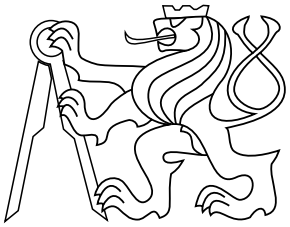
Programovací úlohy v Pokročilé Algoritmizaci

Elektronická skripta předmětu B4M33PAL

Marko Berezovský, Daniel Průša

2. října 2025



Katedra kybernetiky
Fakulta elektrotechnická
České vysoké učení technické v Praze

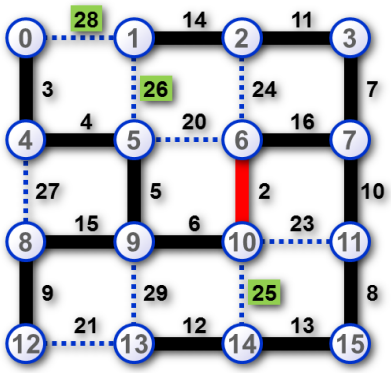
Skripta vznikají ve spolupráci s posluchači PAL
během zimního semestru 2025/2026.

Obsah

1 Náhradní propojení

Úspěšný projekt propojení správních budov na území Graphlandu využil nejmodernější kvantové kabely. Mezi každými dvěma budovami existuje spojení, ale nemusí být přímé. Typicky, signál na cestě z budovy A do budovy B může procházet i dalšími budovami. Na začátku projektu byl sestaven seznam dvojic budov a každé dvojici byla přiřazena pevná hodnota představující cenu kabelového propojení mezi těmito dvěma budovami. Po dokončení projektu jsou nyní všechny budovy propojeny pomocí pouze některých dvojic ze seznamu a celková cena všech kabelů v síti je minimální možná.

Krátce po dokončení sítě se ukázalo, že nejlevnější kabel v síti je nespolehlivý a jeho nevratné selhání je jen otázkou času. Ztráta tohoto kabelu by vedla k rozdělení sítě na dvě oddělené části, které by spolu nemohly komunikovat. Byla proto sestavena technická komise, která navrhla následující řešení: Pokud nejlevnější kabel selže, bude zakoupen a nainstalován náhradní kabel. Ten povede mezi některou takovou dvojicí budov z původního seznamu, které nyní nejsou přímo propojené. Dvojice budov musí být vybrána tak, aby náhradní kabel mezi nimi opět spojil oddělené části sítě a jeho cena nebyla ani příliš vysoká, ani příliš nízká. Po delších konzultacích s finančním oddělením byly stanoveny přesné dolní a horní hranice ceny možné náhradní hrany.



Obrázek 1: Schématický příklad propojení budov. Stávající kabely jsou znázorněny tučnými čarami, všechny ostatní dvojice budov v seznamu jsou vyznačeny čárkovaně. Nejlevnější kabel je zobrazen červeně. Dolní a horní cenový limit pro náhradní kabel je 24 a 28, možné polohy náhradního kabelu jsou označeny cenami zvýrazněnými zeleně.

Úloha

Je dán seznam možných propojení mezi dvojicemi budov a jejich cen. Dále jsou dány dolní a horní hranice ceny náhradního kabelu. Najděte v daném seznamu všechny dvojice budov, které splňují podmínky dané technickou komisí.

Vstup

První řádek vstupu obsahuje čtyři kladná celá čísla N, M, $C1$, $C2$ oddělená mezerami. N je počet budov, M je počet dvojic budov, které lze přímo propojit kabelem, $C1$ a $C2$ jsou dolní a horní hranice ceny náhradního kabelu. Předpokládáme, že budovy jsou očíslovány $0, 1, 2, \ldots, N-1$.
Následuje seznam dvojic budov, který se skládá z M řádků, z nichž každý obsahuje tři celá čísla A, B, C oddělená mezerami, přičemž A a B představují čísla budov, C představuje cenu kabelu, který přímo propojuje A a B. Pořadí dvojic budov v seznamu je libovolné. Žádná dvojice budov se v seznamu nevyskytuje více než jednou. Žádné dvě ceny kabelů v seznamu nejsou shodné.
Hodnota N nepřekročí 2000, hodnota M nepřekročí 1.5×10^6.

Výstup

Výstup obsahuje více řádků. První řádek obsahuje jedno celé číslo – celkovou cenu původního dokončeného propojení budov. Tato cena je vždy kladná a menší než 10^9. Následuje jeden nebo více řádků. Každý z nich obsahuje dvě čísla budov, které mohou být propojeny náhradním kabelem, a cenu tohoto kabelu, ve stejném formátu jako na vstupu úlohy. Hodnoty jsou odděleny mezerami,

menší číslo budovy se píše vždy jako první. Řádky jsou seřazeny vzestupně lexikograficky, přičemž každé celé číslo je považováno za jeden symbol. Je zaručeno, že vždy existuje alespoň jedna dvojice budov, která splňuje podmínky úlohy.

Příklad 1

Vstup

```
16 24 24 28
0 1 28
1 2 14
2 3 11
4 5 4
5 6 20
6 7 16
8 9 15
9 10 6
10 11 23
12 13 21
13 14 12
14 15 13
0 4 3
4 8 27
8 12 9
1 5 26
5 9 5
9 13 29
2 6 24
6 10 2
10 14 25
3 7 7
7 11 10
11 15 8
```

Výstup

```
135
0 1 28
1 5 26
10 14 25
```

Příklad 2

Vstup

```
8 13 8 11
0 1 11
1 4 10
4 7 6
7 6 8
6 3 9
3 0 7
0 2 12
2 5 1
5 7 13
1 2 2
2 3 4
4 5 3
5 6 5
```

Výstup

```
28
1 4 10
3 6 9
```

Příklad 3

Vstup

```
5 10 105 125
0 1 3
0 2 4
0 3 6
0 4 9
1 2 130
1 3 120
1 4 110
2 3 7
2 4 5
3 4 8
```

Výstup

```
18
1 3 120
1 4 110
```

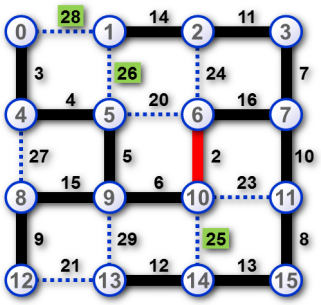
Řešení úlohy

— Abstraktní jádro úlohy —

V daném ohodnoceném neorientovaném grafu G určete váhu minimální kostry. Dále, v řezu definovaném pomocí hrany s minimální cenou v G, najděte všechny hrany, jejichž cena leží v daném rozmezí $\langle C1, C2\rangle$. Hrana h, která je součástí kostry T grafu G, definuje řez $R(h)$ v grafu G takto: Po odstranění hrany h z kostry T se kostra rozpadne na dvě komponenty A a B. Součástí řezu $R(h)$ jsou právě všechny takové hrany grafu G, jejichž jeden koncový uzel leží v komponentě A a druhý v komponentě B.

