[v] = d[u] + graph[u][v];

stop = 0;

}

}

}

}

} while (!stop);

}

int main() {

// init graph

memset(graph, 0, sizeof(graph));

scanf("%u %u", &n, &m);

for (unsigned int i = 0; i < m; i++) {

unsigned int a, b, c;

scanf("%u %u %u", &a, &b, &c);

graph[a][b] = c;

}

unsigned int u, v;

scanf("%u %u", &u, &v);

// init array d

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

d[i] = negative\_infinity;

}

d[u] = 0;

// print\_graph(n);

Ford\_Bellman();

if (d[v] <= 0) {

printf("No solution");

}

else {

printf("%ld\n", d[v]);

}

// printf("%ld\n", d[v]);

return 0;

}

## Input:

6 7

6 5 10

1 4 11

1 2 4

3 1 5

2 4 5

6 3 1

6 1 3

6 4

## Output:

17

## Algorithm description:

* To solve this problem, we can use the Ford-Bellman to find the path with the maximum profit.
* The Ford-Bellman algorithm (also known as Bellman-Ford) is an algorithm for finding the shortest path in a directed graph that can have negative weights. However, if we want to find the path with the largest weight, we can apply some small changes to the algorithm.
* Specifically, to find the path with the largest weight, we can change the Bellman-Ford weight update step from "plus" to "subtract". This means that instead of updating the weight by adding the weight of the edge in question with the weight of the current vertex as in Bellman-Ford, we will update the weight by subtracting the weight of the edge from the weight. number of current peaks.
* Specifically, the Ford-Bellman algorithm to find the most weighted path is as follows:

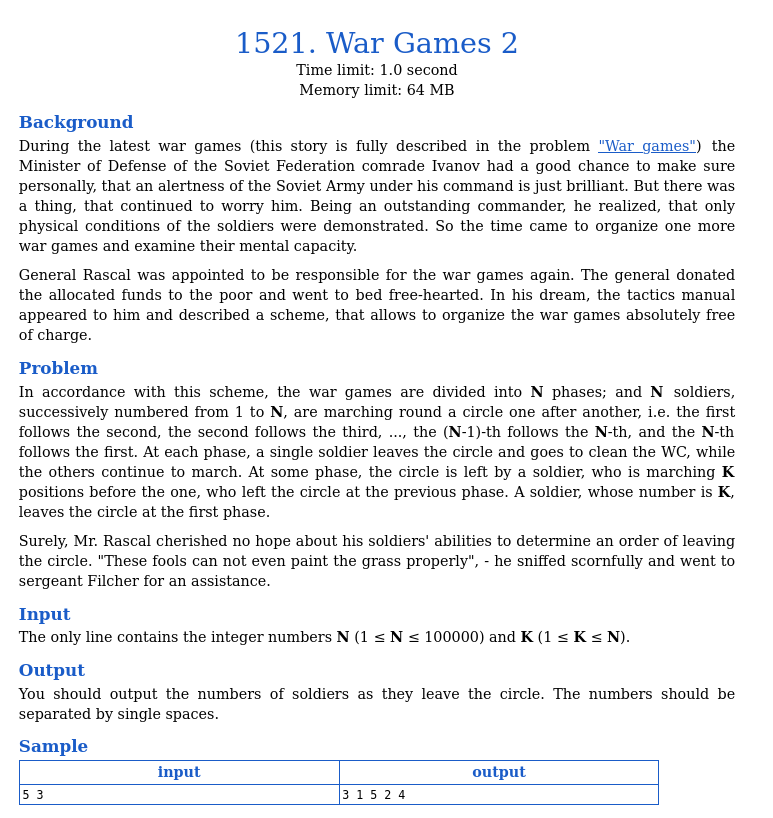
Initialize the dist distance array with the value -INF for each vertex in the graph, except for the starting vertex whose distance is zero.

Repeat V - 1 time, where V is the number of vertices in the graph. In each iteration, we traverse each edge of the graph and update the distance of the vertices adjacent to that edge by the formula: dist[v] = max(dist[v], dist[u] - w(u, v)), where u is the endpoint vertex of the edge, v is the end vertex of the edge, and w(u, v) is the weight of the edge.

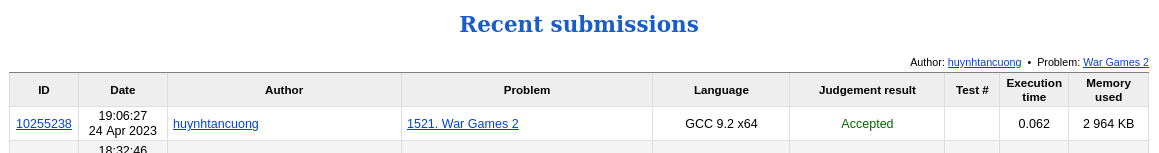
Check if there is a negative cycle in the graph. To do this, we iterate through the edges and check if there is a pair of vertices whose distance from the endpoint can be updated by traversing the edge. If yes, then the graph contains a negative cycle.

