Code Expert :

Exams	DP Exercise	Graph Exercise	Notes (graph)
2022 Summer	Left and Right	Two Trees	1.edgeCount (4) , 2. minDistRoots (8) , 3. cycle (4) , 4. minDistCycle (8)
2022 Winter	Array Compression	Tree Augmentation	1. leaves() , 2. maxChildren() , 3. longestCycle() , 4. minMST()
2021 Summer	Pair - Subsequence	Players on a Graph	1. CycleLength() , 2. distanceToCycle(x) , 3. firstNodeInCycle(x) , 4. distanceInCycle(x,y)
2021 Winter 1	Shortest Uncommon Subsequence	Undirected Graph	1. isPath() , 2.EdgeOfTriangle , 3. NumberOfComponents , 4. LargestPerimeter
2021 Winter 2	Longest Power-of-two Subsequence	Graph Sets	1.hasCycle() , 2. hasCycleWithoutNodeZero() , 3. isSameSet(x,y) , 4. getShortestPath
2020 Summer	Longest Palindromic Subsequence	Undirected Graph	1. Two_Induced_Path , 2.Exists_Euler_Cycle , 3. Two_Colorable , 4. Max_Distance(v)
2020 Winter	Shuffle	Binary Tree	1. min() , 2.depth(k) , 3.children_of_node(k) , 4. Keyofrank(r)

Semester Exercises

		NOTES
Programming Assignment 1	Longest 3-Segmented Subarray	
	Maximum Subset	
Programming Assignment 2	BST	Binary Search Tree
	Ticket Shop	Dp
Programming Assignment 3	Edit Distance to Subsequence	Dp
	Navigation	Graph dfs
Programming Assignment 4	Lava	
	Minimum Cost Edges	Graph Dijkstra
Programming Assignment 5	Railway	Graph Kruskal
	Paths with Small Weights	Graph Floyd - Warshall
New exercises released	Def solve the dp ones !	

Newly Released Exercises

		NOTES
2022		
Programming Assignment 3	Levels	Dp
	Split and Merge	Dp
Programming Assignment 4	BST	Binary Search Tree
	In Between	DFS or BFS
Programming Assignment 5	Grid Search	Graph Dijkstra
	Internet Connectivity	Graph Kruskal
2021		
Programming Assignment 3	Longest Alternating Subsequence	
	Palindromic Edit Distance	Dp
Programming Assignment 4	Height in a Binary Tree	BST
	Coins	DFS
Programming Assignment 5	Shortest Cycle	Graph
	Eulerian Distance	Graph
2020		
Programming Assignment 5	Basic Properties Of A Graph	Graph Problem with subtasks

Ticket Shop

You want to go to a place that is distance D away. There is an array A of n tickets that are availble, where $A[i]$ is the distance that ticket i can cover. You also have k vouchers that allow you to double the distance of **any** k of the tickets that you buy (you can also feel free to not use all of them). Your goal is to calculate the minimum amount of tickets that you need to buy to reach **EXACTLY** distance D (you can assume that this is always possible).

For example, for $D = 20$ and $A = [2, 4, 2, 3, 1, 3, 3, 4, 3]$. With $k = 0$, you can reach 20 with 6 tickets $[3, 3, 3, 3, 4, 4]$. With $k = 1$, you can reach 20 with 5 tickets $[3 \cdot 2, 3, 3, 4, 4]$. With $k = 2$, you can reach 20 with 4 tickets $[3 \cdot 2, 3 \cdot 2, 4, 4]$. With $k = 3$, you can reach 20 with 3 tickets $[3 \cdot 2, 3 \cdot 2, 4 \cdot 2]$.

You need to implement your solution as a method **minTicket(**D, k, n, A**)** in the file Main.java. You get one point for each passing test set. To pass both test sets, your solution is expected to run in time $O(n \cdot D \cdot k)$.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

distance D A [ ]

A[i]: dist. ticket i can cover (n tickets)

k : x2 the dist of ticket's

min # tickets to reach EXACTLY D

Semester Exercise

$O(n \cdot D \cdot k)$

IDEA: DP[i][j][k]: min # tickets to reach j with A 0...i , using k vouchers

DP[i][j][k] = min{ DP[i-1][j][k] , DP[i-1][j-A[i]][k] , DP[i-1][j-2·A[i]][k-1] }

DP[i][0][k] = 0

DP[i-1][j][k]: don't use A[i] ticket

DP[i-1][j-A[i]][k]: use A[i]

DP[i-1][j-2·A[i]][k-1]: use A[i] with k double

```
public static int minTicket(int D, int k, int n, int[] A) {
  // TODO
  int [][][] dp = new int [n+1][D+1] [k+1];
  for(int i = 0 ; i<=n ; i++) {
    for(int j = 0 ; j<=D; j++ ) {
      for(int x = 0 ; x<=k ; x++) {
        dp [i][j][x] = 99999999;
        if(j==0) {
         dp [i][j][x] = 0 ;
        }
      }
    }
  }



  for(int i = 1 ; i<=n ; i++ ){
    for(int x = 0 ; x<=k ; x++) {
      for(int distance = 0 ; distance<=D; distance++ ) {
        if(distance- 2*A[i-1] >=0 && x-1 >=0) dp[i][distance][x]  = min3(dp[i-1][distance][x] , dp[i-1][distance- A[i-1] ][x] + 1 , dp[i-1][distance-2*A[i-1] ][x-1] + 1 ) ;

        else if(distance-A[i-1] >=0 ) dp[i][distance][x] = Math.min(dp[i-1][distance][x] ,dp[i-1][distance- A[i-1] ][x] + 1 ) ;

        else dp[i][distance][x] = dp[i-1][distance][x] ;
      }
    }
  }

  return dp[n][D][k] ;
}

public static int min3(int a, int b, int c) {
  return Math.min ( Math.min(a,b) , c) ;
}
```

Math.min will be used afterwards

DP[i][0][x] = 0 You can reach 0 distance with 0 tickets

Edit Distance to Subsequence

You are given an array of n integers A and an array of m integers B such that $n \leq m$. What is the minimum number of elements of B that you need to replace by other integers so that A is a subsequence of B? (not necessarily on consecutive indices)

For example, for $A = [1, 5, 2, 3]$ and $B = [1, 4, 2, 5, 6, 2, 5]$, the answer is 1, and it is obtained by replacing the last element of B by 3. Also, for $A = [1, 5, 2, 3]$ and $B = [5, 8, 8, 2, 8, 8, 3]$, the answer is 2, and it is obtained by replacing the first two elements of B by 1 and 5.