— Rozbor a postup —

Pro vyřešení úlohy použijeme jednoduchou modifikaci Primova algoritmu. Sledujme obrázek v zadání. Označme nejlacinější hranu v kostře (a, b). Na obrázku je to hrana (10, 6). Po odebrání nejlacinější hrany (a, b) se kostra rozpadá na dvě části. V jedné části jsou všechny uzly, které jsou blíže k uzlu a než k uzlu b, ve druhé části jsou všechny uzly, které jsou blíže k uzlu b než k uzlu a. Na obrázku jsou to v jedné části uzly 0, 4, 5, 8, 9, 10, 12 a ve druhé části uzly 1, 2, 3, 6, 7, 11, 13, 14, 15. Vzájemné vzdálenosti uzlů přitom měříme výhradně v kostře. Takže např. vzdálenost uzlu 11 od uzlu 6 je rovna $10 + 16 = 20$ a vzdálenost uzlu 11 od uzlu 10 je rovna $10 + 16 + 2 = 22$.



Obrázek 2: Schéma zadání

Každá možná záložní hrana spojuje dva uzly z různých částí. Pro nalezení všech možných náhradních hran tedy stačí projít seznam všech hran v grafu a vybrat jen ty, které spojují uzly z různých částí a kromě toho splňují cenový požadavek. Přitom předpokládáme, že u každého uzlu bude zaznamenáno, do které části patří. Tuto informaci lze každému uzlu určit již při samotném budování minimální kostry. Ve standardní podobě Primova algoritmu si u každého uzlu poznamenáváme jeho předchůdce, abychom mohli později, v případě potřeby, minimální kostru rekonstruovat. V našem případě ale vlastně strukturu minimální kostry znát nepotřebujeme. Místo jeho bezprostředního předchůdce budeme u každého uzlu poznamenávat, zda leží blíže k uzlu a nebo k uzlu b minimální hrany, to jest ve které ze dvou zmíněných částí leží. Při inicializaci v počáteční fázi modifikovaného Primova algoritmu vložíme do fronty všechny sousedy uzlu a a u všech těchto sousedů, kromě uzlu b, zaznamenáme, že leží blíže k a. U uzlu b ovšem zaznamenáme, že leží blíže k b. Dále již postup probíhá stejně jako ve standardním Primově algoritmu, z fronty vyjmeme aktuální uzel, a všem jeho sousedům, kteří dosud nebyli uzavřeni, aktualizujeme jejich cenu připojení k vznikající minimální kostře. Pokud se cena připojení souseda k vznikající minimální kostře zmenší, t.j. pokud dojde k relaxaci hrany mezi aktuálním uzlem a sousedem, pak zároveň tento soused převezme od aktuálního uzlu informaci, ke kterému z uzlů a, b minimální hrany je aktuální uzel blíže. Stejně jako ve standardním Primově algoritmu skončíme, když budou uzavřeny všechny uzly.

Všimněme si, že díky tomu, že nezaznamenáváme bezprostředního předchůdce každého uzlu v minimální kostře, vlastně celou úlohu řešíme, aniž bychom tuto minimální kostru vybudovali nebo nějak reprezentovali v paměti.

— Asymptotická složitost —

Naše modifikace samotného Primova algoritmu nemění jeho asymptotickou složitost, ta zůstává $\mathcal{O}(m \cdot \log(n))$, kde m a n představují po řadě počty hran a uzlů v grafu. Závěrečné procházení hran má složitost $\mathcal{O}(m)$, přičemž ale je třeba hrany předtím uspořádat do pořadí požadovaného na výstupu. Asymptotická složitost řazení hran nepřesáhne $\mathcal{O}(m \cdot \log(m))$. Přitom víme, že $m < n^2$, tudíž $\log(m) < 2 \cdot \log(n)$, tudíž asymptotická složitost řazení nepřesáhne asymptotickou složitost modifikovaného Primova algoritmu. Asymptotická složitost celého řešení je tedy $\mathcal{O}(m \cdot \log(n))$.

— Implementační poznámky —

V předloženém kódu je řazení nalezených hran na výstupu vyřešeno tak, že ihned po načtení grafu jsou u každého uzlu položky v seznamu jeho sousedů uspořádány v rostoucím pořadí podle identifikátorů sousedů. Před závěrečným výstupem hledaných hran již pak hrany není nutno zvlášť řadit do požadovaného pořadí, stačí prostý průchod seznamem sousedů aktuálního uzlu. Přitom aktuální uzly se procházejí v rostoucím pořadí jejich identifikátorů, jak je ostatně běžné.