# 1521 War Games 2.

## Problem description



## Timus system acceptance:



## Code:

#include <stdio.h>

#include <stdlib.h>

#define maxN 100000

typedef struct node {

int data;

int value;

} node;

node a[maxN];

node st[4 \* maxN];

void build(int id, int l, int r) {

// if this is a leaf

if (l == r) {

st[id] = a[l];

return;

}

// build the 2 child

int mid = (l+r) >> 1;

build(2\*id, l, mid);

build(2\*id+1, mid+1, r);

// update the value of current node

st[id].value = st[2\*id].value + st[2\*id+1].value;

}

int query(int pos, int id, int l, int r) {

// if this is a leaf

if (l == r) {

return st[id].data;

}

// if pos belong to left child

if (pos <= st[2\*id].value) {

int mid = (l+r) >> 1;

return query(pos, 2\*id, l, mid);

}

// if pos belong to right child

else {

int mid = (l+r) >> 1;

return query(pos-st[2\*id].value, 2\*id+1, mid+1, r);

}

}

void delete(int pos, int id, int l, int r) {

// if this is a leaf

if (l == r) {

st[id].value = 0;

return;

}

// if pos belong to left child

if (pos <= st[2\*id].value) {

int mid = (l+r) >> 1;

delete(pos, 2\*id, l, mid);

}

// if pos belong to right child

else {

int mid = (l+r) >> 1;

delete(pos-st[2\*id].value, 2\*id+1, mid+1, r);

}

// update the value of current node

st[id].value = st[2\*id].value + st[2\*id+1].value;

}

int main() {

int n, k;

scanf("%d %d", &n, &k);

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

a[i].data = i;

a[i].value = 1;

}

build(1, 1, n);

int pos\_to\_remove = 1;

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

pos\_to\_remove = (pos\_to\_remove - 1 + k) % st[1].value;

if (pos\_to\_remove == 0) pos\_to\_remove = st[1].value;

printf("%d ", query(pos\_to\_remove, 1, 1, n));

delete(pos\_to\_remove, 1, 1, n);

}

return 0;

}

## Input:

5 3

## Ouput:

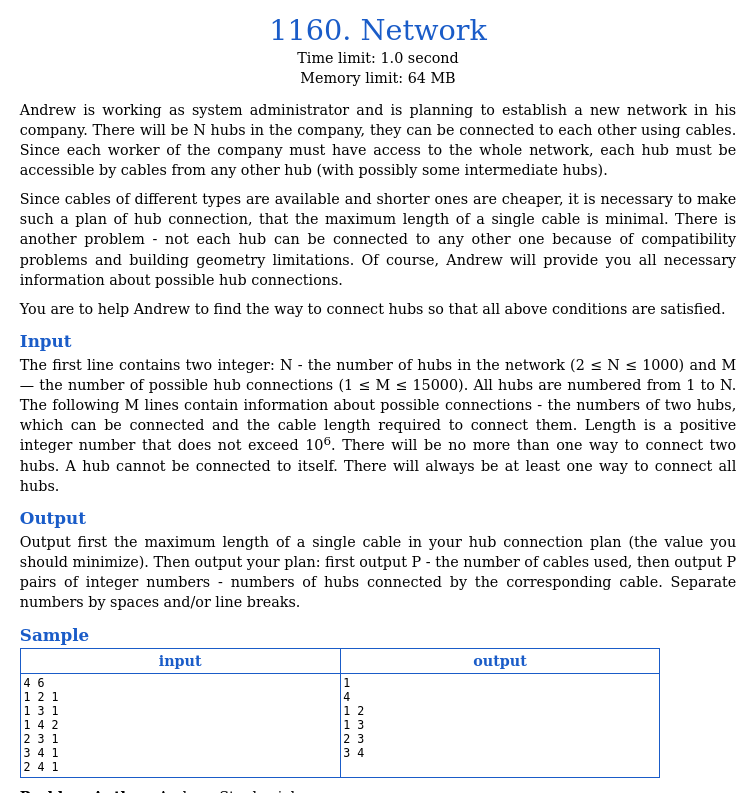
5 3

## Algorithm description:

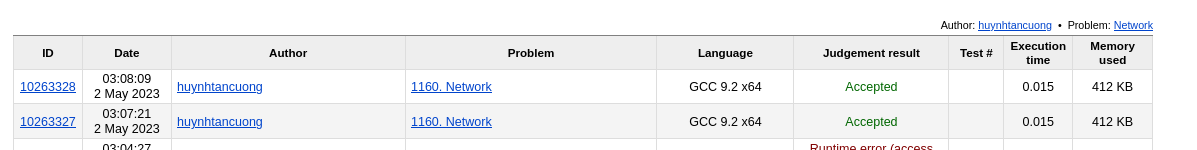
* To solve this problem, we can use the data structure “Interval tree” or “Segment Tree” to represent every soldier as a node.
* We have 3 main functions:
  + Build(): to build the segment tree into the st[] array
  + Query(pos): to get the data value of the pos-th node.
  + Delete(pos): to delete pos-th node out of the segment tree.
* We create a loop to keep deleting the node:
  + pos\_to\_remove = (pos\_to\_remove - 1 + k) % st[1].value;
* After the loop finished, we printed out all the soldier.