You need to implement your solution as a method **editToSubsequence(n, m, A, B)** in the file Main.java. You get one point for each passing test set. To pass both test sets correctly, your solution needs to run in $O(nm)$ time.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

Semester Exercise

A: n — B: m — O(nm)

DP[0...m] x [0...n]

IDEA: DP[i][j] = the min # elements of B' 0...i needed to replace s.t. A' 0...j is a subseq. of B!

$$DP[i][j] = \begin{cases} \text{if } B[i] = A[j] \\ \quad = DP[i-1][j-1] \\ B[i] \neq A[j] \\ \quad = \min\{ DP[i-1][j],\ DP[i-1][j-1] + 1 \end{cases}$$

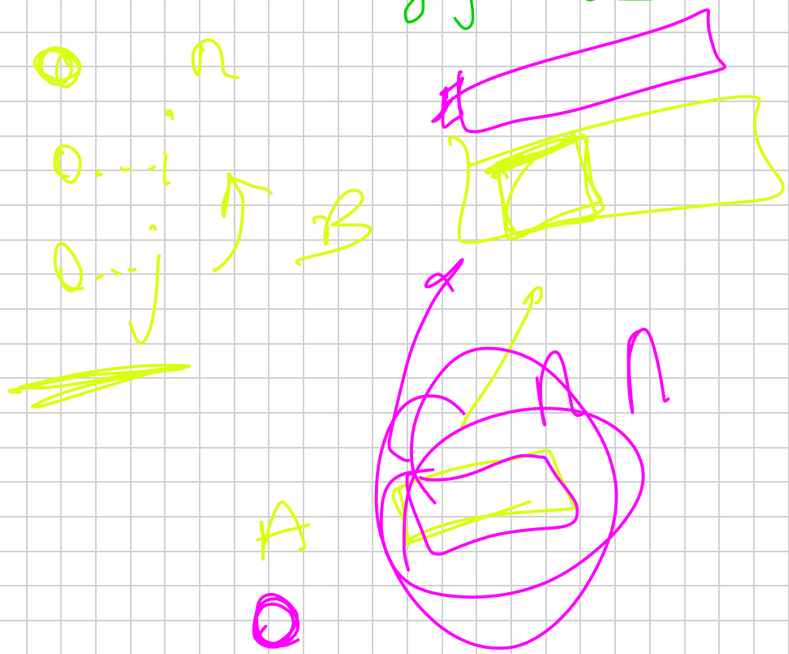
DP[i-1][j]: how did we achieve A 0...j without using B[i] changing

DP[i-1][j-1] + 1: how did we achieve A 0...j-1 + 1 for changing B[i]

```java
public static int editToSubsequence(int n, int m, int[] A, int[] B) {
 // TODO: solve
 int [][]dp = new int [m+1][n+1] ;
 for(int i = 0 ; i<n ; i++ ) {
   dp[0][i] = n ;
 }

 for(int i = 1 ; i<=m ; i++) {
   for(int j = 1 ; j<=n ; j++) {
     if(B[i-1] == A[j-1]) dp[i][j] = dp[i-1][j-1] ;
     else dp[i][j] = Math.min(dp[i-1][j-1] + 1 , dp[i-1][j] ) ;
   }
 }
 return dp[m][n];
```

DP[0][i] = n

we cannot use any element of B, change n with "ADD"ing n elements of A.



Split and Merge

2022
Semester Exercise

You are given an array of n integers A. You can apply the following operation to the array: choose two neighboring values $A[i]$ and $A[i+1]$, and replace them by a single value $\max(A[i], A[i+1]) + \text{floor}(\min(A[i], A[i+1])/2)$. In other words: half of the smaller value is absorbed by the larger value.

calc

You apply $n-1$ such operations until the array has a single value. You can choose the order of the operations. Find the maximum value that you can obtain at the end.

As an example, consider $A = \{9\ 2\ 4\ 8\ 10\}$, the maximum value is 21. A possible sequence of steps is:
$(9\ 2)\ 4\ 8\ 10 \rightarrow 10\ 4\ (8\ 10) \rightarrow 10\ (4\ 14) \rightarrow (10\ 16) \rightarrow 21$.

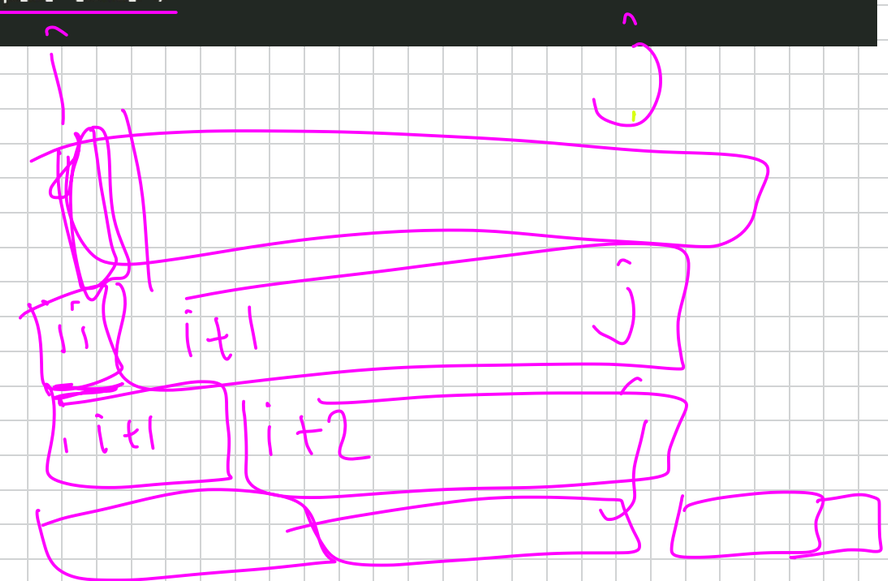
You need to implement your solution as a method **splitMerge(n, A)** in the file Main.java. You get one point for each passing test. To pass both test sets, your solution needs to run in time $O(n^3)$.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

```java
public static int calc(int a , int b ) {
  return Math.max(a, b) + (int)Math.floor(Math.min(a, b) / 2);
}

public static int splitMerge(int n, int[] A) {
  // TODO: implement
  int [][] dp = new int [n][n] ;
  for(int i = 0 ; i<n ; i++ ){
    for(int j = 0 ; j<n ; j++ ){
      dp[i][j] = -1 ;
    }
  }

  for(int i = 0; i < n; i++) {
    dp[i][i] = A[i];
  }

  for(int len = 2 ; len<= n ; len++) {
    for(int i = 0 ; i + len <=n ; i++ ) {
      int j = i +len-1 ;

      for(int k = i ; k<j ; k++) {
        dp[i][j] = Math.max(dp[i][j] , calc(dp[i][k] , dp[k+1][j] ) );
      }
    }
  }
  return dp[0] [n-1] ;
}
```

length

(index j calculation)



Levels

You are given two arrays of n positive integers A and B which satisfy $A[i] > B[i]$ for all $i = 0, \ldots, n-1$. These arrays describe a game with n levels, numbered from 0 to $n-1$, which goes as follows. At the beginning of the game, you have an infinite amount of energy. In order to complete level i, your energy must be at least $A[i]$, and after completing level i your energy becomes $B[i]$.

You can choose to skip some of the levels, but the levels that you complete must be completed in the given order. Return the maximum number of levels that you can complete. That is, return the length of the longest subsequence of levels (not necessarily consecutive) that you can complete.

For example, for $n = 5$, $A = (5, 6, 3, 3, 2)$, and $B = (4, 5, 1, 2, 1)$, the answer is 3, with one such subsequence being formed of the levels 0, 3, and 4.