Java kód

```
1  package pal;
2
3  import java.util.*;
4  import java.io.*;
5
6  public class Main{
7
8  // ------------------------------------------------------------------
9  // Limits defined in problem assignment, unused in the code:
10 // ------------------------------------------------------------------
11 static final int nodesLIMIT    =       2000;
12 static final int edgesLIMIT    =    1500000;
13 static final int edgeCostLIMIT = 1000000000;
14 static final int outputLIMIT   = 1000000000;
15
16 // ------------------------------------------------------------------
17 // Problem solution variables
```

```
// ------------------------------------------------------------------

static Graph graph;
static int C1, C2;
static int MSTcost;
static int CheapestEdgeCost, CheapestEdgeNode1, CheapestEdgeNode2;

// For each node, we have to register on which side in the MST,
// relative to the  cheapest edge, the node is located.  It is either
// the side connected directly to CheapestEdgeNode1 or to CheapestEdgeNode2.
// The array is indexed by node IDs.
static int [] nodeSide;

// ------------------------------------------------------------------
//     M A I N
// ------------------------------------------------------------------

public static void main(String[] args) throws IOException {
  load();
  solution();
  output();
}

// ------------------------------------------------------------------
//     L O A D
// ------------------------------------------------------------------

static void load()throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  StringTokenizer st = new StringTokenizer(br.readLine());
  int n  = Integer.valueOf(st.nextToken());
  int m  = Integer.valueOf(st.nextToken());
  C1     = Integer.valueOf(st.nextToken());
  C2     = Integer.valueOf(st.nextToken());

  graph = new Graph( n );
  nodeSide = new int [graph.N];

  int n1, n2, cost;
  CheapestEdgeCost = Integer.MAX_VALUE;

  for(int i = 0; i < m; i++) {
    st   = new StringTokenizer(br.readLine());
    n1   = Integer.valueOf(st.nextToken());
    n2   = Integer.valueOf(st.nextToken());
    cost = Integer.valueOf(st.nextToken());
    graph.addedge( n1, n2, cost );
    if (cost < CheapestEdgeCost) {
      CheapestEdgeNode1 = n1;
      CheapestEdgeNode2 = n2;
      CheapestEdgeCost  = cost;
    }
  }

  if( CheapestEdgeNode1 > CheapestEdgeNode2 ){
     swap              = CheapestEdgeNode1;
     CheapestEdgeNode1 = CheapestEdgeNode2;
     CheapestEdgeNode2 = swap;
  }

  graph.sortNodeNeighboursInAscendingIdOrder();

  // mere technicality: normalize cheapest edge info:
  if( CheapestEdgeNode1 > CheapestEdgeNode2 ){
     swap              = CheapestEdgeNode1;
     CheapestEdgeNode1 = CheapestEdgeNode2;
```

```
    CheapestEdgeNode2 = swap;
  }

}

// ---------------------------------------------------------------------
//     S O L U T I O N
// ---------------------------------------------------------------------

static void solution(){
  int  currnode, neigh,  currEdgeCost; // current edge = (currnode, neigh)
  long currnodecode;
  int NodesInSpanningTree;
  long queueCodeFactor = graph.N ;
  int UNDEF = -1;                    // denotes an undefined integer value

  // Each node in the priority queue is represented as [nodeCost, nodeId],
  // and coded as long integer nodeCost*queueCodeFactor + nodeId, where
  // queueCodeFactor is a factor greater than or equal to the number of nodes.

  PriorityQueue<Long> nodesQ = new PriorityQueue<>();
  boolean [] closed          = new boolean[graph.N];
  int [] nodeCost            = new int[graph.N]; // node dist to the emerging MST
  Arrays.fill( nodeCost, Integer.MAX_VALUE );

  // initialize nodeSide
  Arrays.fill( nodeSide, UNDEF );
  nodeSide[CheapestEdgeNode1] = CheapestEdgeNode1;

  // Extended initialization:
  // Add all neighbors of CheapestEdgeNode1 to the queue.
  for( int j = 0; j < graph.dg[CheapestEdgeNode1]; j++ ){
    neigh           = graph.edge[CheapestEdgeNode1][j];
    nodeCost[neigh] = graph.cost[CheapestEdgeNode1][j];
    nodeSide[neigh] = nodeSide[CheapestEdgeNode1];
    nodesQ.add( nodeCost[neigh] * queueCodeFactor + neigh );
  }
  nodeSide[CheapestEdgeNode2] = CheapestEdgeNode2; // overwrite nodeside!
  closed[CheapestEdgeNode1]   = true;
  NodesInSpanningTree         = 1;

  // standard Prim's algorithm loop
  while( NodesInSpanningTree < graph.N ){

    // pop current node from queue and update MST cost
    currnodecode = nodesQ.poll();
    currnode = (int)( currnodecode % queueCodeFactor );
    MSTcost += (int)( currnodecode / queueCodeFactor );
    NodesInSpanningTree += 1;

    // check all neighbours of current node
    for( int j = 0; j < graph.dg[currnode]; j++ ){
      neigh        = graph.edge[currnode][j];
      currEdgeCost = graph.cost[currnode][j];
      if( !closed[neigh] && nodeCost[neigh] > currEdgeCost ) {
          nodeCost[neigh] = currEdgeCost;
          nodesQ.add( currEdgeCost * queueCodeFactor + neigh );
          // modification of Prim's algorithm:
          // register on which side of min edge in MST the node is
          nodeSide[neigh] = nodeSide[currnode];
      } // if
    }// for neighs

    closed[currnode] = true;
  } // while
}
```

```
150
151 // ---------------------------------------------------------------------
152 //     O U T P U T
153 // ---------------------------------------------------------------------
154
155 static void output() {
156   int neigh, ecost;
157   System.out.printf( "%d \n", MSTcost );
158   for( int id = 0; id < graph.N; id++ )
159     for( int j = 0; j < graph.dg[id]; j++ ){
160       neigh = graph.edge[id][j];
161       ecost = graph.cost[id][j];
162       if(    id < neigh
163           && nodeSide[id] != nodeSide[neigh]
164           && C1 <= ecost && ecost <= C2
165           && !( id == CheapestEdgeNode1 && neigh == CheapestEdgeNode2 )
166         )
167         System.out.printf( "%d %d %d\n", id, neigh, ecost );
168     } // for j
169 }
170
171 } // end of class Main
172
173 // ---------------------------------------------------------------------
174 //     G R A P H   class
175 // ---------------------------------------------------------------------
176
177 public class Graph {
178
179 int N, M;              // number of nodes and edges, respectively
180 int [][] edge, cost;
181 int []   dg;
182
183 public Graph (int n ){
184   this.N    = n;
185   this.M    = 0;
186   this.edge = new int[n][2];
187   this.dg   = new int[n];
188   this.cost = new int[n][2];
189 }
190
191 public void addedge(int n1, int n2, int cc) {
192   // extend array of neighbours of a node, when array is too short
193   if( this.dg[n1] == this.edge[n1].length ) {
194     this.edge[n1] = Arrays.copyOf( this.edge[n1], 2*this.dg[n1] );
195     this.cost[n1] = Arrays.copyOf( this.cost[n1], 2*this.dg[n1] );
196   }
197   if( this.dg[n2] == this.edge[n2].length ) {
198     this.edge[n2] = Arrays.copyOf( this.edge[n2], 2*this.dg[n2] );
199     this.cost[n2] = Arrays.copyOf( this.cost[n2], 2*this.dg[n2] );
200   }
201   // Register the edge
202   this.cost[n1][this.dg[n1]]   = cc;
203   this.cost[n2][this.dg[n2]]   = cc;
204   this.edge[n1][this.dg[n1]++] = n2;
205   this.edge[n2][this.dg[n2]++] = n1;
206   this.M++;
207 }
208
209 public void sortNodeNeighboursInAscendingIdOrder(){
210   long [] sortarr = new long [this.N];
211   long storageFactor = 4294967296L; // 2^32
212
213   for( int id = 0; id < this.N; id++ ){
214     for( int j = 0; j < this.dg[id]; j++ )
215       sortarr[j] = this.edge[id][j]*storageFactor + this.cost[id][j];
```

```
216       Arrays.sort( sortarr, 0, this.dg[id] );
217       for( int j = 0; j < this.dg[id]; j++ ){
218         this.edge[id][j] = (int) (sortarr[j] / storageFactor);
219         this.cost[id][j] = (int) (sortarr[j] % storageFactor);
220       } // for j
221     } // for id
222   }
223
224   }
```

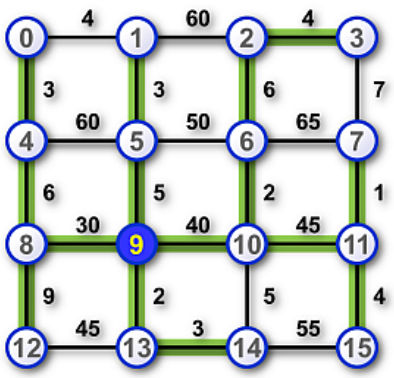
2 Laserové dálnice

Všechna města v zemi jsou propojena dálnicemi. Každá dálnice spojuje právě dvě města, přičemž žádné dvě dálnice se nekříží mimo město. V dopravním systému země se používá pojem tzv. index města. Index města je roven minimálnímu počtu dálnic, které musí vozidlo projet, aby se dostalo z daného města do hlavního města. Jinými slovy, index města je roven jedné plus minimálnímu počtu měst (kromě výchozího města a hlavního města), přes která musí vozidlo projet při cestě z daného města do hlavního města. Index hlavního města je roven 0.

Hlavní město má zásadní ekonomický význam a síť dálnic má být modernizována. Některé pečlivě vybrané dálnice budou vybaveny kvantovými laserovými navigačními systémy. Takto modernizované dálnice nazýváme laserové dálnice. Množina všech laserových dálnic se nazývá laserová síť. Cena modernizace každé dálnice není stejná a byla stanovena pro každou dálnici v zemi zvlášť. Cena laserové sítě je součet všech modernizačních cen dálnic, které do laserové sítě patří. Laserová síť bude velmi nákladná a pravděpodobně nebude obsahovat všechny existující dálnice. Na druhou stranu však síť musí poskytovat uživatelům adekvátní výhody. Úřady se proto snaží nalézt rozumný kompromis.