# 1160.

## Problem description



## Timus system acceptance:



## Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define maxN 2000

#define maxM 15000

// Struct for Edge

typedef struct {

int u, v, c; // hai dinh va trong so

int mark; // danh dau co duoc ket nap vao khung cay hay chua

} Edge\_s;

// Global variables

int Lab[maxN];

int Count[maxN];

Edge\_s E[maxM];

int count = 0;

int connected = 0;

int maximum\_length = 0;

void swap(Edge\_s \*a, Edge\_s \*b) {

Edge\_s temp = \*a;

\*a = \*b;

\*b = temp;

}

void printArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

void BuildMaxHeap(Edge\_s arr[], int n) {

for (int i = n / 2; i >= 0; i--) {

maxHeapify(arr, n, i);

}

}

void maxHeapify(Edge\_s arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if ((left < n) && (arr[left].c > arr[i].c))

largest = left;

else

largest = i;

if ( (right < n) && (arr[right].c > arr[largest].c))

largest = right;

if (largest != i) {

swap(&arr[largest], &arr[i]);

maxHeapify(arr, n, largest);

}

}

void Heapsort(Edge\_s arr[], int n) {

BuildMaxHeap(arr, n);

for (int i = n-1; i>=0; i--) {

swap(&arr[0], &arr[i]);

n--;

maxHeapify(arr, n, 0);

}

}

void Kruskal(Edge\_s E[], int n, int m) {

for (int i = 0; i < m; i++) {

int r1 = getRoot(E[i].u);

int r2 = getRoot(E[i].v);

if (r1 != r2) { // canh E[i] noi hai cay khac nhau

E[i].mark = 1;

count++;

if (count == n-1) { // Neu da dem du so canh

connected = 1;

break;

}

merge(r1, r2);

}

}

}

int getRoot(int v) {

while (Lab[v] > 0) {

v = Lab[v];

}

return v;

}

void merge(int r1, int r2) {

if (Count[r1] < Count[r2]) { // merge r1 into r2

Count[r2] = Count[r1] + Count[r2];

Lab[r1] = r2;

}

else { // merge r2 into r1

Count[r1] = Count[r2] + Count[r1];

Lab[r2] = r1;

}

}

void printResult(int n, int m) {

if (!connected) {

printf("0\n");

}

else {

// printf("Minimal spanning tree:\n");

int count = 0;

int W = 0;

for (int i = 0; i < m; i++) {

if (E[i].mark == 1) {

W += 1;

count++;

if (maximum\_length < E[i].c) {

maximum\_length = E[i].c;

}

}

if (count == n-1) {

break;

}

}

printf("%d\n", maximum\_length);

printf("%d\n", W);

count = 0;

for (int i = 0; i < m; i++) {

if (E[i].mark == 1) {

count++;

printf("%d %d\n", E[i].u, E[i].v);

}

if (count == n-1) {

break;

}

}

}

}

void loadGraph(int m) {

for (int i = 0; i < m; i++) {

scanf("%d %d %d", &E[i].u, &E[i].v, &E[i].c);

}

}

void init(int n, int m) {

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

Lab[i] = -1; // Rừng ban đầu, mọi đỉnh là gốc của cây gồm đúng một nút

Count[i] = 1;

}

for (int i = 0; i < m; i++) {

E[i].mark = 0;

}

}

int main() {

int n, m;

scanf("%d %d", &n, &m);

loadGraph(m);

init(n, m);

Heapsort(E, m);

Kruskal(E, n, m);

printResult(n, m);

return 0;

}

## Input:

4 6

1 2 1

1 3 1

1 4 2

2 3 1

3 4 1

2 4 1

## Ouput:

1

4

1 2

1 3

2 3

3 4

## Algorithm description:

* To solve this problem, we use Kruskal Algorithm to find the smallest spanning tree.
* The Kruskal algorithm is an algorithm for finding the smallest spanning tree in a weighted undirected graph. The main idea of the Kruskal algorithm is to construct the smallest spanning tree by linking the edges in ascending order of their weights.
* The Kruskal algorithm is described as follows:
* Sort the edges of the graph in ascending order of weights.
* Initialize an empty spanning tree.
* Traverse each edge in the sorted list of edges, if that edge does not form a cycle with previously selected edges (previously selected edges form a tree), we add that edge to the spanning tree and mark that edge as selected.
* Repeat step 3 until we have selected enough n-1 edges, where n is the number of vertices of the graph. Then we will get a minimum spanning tree of the graph.
* Note that the Kruskal algorithm not only allows to find the minimum spanning tree, but also allows to find all spanning trees of the graph. To do this, we need to modify step 4 of the algorithm by repeating step 3 until we have selected enough n-1 edges, or until the list of edges has been traversed. Then we will obtain all spanning trees of the graph.