You need to implement your solution as a method **getLevels(**n**,** A**,** B**)** in the file Main.java. You get one point for each passing test. To pass both test sets, your solution needs to run in time $O(n \log n)$.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

2022
Semester Exercise

```java
public static int getLevels(int n, int[] A, int[] B) {
  // TODO: implement
  //memo array for MAX REMAINNING ENERGY AFTER COMPLETING I LEVELS
  int [] memo = new int [n+1] ;
  for(int i = 0 ; i<=n ; i++) {
    memo[i] =-1 ;
  }
  memo[1] = B[0] ;
  int ans = 1 ;

  for(int i = 1 ; i<n ; i++  ) {
    int reqEnergy = A[i-1] ;
    int maxlevel = i ;
    while(maxlevel>=0) {
      if(memo[maxlevel] >= reqEnergy) {
        break ;
      }
      maxlevel -- ;
    }
    if(memo[maxlevel+1] == -1) {
        ans ++ ;
        memo[maxlevel+1] = B[i-1] ;
    }
    else {
        memo[maxlevel+1] = Math.max(memo[maxlevel+1] , B[i-1] );
    }

  }

  return ans;
}
```

```java
public static int getLevels(int n, int[] A, int[] B) {
  // IDEA: Keep track of DP[j] = maximum energy that you can have after
  // completing j levels.
  // Then, the longest subsequence ending in (A[i], B[i]) has lenght DP[j] + 1,
  // where j is the largest number such that DP[j] >= A[i].
  // We update DP by letting DP[j + 1] = max(DP[j + 1], B[i]). Note that it can
  // happen that DP[j + 1] is already larger than B[i] (that is why we need the
  // maximum), which cannot happen in the standard longest ascending subsequence
  // problem.

  int DP[] = new int[n + 1];
  int DPsize;

  DP[1] = B[0];
  DPsize = 1;
  for (int i = 1; i < n; i++) {
    // Find largest j such that DP[j] >= A[i].
    int lf = 0, rg = DPsize;
    while (lf != rg) {
      int mid = (lf + rg + 1) / 2;
      if (DP[mid] >= A[i]) {
        lf = mid;
      } else {
        rg = mid - 1;
      }
    }
    if (lf == DPsize) {
      DP[lf + 1] = B[i];
      DPsize += 1;
    } else {
      DP[lf + 1] = Math.max(DP[lf + 1], B[i]);
    }
  }

  return DPsize;
}
```

Left and Right

You are given an array A of n integers, indexed from 0 to $n-1$.

You play the following game. You start with a score of 0. At each step of the game, you can make one of the following moves:

1. If A contains at least two elements, you can remove the **leftmost** and the **rightmost** element of A and add to your score the absolute value of their difference. For example, if the leftmost and the rightmost elements had values x and y, you add $|x-y|$ to your score.
2. You can remove the **leftmost** element of A with no change to your score.
3. You can remove the **rightmost** element of A with no change to your score.

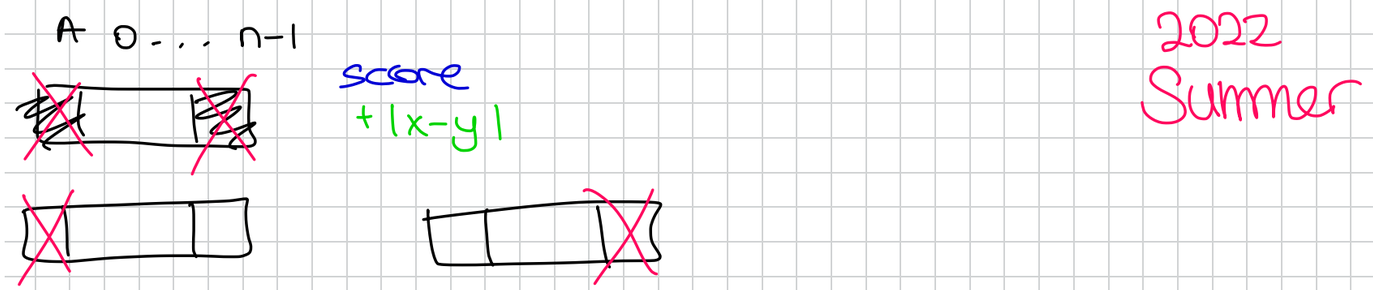
Your task is to find the maximum score that you can obtain in the game. You need to implement your solution as a method **getMaximumScore(**n, A**)**.

Hint: Use dynamic programming with $D[i][j]$ representing the maximum score that you can obtain on $A[i], \ldots, A[j]$.

Grading (16 points):

- An $O(n^2)$ implementation gets 16 points and an $O(n^3)$ implementation gets 6 points.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.




```java
public static int getMaximumScore(int n, int[] A) {
  // TODO: your code
  int [] [] dp = new int [n] [n] ;

  for(int i = n-1 ; i>=0 ; i--) {
    for(int j = 0 ; j<n ; j++) {
      if(i<j)  {
        if(i+1<n && j-1>=0) dp[i][j] = max3(dp[i+1][j-1] +  Math.abs(A[i]-A[j]) , dp[i+1][j] , dp[i][j-1]) ;
        else if(i+1<n && j-1<0) dp[i][j] = Math.max(Math.abs(A[i]-A[j]) , dp[i+1][j] ) ;
        else if(i+1>=n && j-1>=0) dp[i][j] = Math.max(Math.abs(A[i]-A[j]) , dp[i][j-1] ) ;
        else dp[i][j] = Math.abs(A[i]-A[j]) ;
      }
    }
  }

  return dp[0][n-1];
}

public static int max3 (int a, int b , int c ) {
  return Math.max(Math.max(a,b) , c) ;
}
```

Array Compression

You are given an array of n integers $A[0], ..., A[n-1]$.

You can perform the following compression operation to the array: for some $i < j$ such that $A[i] = A[j]$, you can remove from the array all the elements $A[i], A[i+1], ..., A[j-1], A[j]$. After that the array is re-indexed and you can perform other compression operations.

You really love compression operations. Return the maximum number of compression operations that you can perform to the array.

For example, for $n = 6$ and the array $A = [4, 5, 6, 4, 6, 5]$, the answer is 2: you can compress $[6, 4, 6]$, after which the array becomes $[4, 5, 5]$, and then you can compress $[5, 5]$.

You need to implement your solution as a method **maxCompressions(n, A)**.

Hint: Use dynamic programming with $D[i][j]$ representing the maximum number of compression operations that you can perform to the array $A[i], A[i+1], ..., A[j-1], A[j]$.