Laserová síť musí splňovat následující tři pravidla:

1. Z každého města v zemi musí být možné cestovat po laserové síti do hlavního města.
2. Pokud vozidlo cestuje po laserové síti z libovolného města do hlavního města, musí být možné zvolit takovou posloupnost laserových dálnic, že vozidlo nikdy nevjede do města, jehož index je vyšší než index kteréhokoli města, které vozidlo opustilo dříve během cesty. (Posloupnost indexů navštívených měst je tedy neklesající při pohybu směrem k hlavnímu městu.)
3. Cena laserové sítě musí být minimální možná.



Obrázek 3: Schéma měst a dálnic. Je zde 16 měst a 24 dálnic. Hlavní město má označení 9 a je zvýrazněno. Laserová síť, zvýrazněná zeleně, je nejlevnější laserová síť, která splňuje zadané podmínky. Schéma ilustruje Příklad 1 uvedený níže.

Úloha

Je dán seznam všech dálnic v zemi a cena modernizace každé dálnice na laserovou dálnici. Určete minimální možnou cenu laserové sítě.

Vstup

První řádek vstupu obsahuje tři celá čísla N, M, C, kde N je počet měst, M je počet dálnic, C je označení hlavního města. Města jsou označena celými čísly $0, 1, 2, \ldots, N-1$.
Seznam všech dálnic je uveden na následujících M řádcích. Každý z těchto řádků obsahuje tři celá čísla a, b, p oddělená mezerami, kde a a b jsou označení dvou měst spojených dálnicí, p je cena modernizace dálnice mezi těmito dvěma městy na laserovou dálnici. Pořadí dvojic měst v seznamu je libovolné.
Platí: $N \leq 2 \times 10^4$, $M \leq 2 \times 10^6$. Cena každé modernizace je kladné číslo, které nepřesahuje 10^3.

Výstup

Výstup obsahuje jeden řádek s jedním celým číslem – minimální možnou cenou laserové sítě.

Příklad 1

Vstup

```
16 24 9
0 1 4
1 2 60
2 3 4
4 5 60
5 6 50
6 7 65
8 9 30
9 10 40
10 11 45
12 13 45
13 14 3
14 15 55
0 4 3
1 5 3
2 6 6
3 7 7
4 8 6
5 9 5
6 10 2
7 11 1
8 12 9
9 13 2
10 14 5
11 15 4
```

Výstup

```
163
```

Příklad 2

Vstup

```
7 9 6
0 1 5
2 3 5
4 5 90
4 6 5
5 6 100
0 2 10
2 4 5
1 3 10
3 5 5
```

Výstup

```
120
```

Příklad 3

Vstup

```
10 22 6
1 6 16
0 1 15
1 7 17
6 8 10
3 4 10
2 4 13
2 5 14
4 6 13
4 5 19
5 8 18
5 6 13
1 9 18
0 9 11
0 8 15
0 7 18
2 3 17
1 4 10
3 6 12
1 3 11
1 8 12
2 6 16
3 9 13
```

Výstup

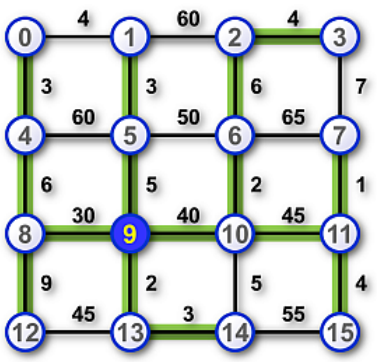
```
109
```

Řešení úlohy

— Abstraktní jádro úlohy —

Každému uzlu v grafu je přiřazena jeho hranová vzdálenost od hlavního města. Hranová vzdálenost mezi dvěma uzly je rovna počtu hran na nejkratší cestě mezi těmito dvěma uzly. Máme najít cenu takové kostry grafu, ve které podél každé cesty z hlavního města do libovolného jiného uzlu je neklesající posloupnost hranových vzdáleností z hlavního města do uzlů této cesty, a přitom je cena této kostry minimální možná.

— Rozbor a postup —



Obrázek 4: Schéma zadání

Sledujme obrázek. Můžeme si představovat, že uzly jsou rozděleny do jednotlivých vrstev podle svých hranových vzdáleností od hlavního města. V každé vrstvě mají všechny uzly stejnou hranovou vzdálenost od hlavního města, tu označíme jako index vrstvy. Je zřejmé, že hlavní město sousedí s každým uzlem ve vrstvě s indexem 1. Každý uzel ve vrstvě s indexem 2 sousedí s nějakým uzlem ve vrstvě s indexem 1. Obecně, Každý uzel ve vrstvě s indexem i ($0 < i$) sousedí s alespoň jedním uzlem ve vrstvě $i - 1$. V obrázku v zadání leží ve vrstvě s indexem 1 uzly 5, 8, 10, 13. Ve vrstvě s indexem 2 leží uzly 1, 4, 6, 11, 12, 14. Ve vrstvě s indexem 3 leží uzly 0, 2, 7, 15. Ve vrstvě s indexem 4 leží pouze uzel

3. Uzly grafu tak máme rozdělené do jednotlivých vzájemně disjunktních vrstev. O hranách grafu teď platí, že krajní uzly každé hrany leží buďto ve vrstvě se stejným indexem nebo ve vrstvách sousedních, jejichž index se navzájem liší právě o 1.

V obrázku v zadání není žádná hrana spojující uzly ve stejné vrstvě, v grafu v příkladu 2 takové hrany jsou (0, 1), (2, 3), (4, 5). Index uzlu pro nás bude index vrstvy, ve které uzel leží, indexem hrany bude pro nás maximum z indexů obou jejích krajních uzlů. V obrázku v zadání je index hrany (5, 6) roven 2, index hrany (3, 7) je roven 4, apod. Index hrany bude zároveň určovat její prioritu. Hrana s nižším indexem má vyšší prioritu než hrana s vyšším indexem, bez ohledu na cenu těchto dvou hran. Pokud mají dvě hrany stejný index, má vyšší prioritu hrana s nižší cenou. Například, nejvyšší prioritu tak budou mít všechny hrany incidentní s hlavním městem, nejnižší prioritu budou mít hrany incidentní s uzly s maximálními indexy v grafu. K vyřešení celé úlohy stačí zjistit cenu minimální kostry, v níž se místo cen hran uvažují pouze jejich priority. Abychom viděli, že postup funguje, představme si, že k hledání minimální kostry použijeme Primův algoritmus. Ten nejprve připojí optimálně k hlavnímu městu všechny uzly ve vrstvě s indexem 1. Zároveň, díky tomu, jak je konstruována priorita hran, algoritmus nebude připojovat uzly z vrstev s vyššími indexy. Analogická situace nastane pro libovolnou vrstvu s indexem k větším než 1. Dokud nebudou minimální kostrou propojeny všechny uzly ve vrstvách s indexy $k-1$ a nižšími, nebudou se uzly ve vrstvě k připojovat k minimální kostře. Potom, dokud nebudou minimální kostrou propojeny všechny uzly ve vrstvě s indexem $k-1$ a nižšími, nebudou se uzly ve vrstvě s indexem k nebo vyšším připojovat k minimální kostře. Bude tak zaručeno splnění podmínky úlohy, že cestovatel, který cestuje takto vybudovanou minimální kostrou směrem k hlavnímu městu, bude vždy nějakou dobu setrvávat ve vrstvě s indexem k a poté přejde do vrstvy s indexem $k-1$, tedy blíže k hlavnímu městu, a nikdy se nedostane do města s indexem větším než k. Protože použijeme algoritmus hledání *minimální* kostry, bude tak zaručeno, že najdeme minimální, ze všech koster, které splňují danou podmínku cestování v kostře. Zároveň, během budování kostry, budeme místo priority každé hrany přidané do kostry, započítávat do celkové ceny kostry pouze původní cenu této hrany.

