input part

```
stk = new StringTokenizer(br.readLine());
N = Integer.parseInt(stk.nextToken());
E = Integer.parseInt(stk.nextToken());
stk = new StringTokenizer(br.readLine());
for (int i = 0; i < E; i++) {
  U[i] = Integer.parseInt(stk.nextToken());
  V[i] = Integer.parseInt(stk.nextToken());
  W[i] = Integer.parseInt(stk.nextToken());
}
```

Bellman-Ford

- Basic Bellman-Ford Algorithm
  초기화하는데 N의 시간, 첫 루프에서 N(Vertext의 개수)
  다음 루프에서 E번 돌고 내부에선 상수시간이므로 theta(NE)의 time complex

```
private static void BellmanFord1() {
  Answer1[1] = 0;
  for (int i = 2; i <= N; ++i) {
    Answer1[i] = Div;
  }
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < E; ++j) {
      if (Answer1[U[j]] != Div) {
        Answer1[V[j]] = (Answer1[U[j]] + W[j] < Answer1[V[j]]) ? (Answer1[U[j]] + W[j]) : (Answer1[V[j]
      }
    }
  }
  for (int i = 1; i <= N; ++i) {
    Answer1[i] %= Div;
  }
}
```

- Advanced Bellman-Ford Algorithm
  Basic 버전의 문제점은, 매 iterate마다 모든 Edge에 대해 relaxation 가능 여부를 체크해야 한다는 것이다.
  이를 개선할 방법으로는 relaxation이 일어나는 시점을 생각하면 relaxation이 일어나는 edge에 대해 egde의 시작점이 바로 직전 iterate에서 relaxation된 경우에만 일어난다.
  따라서 relaxation된 edge들을 모아서 관리하고, 모인 edge들에 대해 다시 relaxation을 진행하는 방법을 N번 반복한다면 비효율을 줄일 수 있다.
  이렇게 하기 위해선 현재 수행하는 리스트와, 다음 단계에 수행할 리스트 두 가지를 가지고 있어야 하며, 매 반복이 끝나면 다음 단계에 수행할 리스트를 복사해야 하는 비효율이 발생한다.

하나의 리스트를 가지고 relaxation이 일어나는 node를 지속적으로 리스트 뒤에 추가해주는 방식을 사용한다면 이러한 비효율을 없앨 수 있다.
마지막으로 남는 비효율은 개념 상 같은 단계에서 하나의 노드가 두 번 이상 relaxation이 일어나는 경우 list에 중복이 생기는 것인데,
이를 위해 정렬 및 중복체크를 하는 것이 더 큰 비효율을 발생시킬 수 있어 이 점은 무시하고 구현했다.
현재 Edge들을 저장하고있는 자료 구조가 독립된 Array의 index로 iterate하므로 효율적 사용에 적합하지 않다.
하지만 이를 위해 Edge라는 static class를 만들고 Edge들을 따로 관리하는 것은 추가적인 연산이 많이 들어가므로 단순히 시간 최적화의 목적으로는 적합하지 않다.
최대 1000개의 Node 조건과 http://etl.snu.ac.kr/mod/ubboard/article.php?id=856254&bwid=1860614 에 따라 하나의 Node는 최대 1000개의 Edge를 가질 수 있으므로, segregatedList와 segregatedListCounter 배열을 Solution1 Class의 static 변수로 선언하였다.
segregatedList[i][x]는 i번째 노드가 시작점인 Edge의 U[], V[], W[] 배열에서의 index를 갖고,
segragatedListCount[i]는 앞의 2차원 배열의 x에 해당하는 index가 최대 몇 까지 갈 수 있는지에 대한 정보를 갖는다.
이와 더불어 전부 체크하던 기존 방식에서 relaxation 된 노드의 주변만 살피는 방식으로 바꾸기 위해 체크할 노드들을 갖는 LinkedList를 선언하였다.
초기 노드 거리 설정 및 선언한 segregatedList 초기화하는데 N+E의 시간이 들고, 이후 relaxation만큼의 시간이 든다.
relaxation은 만약의 경우 모든 edge 체크 마다 relaxation이 일어난다면 O(NE + N + E) = O(NE)의 시간이 들지만 실제로는 그래프의 크기가 커질수록 이럴 가능성은 매우 희박해지며
relaxation이 일어나는 경로는 그래프의 모양, 체크하는 순서에 따라 다르며 이는주어진 input값을 비교할 때 3배에서 20배 가량의 variation이 있음을 통해 알 수 있다.
따라서 구할 수 있는 time complex는 상한인 O(NE)이다.

```
static int[][] segregatedList = new int[1001][1024];
static int[] segregatedListCounter = new int[1001];

private static void BellmanFord2() {
  int i = 0, j = 0, cmpTarget = 0;
  Answer2[1] = segregatedListCounter[1] = 0;
  for (i = 2; i <= N; ++i) {
    Answer2[i] = Div;
    segregatedListCounter[i] = 0;
  }
  for (i = 0; i < E; ++i) {
    segregatedList[U[i]][segregatedListCounter[U[i]]++] = i;
  }
  LinkedList targetQueue = new LinkedList(1);
  LinkedList tail = targetQueue;
  while (targetQueue != null) {
    for (j = 0; j < segregatedListCounter[targetQueue.getTarget()]; ++j) {
      cmpTarget = segregatedList[targetQueue.getTarget()][j];
      if (Answer2[U[cmpTarget]] + W[cmpTarget] < Answer2[V[cmpTarget]]) {
        Answer2[V[cmpTarget]] = Answer2[U[cmpTarget]] + W[cmpTarget];
        LinkedList nextTarget = new LinkedList(V[cmpTarget]);
        tail.setNext(nextTarget);
        tail = nextTarget;
      }
    }
    targetQueue = targetQueue.getNext();
  }
  for (i = 1; i <= N; ++i) {
    Answer2[i] %= Div;
  }
}
```