Grading (16 points): An $O(n^3)$ implementation gets 16 points. An implementation that runs in time less than 0.1 seconds for $n \leq 10$ gets at least 6 points.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

```
public static int maxCompressions(int n, int[] A) {
  // TODO: implement
  int [] [] dp = new int [n][n] ;

  for(int i = n-1 ; i>=0 ; i--) {
    for(int j = 0 ; j<n ; j++) {
      if(i<j) {
        if(A[i]==A[j]) dp[i][j] = 1 + dp[i+1][j-1] ;

        for(int k = i; k<j ; k++) {
          dp[i][j] = Math.max(dp[i][j] , dp[i][k] + dp[k+1][j]) ;
        }

      }
    }
  }

  return dp[0][n-1];
}
```

2022 Winter

IDEA:

DP[i][j]: max # compressions operations one can perform on A[i] ... A[j]

DP[i][j]: (if A[i]=A[j] then 1+DP[i+1][j-1])

dp(i,j) max { dp(i,j) , dp(i,k) + dp(k+1,j) }

checking all compressions opportunities inbetween

dp i j

decreasing i
increasing j

+1 compression and everything inbetween

checking i...k k+1...j

every possib. of division

k=i ... <j

i i i+1 j

i i+1 i+2 j

i i+2 i+3 j

i k k j

k=i ...<j

Palindromic Edit Distance

2021 Semester Exercise

Given a sequence A of n characters, your task is to compute the minimum number of operations that is required to turn A into a *palindrome*. A ***palindrome*** is a sequence of characters which reads the same backward as forward, e.g., "level", "noon","racecar". We consider the following operations:

- Change the character at any position
- Remove the character at any a position
- Insert a character at any position.

For example, if A is "ETHZETHZ", the answer is 3.

Please see the Main.java file. The task is to implement the method "Palindromic_Edit_Distance". You are free to create auxiliary methods. You do not need to change other methods.

You get one point for each passing test set. To pass both test sets correctly, your solution has to be in $O(n^2)$ time.

Attention: The index of an array in the code template starts at **0** following the convention of Java.

dp[i][j] := min # operations i ... j

dp[i][j] = min { dp[i][j-1] + 1 , dp[i+1][j] + 1 } remove

dp[i+1][j-1] + 1 → change

+0 → if(A[i] == A[j]

```java
public static int Palindromic_Edit_Distance(char []A, int n)
{
    int [][]dp=new int[n][n];

    for(int i = 0 ; i<n ; i++) {
       for(int j = 0 ; j<n ; j++) {
         dp[i][j] = 999999;
       }
    }
    for(int i = 0 ; i<n ; i++) {
       dp[i][i] = 0 ;
       if(i+1 <n && A[i] == A[i+1]) dp[i][i+1] = 0;
       if(i+1 <n && A[i] != A[i+1]) dp[i][i+1] = 1 ;
    }

    for(int j = 2 ; j <n ; j++) {
       for(int i = 0 ; i+j <n ; i++) {

         if(i+j - 1 >=0) dp[i][i+j] = Math.min (dp[i][i+j] , dp[i][i+j-1] + 1 ) ;
         if(i+1 < n) dp[i][i+j] = Math.min (dp[i][i+j] , dp[i+1][i+j] + 1 ) ;
         if(i+j - 1 >=0 && i+1 < n ) {
           if(A[i+j] == A[i] ) dp[i][i+j] = Math.min (dp[i][i+j] , dp[i+1][i+j-1]  ) ;
           else dp[i][i+j] = Math.min (dp[i][i+j] , dp[i+1][i+j-1] + 1 ) ;
         }
       }
    }

    return dp[0][n-1];

}
```

init for len 1 len 2

len

Pair-Subsequence

You are given an array A of n integers, some of them positive and some of them negative, indexed from 0 to $n-1$.

Your task is to find the maximum sum that the elements of a *pair-subsequence* of A can have. A pair-subsequence is defined as a subsequence that also satisfies the following properties:

- if the element with index i appears in the subsequence, then either the element with index $i-1$ or the element with index $i+1$ also appears in the subsequence
- no three elements with consecutive indices appear in the subsequence

More informally, a pair-subsequence is a subsequence obtained by selecting pairs of consecutive elements of A, such that the pairs are not adjacent to each other. A pair-subsequence is also allowed to be empty, case in which the sum of its elements is defined to be 0.

For example, for $n = 8$ and $A = [1, 2, 3, 3, 2, 2, -10, 10]$, the pair-subsequence of maximum sum is the one with indices $1, 2, 4, 5$, corresponding to values $2, 3, 2, 2$, and hence with sum 9.

Return the maximum sum of the elements of a pair-subsequence of A.

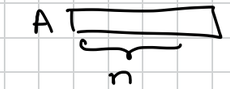
You need to implement your solution as a method **getMaximumSum(n, A)**.

Grading (16 points):

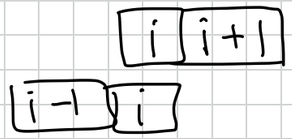
- An $O(n^2)$ solution gets 6 points and an $O(n)$ solution gets 16 points.

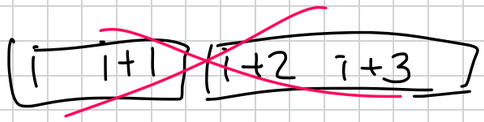
Attention:

- You are **NOT** allowed to use imports explicitly or implicitly, other than the imports already included in the code template.
- When you submit the code, you see your score as a **percentage**, not as a number of points.



0 ... n-1
pos and neg





i i+1 ~~i+2~~ i+3 i+4 ✓

2021 Summer

IDEA:

DP[i]: max sum of the elements of a pair subseq of A' 0...i

$$DP[i] = \max\{DP[i-1],\ DP[i-3] + A[i-1] + A[i]\}$$

```java
public static int getMaximumSum(int n, int[] A) {
  // TODO: implement
  int[] dp = new int [n] ;

  for(int i = 1 ; i<n ; i++ ){
    if(i-3 >= 0 ) {
      dp[i] = Math.max ( dp[i-3] + A[i-1] + A[i] , dp[i-1]) ;
    }
    else dp[i] = Math.max ( A[i-1] + A[i] , dp[i-1]  ) ;
  }

  return dp[n-1];
}
```

Shortest Uncommon Subsequence

Given an n-element string A and an m-element string B such that A is not a subsequence of B and $n \leq m$, your task is to compute the length of a ***shortest uncommon subsequence*** of A with respect to B, i.e., the length of a shortest subsequence of A that is not a subsequence of B. For example, if $A = EETTT$ and $B = TETTE$, EET is a shortest uncommon subsequence of A with respect to B, and its length is 3.

Grading (16 points):

- An $O(nm^2)$-time implementation gets 16 points, while an $O(2^n \cdot m)$-time implementation still gets 6 points.

HINT:

- You may use an $(n+1) * (m+1)$-size DP table where, e.g., $\mathrm{DP}[i][j]$ stores the answer for the first i characters of A and the first j characters of B.
- For a dynamic program, you can decompose the problem based on the largest position in B that matches the last character of A.

Note:

- The input array A is indexed from 0, and all characters are capital letters.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You see the **percentage** of your submission in the CodeExpert system instead of points.