— Asymptotická složitost —

Určení priorit hran má stejnou složitost jako BFS, jímž se určí indexy uzlů, tedy $\mathcal{O}(n+m)$), kde n a m představují po řadě počty uzlů a hran v grafu. Algoritmus hledání minimální kostry bude obohacen jen o přidaný výpočet ceny kostry, která bude uvažovat jen původní ceny hran. To proběhne v konstantním čase pro každou hranu přidanou do kostry, celková náročnost této přidané operace tak bude $\mathcal{O}(n)$, což neovlivní asymptotickou složitost algoritmu hledání minimální kostry.

— Implementační poznámky —

Priorita hrany má dvě nezávislé složky – první složka je větší z indexů jejích krajních uzlů a druhá složka je původní cena hrany. V jednoduché implementaci jako je tato můžeme obě složky uložit do jednoho celého čísla, které vypočteme podle vztahu
Priorita hrany = (index hrany krát maximální cena hrany) plus cena hrany.
Pomocí operace celočíselného dělení a zbytku po dělení (modulo), pak z priority získáme opět index hrany a její původní cenu.V mnoha programovacích jazycích, a zejména v Javě je manipulace s jediným celým číslem technicky jednodušší než manipulace s dvojicí čísel, které v Javě vyžaduje definici specifického typu dvojice čísel atd. V kódu proto udržujeme priority hran v datové struktuře, která je původně vyhrazena pro ceny hran. Mohli bychom pro větší přehlednost ukládat priority hran do separátní struktury, ale tím bychom zbytečně udržovali informaci o cenách hran na dvou místech zároveň.

Java kód

```
1  package pal;
2
3  import java.util.*;
4  import java.io.*;
5
6  public class Main {
7
8  // -----------------------------------------------------------------
9  // Values defined in problem assignment, unused in the code:
10 // -----------------------------------------------------------------
```

```
static final int nodesLIMIT    =    20000;
static final int edgesLIMIT    =  2000000;
static final int edgeCostLIMIT =     1000;
static final int outputLIMIT   = 20000000;

// ---------------------------------------------------------------
// problem solution variables
// ---------------------------------------------------------------

static Graph graph;
static int   MSTcost;
static int   capitalID;

// For each node, register its distance to the node representing the capital.
// The array is indexed by node IDs.
static int   [] distToCapital;

static int   edgePriorityFactor; // used for effective edge cost recalcualtion

// ---------------------------------------------------------------
//     M A I N
// ---------------------------------------------------------------

public static void main(String[] args) throws IOException  {
  load();
  BFSfromCapital();
  turnEdgeCostsIntoEdgePriorities();
  int MSTvalue = PrimAlgorithmWithEdgePriorities();
  System.out.printf( "%d\n", MSTvalue );
}


// ---------------------------------------------------------------
//     L O A D
// ---------------------------------------------------------------

static void load() throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  StringTokenizer st = new StringTokenizer(br.readLine());
  int n = Integer.valueOf(st.nextToken());
  graph = new Graph( n );
  int m = Integer.valueOf(st.nextToken());
  capitalID = Integer.valueOf(st.nextToken());

  int n1, n2, cost, maxEdgeCost = 0;
  for(int i = 0; i < m; i++) {
    st   = new StringTokenizer(br.readLine());
    n1   = Integer.valueOf(st.nextToken());
    n2   = Integer.valueOf(st.nextToken());
    cost = Integer.valueOf(st.nextToken());
    graph.addedge( n1, n2, cost );
    maxEdgeCost = Math.max( maxEdgeCost, cost );
  }
  // Each edge priority is calculated as
  // edge priority =  edgeIndex * edgePriorityFactor + edgeCost
  // Thus, edgePriorityFactor must be greater than maximum edge cost.
  edgePriorityFactor = maxEdgeCost + 1;
}

// ---------------------------------------------------------------
//     S O L U T I O N
// ---------------------------------------------------------------

static void BFSfromCapital( ){

  distToCapital = new int[graph.N];
```