Kruskal

Kruskal Algorithm을 통해 최대 신장 트리를 구한다.
먼저 주어진 데이터는 정렬을 하기 좋지 않기 때문에 Edge Class로 묶어서 초기화한 후 내림차순으로 정렬한다.
정렬은 HeapSort를 사용하였고, E개를 정렬하였으므로 수행시간은 O(ElogE)이다.
이후 N개의 Make-Set을 한 뒤 E번 루프를 돌면서(kruskal의 while문) *2E번의 Find-Set(with path compression)을 통해 Set 비교를 한 뒤,*
Set이 다른 경우에 대해 N-1번의 Union이 일어난다.
Set operation에 대해서는 총 N+2E+N-1번의 Make-Set, Union, Find-Set(with path compression) 중 N번이 Make-Set이다.
Rank를 이용한 Union과 경로압축을 이용한 Find-Set은 동시에 사용하기 힘드므로
(Path compression 과정에서 path compression이 일어나지 않은 부분의 rank가 훼손될 수 있고,
이를 해결하기 위해 모든 자식으로 가는 연결고리를 가지고, 모든 자식들의 rank를 확인하고 업데이트해줘야

하는데 이는 overhead가 크다, 수업시간 질문 나온 사항)
경로압축을 이용한 Find-Set만 이용하였고, 이는 실제 수행시간이 트리의 높이변화에 따라 바뀌어 구하기 힘들지만
*rank를 이용한 union과 경로압축을 이용한 Find-Set을 동시에 사용할 경우 O((2E+2N-1)log*N)으로 알려져 있다. 이를 크게 벗어나지는 않는다.
따라서 Kruskal 알고리즘을 지배하는 부분은 EdgeSort의 O(ElogE)이다.

```
static Edge[] EdgeBucket = new Edge[MAX_E];
static TreeSet[] NodeBucket = new TreeSet[MAX_N + 1];

static void DoKruskal() {
  for (int i = 0; i < E; ++i) {
    EdgeBucket[i] = new Edge(U[i], V[i], W[i]);
  }
  EdgeSort();
  for (int i = 1; i <= N; ++i) {
    NodeBucket[i] = new TreeSet();
    NodeBucket[i].Union(NodeBucket[i]);
  }
  for (int i = 0; i < E; ++i) {
    int st = EdgeBucket[i].getStart();
    int ed = EdgeBucket[i].getEnd();
    if (NodeBucket[st].findSetWithPathCompression() != NodeBucket[ed].findSetWithPathCompression()) {
      Answer += EdgeBucket[i].getWeight();
      NodeBucket[ed].findSetWithPathCompression().Union(NodeBucket[st].findSetWithPathCompression());
    }
  }
}

static void EdgeSort() {
  for (int i = E / 2; i  0; --i) {
    percolateDown(i, E);
  }
  for (int i = E; i  1; --i) {
    Edge tmp = EdgeBucket[0];
    EdgeBucket[0] = EdgeBucket[i - 1];
    EdgeBucket[i - 1] = tmp;
    percolateDown(1, i - 1);
  }
}

static void percolateDown(int i, int n) {
  int lChild = 2 * i;
  int rChild = 2 * i + 1;
  if (lChild <= n) {
    if ((rChild <= n) && (EdgeBucket[lChild - 1].getWeight()  EdgeBucket[rChild - 1].getWeight())) {
      lChild = rChild;
    }
    if (EdgeBucket[i - 1].getWeight()  EdgeBucket[lChild - 1].getWeight()) {
      Edge tmp = EdgeBucket[i - 1];