IDEA (n+1) x (m+1)

DP[i][j] : the length of a shortest subseq of A 0...i that is not a subseq of B 0...j.

compute: largestpos of A[i] in B (-1 if no such pos.)

$$DP[i][j] := \begin{cases} A[i] \notin B[0..j] & DP[i][j] = 1 \\ A[i] \in B[0..j] & \text{largestpos} \\ DP[i][j] = \min \begin{cases} DP[i-1][j] \\ DP[i-1][\text{largestpos}] + 1 \end{cases} & \end{cases}$$

dp[i][0] = 1

2021 Winter 1

we use A[i] with 1

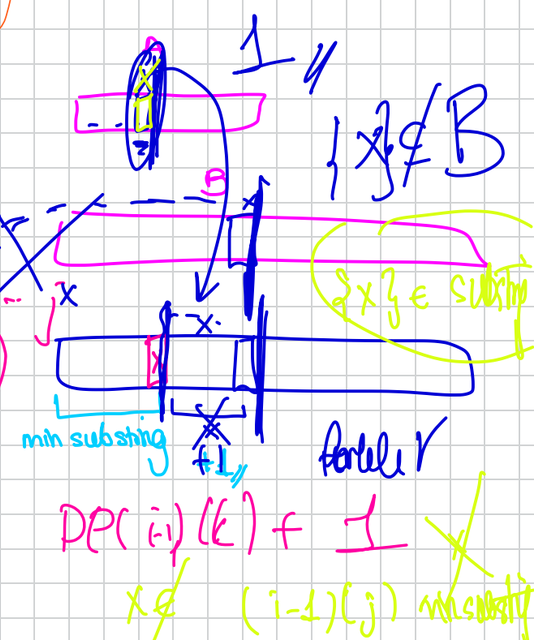
-> we don't consider A[i]

-> we consider A[i] till largest pos

if we have to add A[i] to achieve shorter than i+1

```
public static int ShortestUncommonSubsequence(int n, int m, char[] A, char B[]) {
  int[][] dp = new int[n+1][m+1] ;

  for (int i = 0; i <= n; i++) {
    dp[i][0] = 1;
  }
  for (int j = 0; j <= m; j++) {
    dp[0][j] =9999999;
  }
  // TODO: implement
  for(int i = 1 ; i<=n ; i++ ) {
    for(int j = 1 ; j<=m ; j++) {
      int largestpos = -1 ;
      for(int x = j-1 ; x>=0 ; x--) {
        if(B[x] == A[i-1]) {
          largestpos = x ;
          break ;
        }
      }
      if(largestpos==-1) dp[i][j] = 1 ;
      else {
        dp[i][j] = Math.min (dp[i-1][j]  , dp[i-1][largestpos] + 1 ) ;
      }
    }
  }

  return dp[n][m];
}
```

```
public static int ShortestUncommonSubsequence(int n, int m, char[
  int[][] dp = new int[n+1][m+1];
  // TODO: implement

  for(int i = 0 ; i<=n ; i++ ){
    for(int j = 0 ; j<=m ; j++) {
      dp[i][j] = 9999999;
    }
  }
  for(int i = 0 ; i<=n ; i++ ){
      dp[i][0] = 1;
  }
  dp[0][0] = 9999999;
  for(int i = 1 ; i<=n ; i++) {
    for(int j = 1 ; j<=m ; j++) {
      int maxpos = i ;
      int index;
      for(index= j-1 ; index>=0 ; index --) {
        if(B[index] == A[i-1]) break ;
      }
      if(index == -1) dp[i][j] = 1 ;
      else {
        dp[i][j] = Math.min(dp[i-1][j] , dp[i-1][index] + 1) ;
      }
    }
  }


  return dp[n][m];
}
```



Longest Power-of-Two Subsequence

You are given an array of n integers between (and including) 0 and $2^{20} - 1 = 1048575$. Your task is to compute the length of the longest subsequence such that the absolute difference between consecutive elements is a power of two (that is, one of 1, 2, 4, 8, and so on). Note that the elements of the subsequence do not need to be on consecutive positions in the original array.

For example, for the input array 1, 6, 3, 7, 3, 9, the answer is 4. This is because there exists a valid subsequence of length 4: this is $1, 3, 7, 3$. On the other hand, there exists no valid subsequence of length 5.

You need to implement your solution as a method **getLongestSubsequence(n, A)**, where A is the given array.

Grading (16 points):

- Let h be the maximum value of an integer in the input. So, h is at most $2^{20} - 1 = 1048575$. An $O(h + n \log h)$-time implementation gets 16 points and an $O(n^2)$-time implementation gets 6 points.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You see the **percentage** of your submission in the CodeExpert system instead of points.

```
public static int getLongestSubsequence(int n, int[] A) {
  // TODO: implement
  int pow2 = 1 ;
  for(int i = 0 ; i<20 ; i++ ) {
    pow2 *= 2 ;
  }

  int h = 0 ;
  for(int i = 0 ; i<n ; i++) {
    h = Math.max(h , A[i]) ;
  }

  int [] lastseenpos = new int[h+1] ;
  for(int i = 0 ; i<h ; i++ ){
    lastseenpos[i] = -1 ;
  }
  int [] dp = new int[n] ;

  for (int i = 0 ; i<n ; i++ )  {
    dp[i] = 1 ;
    for( int last= 1 ; last< pow2 ; last*=2 ) {
      if(A[i] + last <= h && lastseenpos[A[i] + last] != -1 ) {
        dp[i] = Math.max (dp[i], dp[lastseenpos[A[i] + last]] + 1 ) ;
      }
      if(A[i] - last >= 0 && lastseenpos[A[i] - last] != -1 ) {
        dp[i] = Math.max (dp[i] , dp[lastseenpos[A[i] - last]] + 1 ) ;
      }
    }

    lastseenpos[A[i]] = i ;
  }

  int max = 0 ;
  for(int i = 0 ; i<n ; i++) {
    max = Math.max(max, dp[i] ) ;
  }
  return max;
}
```

) computing 2^{20}

) max A[i] = h

) marking last seen ops in array

A — n integers — $0 \le A[i] \le 2^{20} - 1$

2021 Winter 2

compute h (max val of integer in input)

lastseen pos [ ] 0 ... h — init -1

as we're looping, we mark last seen pos

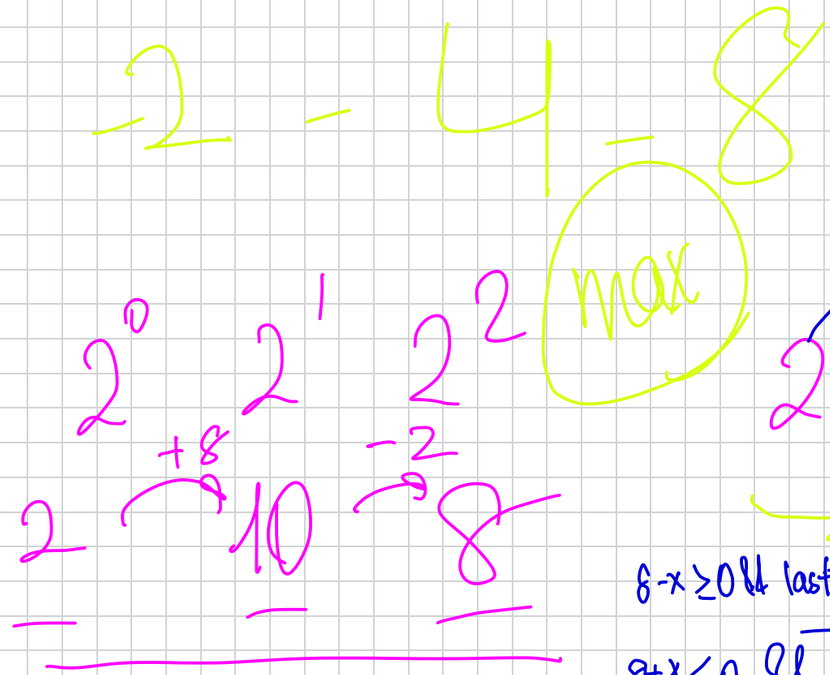
dp 0 ... n

dp[i]: longest power of 2 subseq. that ends in i — for A[i]