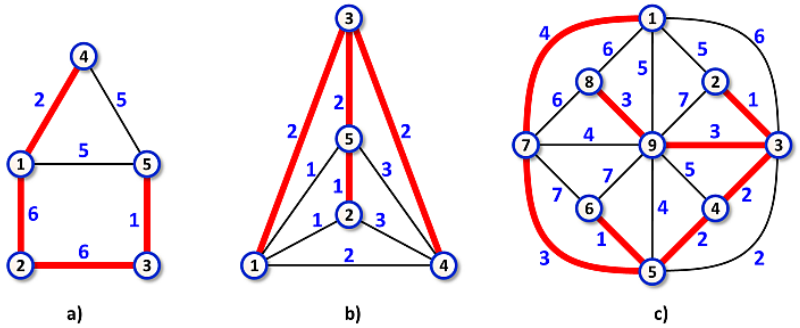
```
77    Arrays.fill( distToCapital, Integer.MAX_VALUE);
78    distToCapital[capitalID] = 0;
79
80    int [] queue = new int[graph.N];
81        queue[0] = capitalID;
82    int     head = 0;
83    int     tail = 0;
84
85    int currnode, neigh;
86    while( head <= tail ){
87      currnode = queue[head++];
88      for( int j = 0; j < graph.dg[currnode]; j++ ){
89        neigh = graph.edge[currnode][j];
90        if( distToCapital[neigh] == Integer.MAX_VALUE ){
91          distToCapital[neigh] = distToCapital[currnode]+1;
92          queue[++tail]        = neigh;
93        } // if
94      } // for
95    } // while
96  }
97
98
99  // Substitute each edge cost by priority of that edge.
100 // Store the priority in the original cost array.
101 // Each edge priority is calculated as
102 // edge priority = edgeIndex * edgePriorityFactor + edgeCost
103 static void turnEdgeCostsIntoEdgePriorities(){
104   int neigh, edgeIndex;
105   // scan all edges:
106   for( int node = 0; node < graph.N; node++ )
107     for( int j = 0; j < graph.dg[node]; j++ ){
108       neigh               = graph.edge[node][j];
109       edgeIndex           = Math.max(distToCapital[node], distToCapital[neigh]);
110       graph.cost[node][j] = edgeIndex * edgePriorityFactor + graph.cost[node][j];
111     } // for j
112 }
113
114 static int PrimAlgorithmWithEdgePriorities() {
115
116   boolean [] closed   = new boolean[graph.N];
117   long    [] nodePrio = new long[graph.N]; // node distance to the emerging MST
118   Arrays.fill( nodePrio, Integer.MAX_VALUE );
119
120   int  UNDEF               = -1;  // denotes an undefined integer value
121   int  MSTcost             = 0;
122   int  NodesInSpanningTree = 0;
123   long queueCodeFactor     = graph.N; // must be grater than maximum node ID.
124
125   // Each node in the priority queue is represented as [nodePriority, nodeId],
126   // and coded as long integer:   nodePriority * queueCodeFactor + nodeId.
127
128   PriorityQueue<Long> nodesQ = new PriorityQueue<>();
129   nodesQ.add( 0 * queueCodeFactor + capitalID );
130
131   // standard Prim's algorithm loop with its local variables:
132
133   int  currnode, neigh;
134   long currEdgePrio; // currEdge from curr to neigh
135   long currnodecode;
136
137   while( NodesInSpanningTree < graph.N ){
138
139     // pop current node from queue
140     currnodecode = nodesQ.poll();
141     currnode     = (int)( currnodecode % queueCodeFactor );
142
```

```
143     // duplicate nodes in the queue have to be dealt with properly
144     if( closed[currnode] ) continue;
145
146     // update MST size and cost
147     NodesInSpanningTree += 1;
148     MSTcost += (int)( (currnodecode / queueCodeFactor) % edgePriorityFactor );
149
150     // check all neighbours of current node and relax edges (currnode -- neigh)
151     for( int j = 0; j < graph.dg[currnode]; j++ ){
152       neigh        = graph.edge[currnode][j];
153       currEdgePrio = graph.cost[currnode][j];
154       if( !closed[neigh] && nodePrio[neigh] > currEdgePrio ) {
155           nodePrio[neigh] = currEdgePrio;
156           nodesQ.add( currEdgePrio * queueCodeFactor + neigh );
157       } // if
158     }// for
159
160     closed[currnode] = true;
161   } // while
162
163   return MSTcost;
164 }
165
166 } // end of class Main
167
168
169 // ---------------------------------------------------------------------
170 //     G R A P H   class
171 // ---------------------------------------------------------------------
172
173 public class Graph {
174   int N, M;
175
176   int  [][] edge;
177   long [][] cost;
178   int  []     dg;
179
180 public Graph (int n ){
181   this.N    = n;
182   this.edge = new int[n][2];
183   this.dg   = new int[n];
184   this.cost = new long[n][2];
185   this.E    = 0;
186 }
187
188 public void addedge(int n1, int n2, int cc) {
189   // enlarge the list of neighbours of a node, if necessary
190   if( this.dg[n1] == edge[n1].length ) {
191     this.edge[n1] = Arrays.copyOf( this.edge[n1], 2*this.dg[n1] );
192     this.cost[n1] = Arrays.copyOf(this.cost[n1], 2*this.dg[n1] );
193   }
194   if( this.dg[n2] == this.edge[n2].length ) {
195     this.edge[n2] = Arrays.copyOf( this.edge[n2], 2*this.dg[n2] );
196     this.cost[n2] = Arrays.copyOf(this.cost[n2], 2*this.dg[n2] );
197   }
198
199   // Register the edge
200   this.cost[n1][this.dg[n1]]   = cc;
201   this.cost[n2][this.dg[n2]]   = cc;
202   this.edge[n1][this.dg[n1]++] = n2;
203   this.edge[n2][this.dg[n2]++] = n1;
204   this.M++;
205 }
206
207 }
```

3 Nová síť pro pozorování gravitačních vln

Observatoř ALIGO II (Advanced Laser Interferometer Gravitational-Wave Observatory Two) prochází rozsáhlou modernizací. Je nutné vybudovat kalibrační síť propojující všechny detektory vln. Každé propojení dvojice detektorů je finančně nákladné, a proto topologie sítě nesmí obsahovat žádné cykly.
Cena každého propojení je úměrná jeho délce. Polohy všech detektorů jsou pevně dané. Náklady na propojení každé možné dvojice detektorů jsou tedy předem známé. Pokud by délka byla jediným rozhodujícím parametrem, pak by se síť měla vybudovat ve tvaru minimální kostry úplného grafu, který obsahuje všechna možná propojení a jejich délky. Jak se ale dalo očekávat, u projektu takového rozsahu je třeba zohlednit i další okolnosti. Teoretické výpočty ukázaly, že spolehlivost systému výrazně roste, pokud některá propojení v síti mají stejnou délku. Dokonce se ukázalo, že v této specifické síti klesá rušivý šum exponenciálně s počtem propojení stejné délky. Na základě této teorie rozhodla technická rada observatoře, že strategie budování sítě bude respektovat tři základní pravidla:

1. Síť bude mít stromovou topologii.
2. Síť bude obsahovat maximální možný počet propojení určité konkrétní délky, a to bez ohledu na cenu těchto propojení.
3. Cena sítě bude minimální možná.



Obrázek 5: Schémata možných propojení mezi dvojicemi detektorů. Detektory jsou znázorněny jako uzly, možná propojení jako hrany. Propojení, která mohou tvořit požadovanou síť, jsou zvýrazněna červeně. Případy a), b), c) odpovídají příkladům 1, 2 a 3 níže. V některých případech (např. b a c) může existovat více sítí, které splňují požadovaná pravidla. Všechny takové sítě ale mají stejnou celkovou délku. Upozorňujeme také, že případ a) ilustruje důležitost pravidla 3. Existují kostry obsahující hrany (1, 5) a (5, 4), které splňují pravidlo 2, ale nesplňují pravidlo 3, protože jejich celková délka je alespoň 17.

Úloha

Je dán seznam všech možných propojení mezi detektory a délka každého z nich. Určete celkovou délku sítě, která splňuje všechna tři pravidla stanovená technickou radou.

Vstup

První řádek vstupu obsahuje dvě celá čísla N a M oddělená mezerou. Hodnota N udává počet detektorů. Hodnota M udává počet možných propojení mezi dvojicemi detektorů. Následuje M řádků, každý reprezentuje jedno možné propojení. Řádek obsahuje tři celá čísla $D1$, $D2$, L oddělená mezerou, což znamená, že délka možného propojení mezi detektory $D1$ a $D2$ je rovna L. Detektory jsou očíslovány od 1 do N. Pořadí dvojic je libovolné.
Platí: $N \leq 30000$ a $M \leq 300000$. Každá délka propojení je nejvýše 10^9.

Výstup

Výstup obsahuje jeden řádek s jedním celým číslem – minimální možnou délkou laserové sítě, přičemž délka sítě je rovna součtu všech délek propojení zahrnutých v síti.

Příklad 1

Vstup

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

Výstup

```
15
```

Příklad 2

Vstup

```
5 9
1 2 1
2 4 3
4 3 2
3 1 2
1 4 2
3 5 2
1 5 1
2 5 1
4 5 3
```

Výstup

```
7
```

Příklad 3

Vstup

```
9 20
1 2 5
2 3 1
3 4 2
4 5 2
5 6 1
6 7 7
7 8 6
8 1 6
9 1 5
9 2 7
9 3 3
9 4 5
9 5 4
9 6 7
9 7 4
9 8 3
7 1 4
1 3 6
3 5 2
5 7 3
```

Výstup

```
19
```

Řešení úlohy

— Abstraktní jádro úlohy —

Abstraktní jádro úlohy je prakticky zformulováno již v zadání. Máme najít takovou kostru váženého grafu, pro kterou platí, že obsahuje maximální možný počet hran se stejnou cenou. Při zachování tohoto požadavku má být cena kostry co nejmenší.