      EdgeBucket[i - 1] = EdgeBucket[lChild - 1];
      EdgeBucket[lChild - 1] = tmp;
      percolateDown(lChild, n);
    }
  }
}

static class Edge {
```

```
  int start;
  int end;
  int weight;

  public Edge(int st, int ed, int wg) {
    this.start = st;
    this.end = ed;
    this.weight = wg;
  }

  int getStart() {
    return this.start;
  }

  int getEnd() {
    return this.end;
  }

  int getWeight() {
    return this.weight;
  }

}

static class TreeSet {
  TreeSet root;

  public TreeSet() {
    this.root = null;
  }

  TreeSet findSetWithPathCompression() {
    if (this.root == this) {
      return this;
    }
    TreeSet findRoot = this.root;
    while (findRoot.findSetWithPathCompression() != findRoot) {
      findRoot = findRoot.findSetWithPathCompression();
    }
    TreeSet arrangeRoot = this.root;
    this.root = findRoot;
    while (arrangeRoot != findRoot) {
      TreeSet newArrangeRoot = arrangeRoot.root;
      arrangeRoot.root = findRoot;
      arrangeRoot = newArrangeRoot;
    }
    return findRoot;
  }

  void Union(TreeSet rootLL) {
    this.root = rootLL;
```

```
  }
}
```

Floyd-Warshall

N*N 배열을 INFINITY로 채운다(주어진 조건에서 구할 수 있는 최댓값인 200000+1), theta(N*N).
이후 E개의 edge들을 배열에 넣는다. 중복을 고려하여 기존 배열의 값과 weight의 값을 비교한다, theta(E).
(algorithm에선 theta(V*V)인 부분)
*이후 N번 N*N 배열을 순회하여 floyd-warshall algorithm을 수행한다, theta(N*N*N).
마지막으로 N*N 배열을 순회하며 Answer를 업데이트한다, theta(N*N).
dominant한 수행시간은 theta(N*N*N)이다.

```
static int[][] D = new int[MAX_N + 1][MAX_N + 1];

static void FloydWarshall() {
  for (int i = 1; i <= N; ++i) {
    for (int j = 1; j <= N; ++j) {
      if (i != j) {
        D[i][j] = INFINITY;
      }
    }
  }
  for (int i = 0; i < E; ++i) {
    D[U[i]][V[i]] = (W[i] < D[U[i]][V[i]]) ? (W[i]) : (D[U[i]][V[i]]);
  }
  for (int k = 1; k <= N; ++k) {
    for (int i = 1; i <= N; ++i) {
      for (int j = 1; j <= N; ++j) {
        D[i][j] = (D[i][j] < D[i][k] + D[k][j]) ? (D[i][j]) : (D[i][k] + D[k][j]);
      }
    }
  }

  for (int i = 1; i <= N; ++i) {
    for (int j = 1; j <= N; ++j) {
      if (i != j) {
        Answer += D[i][j];
      }
    }
  }
}
```