dp[i]: max{ 1, dp[lastseenpos[A[i] - pow2]] + 1 , dp[lastseenpos[A[i] + pow2]] + 1 }

pow2 : 1 2 ... 2^{20}

max dp value is the answer



Longest Palindromic Subsequence

A ***palindrome*** is a sequence of characters which reads the same backward as forward, e.g., "level", "noon","racecar". Given a sequence A of n characters, your task is to compute the ***length*** of the **longest palindromic subsequence** of A, i.e., the *length* of the *longest subsequence* of A that is a *palindrome*. For instance, if A is "ETZHEEHU", "HEEH" is the longest palindromic subsequence of A, and its length is **4**.

Grading (16 points):

- An $O(n^2)$-time implementation gets 16 points, while an $O(2^n)$-time implementation still gets 6 points.

Note: The input array A is indexed from 0, and all characters are capitals.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You see the **percentage** of your submission in the CodeExpert system instead of points.

2020 Summer

dp[i][j] := length of longest palin. subseq from A[i] ... A[j].

$$dp[i][j] := \begin{cases} \text{if } A[i]=A[j] \\ \max\{dp(i+1)(j),\ dp(i)(j-1),\ dp(i+1)(j-1)+2\} \\ \text{else} \\ \max\{dp(i+1)(j),\ dp(i)(j-1),\ dp(i+1)(j-1)\} \end{cases}$$

(i+1 ... j) (i ... j-1) (i+1 ... j-1 + 2)

dp[i][i] = 1 for all i

```
public static int Palindrome(char[] A, int n) {
  int [][] dp=new int[n][n];
  for(int i = 0 ; i<n ; i++ ){
    dp[i][i] = 1 ;
  }

  for(int i = n-1 ; i>=0 ; i--) {
    for(int j = 0 ; j<n ; j++) {
      if(i< j) {
        if(A[i]==A[j]) {
          if(i+1<n && j-1>=0) dp[i][j] = max3(dp[i+1][j] , dp[i][j-1] , dp[i+1][j-1] + 2 ) ;
          else if(i+1<n && j-1<0) dp[i][j] = Math.max(dp[i+1][j] , 2  ) ;
          else if(i+1>=n && j-1>=0)dp[i][j] = Math.max(dp[i][j-1] , 2  ) ;
        }
        else {
          if(i+1<n && j-1>=0) dp[i][j] = max3(dp[i+1][j] , dp[i][j-1] , dp[i+1][j-1] ) ;
          else if(i+1<n && j-1<0) dp[i][j] =dp[i+1][j] ;
          else if(i+1>=n && j-1>=0)dp[i][j] = dp[i][j-1] ;
        }
      }
    }
  }

  return dp[0][n-1] ;
}

public static int max3(int a , int b , int c ) {
  return Math.max(Math.max(a,b) , c) ;
}
```

Shuffle

Given three strings A, B and C with $|A| = n$, $|B| = m$ and $|C| = n + m$, your task is to decide if C is a ***shuffle*** of A and B. A *shuffle* of A and B is formed by merging A and B into a new string while maintaining both the internal order of the characters of A and the internal order of the characters of B. For example, C = P*INE*AP*P*LE is a shuffle of A = **PAPLE** and B = *INEP*.

Grading (16 points):

- An $O(n \cdot m)$-time or $O((n+m)^2)$-time implementation gets 16 points, while an $O(2^{n+m})$-time implementation still gets 5 points.

Note: The input arrays A, B and C are indexed from 1 instead of 0 in the code.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You only can see the **percentage** of your submission in the CodeExpert system instead of points.

IDEA :

2020 Winter

DP[i][j] : A 0...i is a shuffle for C 0... i+j
B 0...j

DP[0][0] true

DP[i][j] := DP[i-1][j] ∧ A[i]==C[i+j] we use A[i]
OR
DP[i][j-1] ∧ B[j]==C[i+j] we use B[j]

```java
public static boolean Shuffle(int n, int m, char[] A, char[] B, char[] C)
{
  //The input arrays A, B and C are indexed from 1 instead of 0 in the code.
  boolean[][] dp = new boolean[n+1][m+1];
  dp[0][0] = true;

  for(int i = 0 ; i<n+1 ; i++ ){
    for(int j = 0  ; j<m+1 ; j++) {
      if(i-1>=0) dp[i][j] = dp[i][j] || (dp[i-1][j] && A[i]==C[i+j] ) ;
      if(j-1>=0)  dp[i][j] = dp[i][j] ||(dp[i][j-1] && B[j]==C[i+j] ) ;
    }
  }


  return dp[n][m];
}
```

GRAPH

Two Trees

You are given two rooted trees, A and B, with disjoint vertex sets. Tree A has vertices indexed by a_0, ..., a_{n-1}, with the root at index a_0, and tree B has vertices indexed by b_0, ..., b_{n-1}, with the root at index b_0. The edges in each tree are weighted by positive integers.

You want to add some new edges that connect leaves of A with leaves of B, thus creating a connected graph. Specifically, you can add an edge only if it goes between a leaf of A and a leaf of B. Any edge that you add has weight 0.

The distance between two vertices is defined as minimum total weight of a path that connects the two vertices.

Given these two trees, you have to implement the following methods. All the answers are guaranteed to fit on an "int" type.

1. **edgeCount():** Return the total number of edges that you can add between the two trees.
2. **minDistRoots():** Return the minimum distance between the roots of the two trees that you can achieve by adding exactly one edge.
3. **cycle():** Return 1 if you can add exactly two edges such that the resulting graph has a simple cycle (no repeated vertices) that contains the two roots.
4. **minDistCycle():** Return the minimum length of a simple cycle (no repeated vertices) that contains the two roots that you can achieve by adding exactly two edges. You can assume such a simple cycle exists.