— Rozbor a postup —

Úloha je poměrně přímočará. Pro každou skupinu hran stejné délky zjistíme cenu minimální kostry, která obsahuje maximální možný počet hran z této skupiny. Předtím si musíme uvědomit, že pro určitou skupinu hran nemusí nutně platit, že všechny její hrany lze použít v nějaké kostře. Ve skupině hran mohou existovat kružnice, jako například v prostředním obrázku v zadání, kde skupina čtyř hran s cenou 2 obsahuje kružnici délky 3 procházející uzly 1, 3, 4. Je tedy třeba pro každou skupinu hran se stejnou cenou nejprve zjistit, kolik nejvíce těchto hran lze použít v kostře. Tento počet pojmenujme použitelnost (usability) skupiny. Je zřejmé, že pro řešení úlohy připadají v úvahu jen ty skupiny hran se stejnou cenou, které mají největší použitelnost ze všech těchto skupin. Určení použitelnosti skupiny hran se stejnou cenou provedeme s pomocí Union-Find struktury.

Union-Find strukturu využijeme stejně, jako se využívá např. v Kruskalově algoritmu. Představíme si že v daném grafu máme k dispozici pouze hrany z této skupiny. Nejprve Union-Find strukturu inicializujeme uzly grafu, z nichž každý představuje iniciální samostatnou množinu, a potom postupně sjednocujeme množiny pomocí jednotlivých hran ve větší celky představující malé samostatné podstromy ve vznikající kostře. Přitom odmítneme hrany, které by ve vznikající kostře vytvořily kružnici. Když takto projdeme všechny hrany v dané skupině hran se stejnou cenou, pak počet hran použitých a neodmítnutých během této konstrukce představuje právě použitelnost (usability) této skupiny. Vypočtená použitelnost skupiny nezávisí na pořadí přidávaných hran. Bude rovna počtu hran v kostře podgrafu indukovaného právě hranami v dané skupině hran. Podgraf je také graf, a počet hran v kostře libovolného grafu je dán jednoznačně, je to počet uzlů v grafu zmenšený o počet komponent grafu.

Pro všechny skupiny hran se stejnou cenou, které obsahují alespoň 2 hrany, určíme jejich použitelnost. Potom pro každou skupinu hran se stejnou cenou a s maximální použitelností najdeme minimální kostru grafu, která obsahuje maximální možný počet hran této skupiny. To provedeme tak, že nejprve do kostry vložíme maximální možný počet hran této skupiny, analogicky jako při určování použitelnosti. Dále pak budeme pokračovat jako v libovolném algoritmu hledání minimální kostry. Protože jsme v předchozí fázi použili Union-Find strukturu, nabízí se automaticky, že ji využijeme i nadále a aplikujeme Kruskalův algoritmus na hledání minimální kostry. Jako řešení celé úlohy potom vrátíme minimální nalezenou cenu ze všech koster.

— Asymptotická složitost —

Je zřejmé, že nalezení jedné minimální kostry trvá delší dobu než určení použitelnosti skupiny hran, protože součástí hledání minimální kostry je právě také vložení do kostry všech příslušných hran této skupiny. Minimální kostru budeme muset hledat pro každou skupinu hran se stejnou cenou a s maximální použitelností. Pokud jsou tyto skupiny malé, může jich být řádově celkem až tolik, kolik je hran v grafu. Celkem tedy, protože počítáme s nejhorším možným případem, vychází asymptotická složitost řešení jako složitost jednoho hledání minimální kostry násobená počtem hran v grafu, tedy $\mathcal{O}(e \cdot e \cdot log(n))$.

— Implementační poznámky —

Předkládané řešení využívá kompletně myšlenku Kruskalova algoritmu. Minimální kostry hledá podle postupu popsaného výše, přitom však seřazení hran podel její ceny provede pouze jednou na začátku řešení celé úlohy. Tím se složitost hledání každé minimální kostry sníží na hodnotu přímo úměrnou pouze počtu hran v grafu, celé řešení má pak složitost $\mathcal{O}(e \cdot e)$. Když uvážíme, že horní mez počtu hran grafu v úloze je $3 \cdot 10^5$, vychází nám, že na velkých datech může implementace být nucena zpracovat až téměř 10^{11} hran. Pokud optimisticky předpokládáme, že řádově cca 10^8 hran lze běžně zpracovat v řádu jedné sekundy, vychází nám, že v nejhorším případě může výpočet trvat výrazně přes desítku minut. Vyhodnocení ve školním systému ale běží jen několik sekund, takže je zřejmé, že autoři do vyhodnocení vložili data data výrazně méně časově náročná.