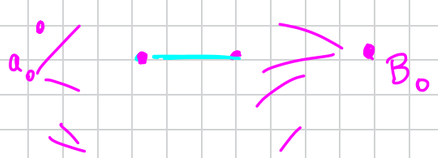
Grading (24 points):

1. **edgeCount()** (4 points): An $O(n)$-time implementation gets 4 points.
2. **minDistRoots()** (8 points): An $O(n)$-time implementation gets 8 points and an $O(n^2)$-time implementation gets 4 points.
3. **cycle()** (4 points): An $O(n)$-time implementation gets 4 points.
4. **minDistCycle()** (8 points): An $O(n)$-time implementation gets 8 points and an $O(n^2)$-time implementation gets 4 points.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

Tree A $a_0 \dots a_{n-1}$

Tree B $b_0 \dots b_{n-1}$

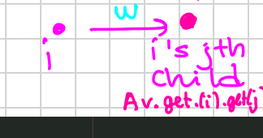


Layout:

```
class Main {
  public static void main(String[] args) {
    // Uncomment the following two lines if you want to read from a file
    // In.open("public/example.in");
    // Out.compareTo("public/example.out");

    int t = In.readInt(); // number of tests
    for (int test = 0; test < t; test++) {
      int n = In.readInt(); // number of vertices
      int query = In.readInt(); // method to test

      // We consutrct Av, Aw, Bv, and Bw such that:
      //   Av.get(i).get(j) = the j-th child of vertex i in A
      //   Aw.get(i).get(j) = the weight of the edge from vertex i in A to its j-th child
      //   (same for Bv, Bw)
      // The number of elements of Av.get(i) is given by Av.get(i).size() (same for Aw, Bv, Bw).
      ArrayList<ArrayList<Integer>> Av = new ArrayList<ArrayList<Integer>>(n);
      ArrayList<ArrayList<Integer>> Aw = new ArrayList<ArrayList<Integer>>(n);
      ArrayList<ArrayList<Integer>> Bv = new ArrayList<ArrayList<Integer>>(n);
      ArrayList<ArrayList<Integer>> Bw = new ArrayList<ArrayList<Integer>>(n);
      for (int i = 0; i < n; i++) { // initialization with empty arrays
        Av.add(new ArrayList<Integer>());
        Aw.add(new ArrayList<Integer>());
        Bv.add(new ArrayList<Integer>());
        Bw.add(new ArrayList<Integer>());
      }

      int u, v, w;
      for (int i = 0; i < n - 1; i++) { // read edges of A
        u = In.readInt(); // parent is u
        v = In.readInt(); // child is v
        w = In.readInt(); // weight is w
        Av.get(u).add(v);
        Aw.get(u).add(w);
      }
      for (int i = 0; i < n - 1; i++) { // read edges of B
        u = In.readInt(); // parent is u
        v = In.readInt(); // child is v
        w = In.readInt(); // weight is w
        Bv.get(u).add(v);
        Bw.get(u).add(w);
      }

      if (query == 1) {
        Out.println(edgeCount(n, Av, Aw, Bv, Bw));
      } else if (query == 2) {
        Out.println(minDistRoots(n, Av, Aw, Bv, Bw));
      } else if (query == 3) {
        Out.println(cycle(n, Av, Aw, Bv, Bw));
      } else {
        Out.println(minDistCycle(n, Av, Aw, Bv, Bw));
      }
    }

    // Uncomment this line if you want to read from a file
    // In.close();
  }
```

2022 Summer

Av.get(i).get(j)
Aw.get(i).get(j)




```
// For all methods, we have Av, Aw, Bv, and Bw such that:
//   Av.get(i).get(j) = the j-th child of vertex i in A
//   Aw.get(i).get(j) = the weight of the edge from vertex i in A to its j-th child
//   (same for Bv, Bw)
// The number of elements of Av.get(i) is given by Av.get(i).size() (same for Aw, Bv, Bw).
// The root of each tree is the vertex with index 0.

public static int edgeCount(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                            ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  // TODO: your code
  return 0;
}

public static int minDistRoots(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                               ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  // TODO: your code
  return 0;
}

public static int cycle(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                        ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  // TODO: your code
  return 0;
}

public static int minDistCycle(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                               ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  // TODO: your code
  return 0;
}
```

```
public static int edgeCount(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                            ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  // TODO: your code
  int leavesA = 0 ;
  int leavesB = 0 ;
  for(int i = 0; i<n ; i++) {
    if(Av.get(i).size() == 0 ) leavesA  ++ ;
    if(Bv.get(i).size() == 0 ) leavesB  ++ ;
  }

  return leavesA * leavesB ;
}
```

```
public static int cycle(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                        ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  // TODO: your code
  int leavesA = 0 ;
  int leavesB = 0 ;
  for(int i = 0; i<n ; i++) {
    if(Av.get(i).size() == 0 ) leavesA  ++ ;
    if(Bv.get(i).size() == 0 ) leavesB  ++ ;
  }
  if(leavesA >=2 && leavesB >=2) return 1 ;
  return 0;
}
```

```
private static void weightDFS(int x, int[] D, ArrayList<ArrayList<Integer>> v, ArrayList<ArrayList<Integer>> w) {
  for (int i = 0; i < v.get(x).size(); i++) {
    weightDFS(v.get(x).get(i), D, v, w);
    if (D[x] == 0) {
      D[x] = D[v.get(x).get(i)] + w.get(x).get(i);
    } else {
      D[x] = Math.min(D[x], D[v.get(x).get(i)] + w.get(x).get(i));
    }
  }
}
```

with weight DFS

```
public static int minDistRoots(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                               ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  int[] Apath = new int[n];
  int[] Bpath = new int[n];
  weightDFS(0, Apath, Av, Aw);
  weightDFS(0, Bpath, Bv, Bw);
  return Apath[0] + Bpath[0];
}
```

```
public static int minDistCycle(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                               ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  // TODO: your code
  int[] Apath = new int[n];
  int[] Bpath = new int[n];
  weightDFS(0, Apath, Av, Aw);
  weightDFS(0, Bpath, Bv, Bw);

  int minA1 = Integer.MAX_VALUE, minA2 = Integer.MAX_VALUE, minB1 = Integer.MAX_VALUE, minB2 = Integer.MAX_VALUE;

  for (int i = 0; i < Av.get(0).size(); i++) {
    if (Apath[Av.get(0).get(i)] + Aw.get(0).get(i) < minA1) {
      minA2 = minA1;
      minA1 = Apath[Av.get(0).get(i)] + Aw.get(0).get(i);
    } else if (Apath[Av.get(0).get(i)] + Aw.get(0).get(i) < minA2) {
      minA2 = Apath[Av.get(0).get(i)] + Aw.get(0).get(i);
    }
  }
  for (int i = 0; i < Bv.get(0).size(); i++) {
    if (Bpath[Bv.get(0).get(i)] + Bw.get(0).get(i) < minB1) {
      minB2 = minB1;
      minB1 = Bpath[Bv.get(0).get(i)] + Bw.get(0).get(i);
    } else if (Bpath[Bv.get(0).get(i)] + Bw.get(0).get(i) < minB2) {
      minB2 = Bpath[Bv.get(0).get(i)] + Bw.get(0).get(i);
    }
  }
  return minA1 + minA2 + minB1 + minB2;

}
```

with BFS (mine)

```
public static int minDistRoots(int n, ArrayList<ArrayList<Integer>> Av, ArrayList<ArrayList<Integer>> Aw,
                               ArrayList<ArrayList<Integer>> Bv, ArrayList<ArrayList<Integer>> Bw) {
  int[] Apath = new int[n];
  int[] Bpath = new int[n];

 LinkedList<Integer> a = new LinkedList<> () ;
 a.addFirst(0) ;
 while(! a.isEmpty()) {
   int curr = a.removeFirst() ;
   for(int j = 0; j < Av.get(curr).size() ; j++) {
     if(Apath[Av.get(curr).get(j)] == 0 ) {
       Apath[Av.get(curr).get(j)] = Apath[curr] + Aw.get(curr).get(j) ;
        a.addLast(Av.get(curr).get(j)) ;
     }
   }
 }

 LinkedList<Integer> b = new LinkedList<> () ;
 b.addFirst(0) ;
 while(! b.isEmpty()) {
   int curr = b.removeFirst() ;
   for(int j = 0; j < Bv.get(curr).size() ; j++) {
     if(Bpath[Bv.get(curr).get(j)] == 0 ) {
       Bpath[Bv.get(curr).get(j)] = Bpath[curr] + Bw.get(curr).get(j) ;
       b.addLast(Bv.get(curr).get(j)) ;
     }
   }
 }

 int mina = 9999999; int minb = 9999999;
 for(int i = 0; i<n ; i++) {
   if(Av.get(i).size() == 0 ) mina = Math.min( mina , Apath[i]) ;
   if(Bv.get(i).size() == 0 ) minb = Math.min( minb , Bpath[i]) ;
 }
 return mina + minb ;

}
```