Java kód

```
1  package pal;
2
3  import java.util.*;
4  import java.io.*;
5
6  // ----------------------------------------------------------------------
7  //    S U P P O R T    S T R U C T U R E S
8  //    EdgeRecord and EdgeGroup
9  // ----------------------------------------------------------------------
10 // Sorted edges are stored in in non-decreasing order of weights
11 // in an array of EdgeRecord type.
12 // Groups of all edges with equal weight occupy certain ranges in that array.
13 // Each group contains at least two edges.
14 // We register all those groups in a extra list of type EdgeGroup.
15 // Type EdgeGroup registers the first and the last index of the
16 // edges in the group in the sorted array,
17 // and the maximum number of edges in the group which can  be used in a MST.
18 // this maximum number is called usability of the group.
19 // Note that last index respects Java paradigm of being the first index past the range,
20 // the so called half-open range convention.
21
22 /*
23 Example Graph:
24
25             20
26          2 ----- 4
27        / |       |
28    30/   |30     |20
29    /     |       |
30  1 ----- 3 ----- 5
```

```
    30      10

Sorted array of edge record type,
each item contains IDs of edge endnodes, and edge cost.
  0        1        2        3        4        5
 +--------+--------+--------+--------+--------+--------+
 | 3 5 10 | 4 5 20 | 2 4 20 | 2 3 30 | 1 2 30 | 1 3 30 |
 +--------+--------+--------+--------+--------+--------+
           group1   group1   group2   group2   group2

In the sorted array, there are two edge groups, recorded initially in the list as :
  0        1
 +........+........+
 | 1 3 2  | 3 6 3  |
 +........+........+

However, the usability of the second group in the list is smaller than 3,
because edges in this group form a cycle in the graph. Thus, at most 2 of them
can appear in a spanning tree. In subsequent procesing of the list of the groups,
before separate spannig trees are built and analyzed,
the usability of the second group will be decreased to 2.
*/

// ---------------------------------------------------------------------
//    E D G E   R E C O R D   class
// ---------------------------------------------------------------------
class EdgeRecord implements Comparable<EdgeRecord> {
  int n1, n2, cost;

  public EdgeRecord( int a, int b, int c ) {
    n1 = a; n2 = b; cost = c;
  }
  @Override
  public int compareTo(EdgeRecord e2) {
    return (int) Math.signum( this.cost - e2.cost );
  }
} // class

// ---------------------------------------------------------------------
//    E D G E   G R O U P   class
// ---------------------------------------------------------------------
class EdgeGroup implements Comparable<EdgeGroup> {
  int start, end, usability;

  public EdgeGroup( int st, int en, int uc ) {
    start = st; end = en; usability = uc;
  }

  @Override
  public int compareTo(EdgeGroup c2) {
    return (int) Math.signum( (this.end-this.start) - (c2.end - c2.start));
  }
} // class


// ---------------------------------------------------------------------
//    M A I N  class
// ---------------------------------------------------------------------

public class Main {

// ---------------------------------------------------------------------
// Values defined in problem assignment, unused in the code:
// ---------------------------------------------------------------------
static final int nodesLIMIT    =      30000;
static final int edgesLIMIT    =     300000;
```

```
97  static final int edgeCostLIMIT =  1000000000;
98  static final int outputLIMIT   =  1073741824; // = 2^30
99
100 // ---------------------------------------------------------------------
101 // problem solution variables
102 // ---------------------------------------------------------------------
103 static Graph graph;
104 static ArrayList <EdgeGroup> edgeGroups;  // edge groups in sortedEdges in graph;
105 static UnionFind UF; // Union-Find
106
107 static int maxUsabilityOfEdgeGroup;
108 static long currTreeCost;
109 static long  minTreeCost = Long.MAX_VALUE;
110
111
112 // ---------------------------------------------------------------------
113 //    M A I N  program
114 // ---------------------------------------------------------------------
115
116 public static void main(String[] args)  throws IOException {
117   load();
118   Arrays.sort( graph.sortedEdges);
119   registerEdgeGroups();
120   UF = new UnionFind( graph.N );
121   setUsabilityOfEdgeGroups();
122   evaluateAllCandidateTrees();
123   System.out.printf( "%d\n", minTreeCost );
124 }
125
126 // ---------------------------------------------------------------------
127 //     L O A D
128 // ---------------------------------------------------------------------
129
130 static void load() throws IOException {
131   int maxEdgeCost = 0;
132   BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
133   StringTokenizer st = new StringTokenizer(br.readLine());
134
135   int n = Integer.valueOf(st.nextToken());
136   int m = Integer.valueOf(st.nextToken());
137   graph = new Graph( n, m );
138
139   int n1, n2, cost;
140   for(int i = 0; i < m; i++) {
141     st = new StringTokenizer(br.readLine());
142     n1   = Integer.valueOf(st.nextToken());
143     n2   = Integer.valueOf(st.nextToken());
144     cost = Integer.valueOf(st.nextToken());
145     // Re-index nodes so that node indexing starts from 0
146     graph.addedge( n1-1, n2-1, cost);
147   }
148 }
149
150 // ---------------------------------------------------------------------
151 //     S O L U T I O N
152 // ---------------------------------------------------------------------
153
154 static void registerEdgeGroups() {  // supposes sorted edges.
155   // Register only groups of size 2 or bigger
156   // Use technique of sliding window [istart, iend] to detect groups
157   edgeGroups = new ArrayList<>();
158   int istart = 0;
159   for( int iend = 1; iend < graph.M; iend++ )
160     if( graph.sortedEdges[iend-1].cost != graph.sortedEdges[iend].cost ) {
161       if( iend - istart >= 2 )
162          edgeGroups.add( new EdgeGroup( istart, iend,  iend-istart ) );
```

```
163       istart = iend;
164     }
165   edgeGroups.add( new EdgeGroup( istart, graph.M,  graph.M-istart ) );
166 }
167
168 static void setUsabilityOfEdgeGroups() {
169   maxUsabilityOfEdgeGroup = 0;
170   for( EdgeGroup edgeGroup : edgeGroups ) {
171     // skip groups with no perspective of being used in MST
172     if( edgeGroup.end - edgeGroup.start < maxUsabilityOfEdgeGroup )
173       continue;
174     UF.reset();  // should run more efficiently
175     edgeGroup.usability = addEdgesToTree( edgeGroup.start, edgeGroup.end, 0 );
176       maxUsabilityOfEdgeGroup =
177          Math.max( maxUsabilityOfEdgeGroup, edgeGroup.usability );
178   }
179 }
180
181 static int addEdgesToTree(int ieStart, int ieEnd, int edgesSoFar) {
182   if( ieStart >= ieEnd ) return edgesSoFar;
183   int boss1, boss2;
184
185   if( UF.numberOfSets == 1  )
186     return edgesSoFar;
187
188   for( int iEdge = ieStart; iEdge < ieEnd; iEdge++ ) {
189     boss1 = UF.find( graph.sortedEdges[iEdge].n1 );
190     boss2 = UF.find( graph.sortedEdges[iEdge].n2 );
191     if( boss1 != boss2 ) {
192       UF.union( boss1, boss2 );
193       // not needed in setUsabilityOfEdgeGroups():
194       currTreeCost += (long) graph.sortedEdges[iEdge].cost;
195       edgesSoFar   += 1;
196       if( UF.numberOfSets == 1)  break;       // tiny optimization
197     } //if
198    } //for
199    return edgesSoFar;
200 }
201
202 static void evaluateAllCandidateTrees() {
203   int edgesSoFar;
204   for(  EdgeGroup edgeGroup : edgeGroups ) {
205     if( edgeGroup.usability < maxUsabilityOfEdgeGroup ) continue;
206     currTreeCost = 0;
207     UF.reset();  // should run more efficiently
208     edgesSoFar  = addEdgesToTree( edgeGroup.start, edgeGroup.end, 0 );
209     edgesSoFar += addEdgesToTree(             0, edgeGroup.start, edgesSoFar );
210     edgesSoFar += addEdgesToTree( edgeGroup.end,         graph.M, edgesSoFar );
211     minTreeCost = Math.min(minTreeCost,  currTreeCost);
212   } //for
213 }
214
215 } // Main class
216
217
218 // ------------------------------------------------------------------
219 //     G R A P H   class
220 // ------------------------------------------------------------------
221
222 // The graph is represented by sorted list of edges, in the entire problem.
223 class Graph {
224   int N, M;
225   int MSTcost;
226   EdgeRecord [] sortedEdges;
227
228 public Graph( int n, int m ){
```

```
  this.N = n;
  this.sortedEdges = new EdgeRecord[m];
  this.M = 0;
}

public void addedge ( int n1, int n2, int cost ){
  this.sortedEdges[M] = new EdgeRecord( n1, n2, cost );
  M += 1;
}

} // Graph class


// ---------------------------------------------------------------
//     U N I O N - F I N D   class
// ---------------------------------------------------------------

public class UnionFind {

int [] boss;
int [] rank;
int numberOfSets;

// constructor
public UnionFind ( int n ) {
  boss = new int [n];
  rank = new int [n];
  reset();
}

public void reset() {
  for( int i = 0; i < boss.length; i++ ) {
    boss[i] = i;
    rank[i] = 0;
  }
  numberOfSets = boss.length;
}

public void union( int a, int b ) {
  if( rank[b] > rank[a] )
    boss[a] = b;
  else {
    boss[b] = a;
    if( rank[b] == rank[a] )
      rank[a]++;
  }
  this.numberOfSets--;
}

public int find( int a ) {
  return ( boss[a] == a ?  a  : ( boss[a] = find( boss[a] ) ) );
}

}
```