Tree Augmentation

You are given a tree with n vertices, numbered from 0 to $n-1$, in which vertex 0 is the root of the tree. The tree is given as an array of $n-1$ tuples $(U[i], V[i], C[i])$, which means there is an edge between parent $U[i]$ and child $V[i]$ of cost $C[i]$, where $C[i]$ is a positive integer. You are also given another array of $n-1$ positive integers $D[1], \ldots, D[n-1]$, which we will explain in a moment.

You want to augment this tree by adding **one** edge to it, from the root (vertex 0) to one of the other vertices. If you choose to add the edge from the root to vertex i, the cost to traverse it is $D[i]$.

You want to understand the structure of the tree and the effect of adding an edge to it. You have to implement the following methods:

1. **leaves()**: Return the number of leaves (that is, vertices with no children) in the tree.
2. **maxChildren()**: Return the maximum number of children of a vertex in the tree.
3. **longestCycle()**: Return the length of the longest cycle that you can create by adding one edge between the root and one of the other vertices. The length of a cycle is the sum of the costs of the edges in it. (A cycle may also consist of only two vertices, if you add an edge between the root and one of its children; in that case the length is equal to the cost of the original edge plus the cost of the added edge.)
4. **minMST()**: Return the minimum cost of a minimum spanning tree that can be obtained by adding one edge between the root and one of the other vertices. (Note that, if you do not add any edge, the minimum spanning tree is the tree itself, and it has a cost equal to the sum of the costs of the edges in the tree. By adding one more edge you may make the cost of a minimum spanning tree smaller.) This question is independent of the the **longestCycle()** question (the edge that you add does not need to lead to a longest cycle).

Hint: Try to understand under what conditions the cost of the minimum spanning tree decreases after adding one edge to the tree.

Grading (24 points):

1. **leaves()** (4 points): An $O(n)$-time implementation gets 4 points.
2. **maxChildren()** (4 points): An $O(n)$-time implementation gets 4 points.
3. **longestCycle()** (8 points): An $O(n)$-time implementation gets 8 points, and an $O(n^2)$-time implementation gets 4 points.
4. **minMST()** (8 points): An $O(n)$-time implementation gets 8 points, and an $O(n^2 \log n)$-time implementation gets 4 points.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

0 . . . n-1

n vertices tree

vertex 0 : root

n-1 tuples

U(i) V(i) C(i)

C(i)

D(i)

2022 Winter

layout:

```java
import java.util.Arrays;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
// ADDITIONAL IMPORTS ARE NOT ALLOWED

class Main {
  public static void main(String[] args) {
    // Uncomment the following two lines if you want to read from a file
    In.open("public/5-min_mst_small.in");
    Out.compareTo("public/5-min_mst_small.out");

    int t = In.readInt(); // number of tests
    for (int test = 0; test < t; test++) {
      int n = In.readInt(); // number of vertices
      int query = In.readInt(); // method to test

      int[] U = new int[n];
      int[] V = new int[n];
      int[] C = new int[n];
      int[] D = new int[n];

      for (int i = 0; i < n - 1; i++) { // read edges
        U[i] = In.readInt(); // edge between parent U[i]
        V[i] = In.readInt(); // and child V[i]
        C[i] = In.readInt(); // of cost C[i]
      }
      for (int i = 1; i < n; i++) { // read D[1], ..., D[n - 1]
        D[i] = In.readInt();
      }

      if (query == 1) {
        Out.println(leaves(n, U, V, C, D));
      } else if (query == 2) {
        Out.println(maxChildren(n, U, V, C, D));
      } else if (query == 3) {
        Out.println(longestCycle(n, U, V, C, D));
      } else {
        Out.println(minMST(n, U, V, C, D));
      }
    }

    // Uncomment this line if you want to read from a file
    // In.close();
  }
```

my sol:

```java
public static int leaves(int n, int[] U, int[] V, int[] C, int[] D) {
  // TODO: implement
  boolean [] isParent = new boolean [n] ;
  for(int i = 0 ; i<n ; i++) {
    isParent[U[i]] = true ;
  }

  int countLeaves = 0 ;
  for(int i = 0 ; i<n ; i++) {
    if(isParent[i] == false)countLeaves++ ;
  }
  return countLeaves;
}

public static int maxChildren(int n, int[] U, int[] V, int[] C, int[] D) {
  // TODO: implement
  int [] numChildren = new int [n] ;
  for(int i = 0 ; i<n ; i++) {
    numChildren[U[i]] ++ ;
  }

  int max = 0 ;
  for(int i = 0 ; i<n ; i++) {
    max = Math.max( max , numChildren[i]) ;
  }
  return max;
}
```

```java
public static int longestCycle(int n, int[] U, int[] V, int[] C, int[] D) {
  // For your convenience, we consutrct G and H such that:
  //   G.get(i).get(j) = the j-th child of vertex i
  //   H.get(i).get(j) = cost of the edge between vertex i and the j-th child of vertex i
  // The number of elements of G.get(i) is given by G.get(i).size().
  ArrayList<ArrayList<Integer>> G = new ArrayList<ArrayList<Integer>>(n);
  ArrayList<ArrayList<Integer>> H = new ArrayList<ArrayList<Integer>>(n);
  for (int i = 0; i < n; i++) { // initialization with empty arrays
    G.add(new ArrayList<Integer>());
    H.add(new ArrayList<Integer>());
  }
  for (int i = 0; i < n - 1; i++) { // adding information from (U, V, C) to (G, H)
    G.get(U[i]).add(V[i]);
    H.get(U[i]).add(C[i]);
  }

  LinkedList<Integer> Q = new LinkedList<> () ;
  Q.addFirst(0) ;
  int[] distance = new int [n] ;

  while(!Q.isEmpty()) {
    int i = Q.removeFirst() ;
    for(int j = 0 ; j<G.get(i).size() ; j++) {
      if(distance[G.get(i).get(j) ] == 0 ) {
        distance[G.get(i).get(j) ] = distance[i ] + H.get(i).get(j) ;
        Q.addLast(G.get(i).get(j)) ;
      }
    }
  }

  int max = 0 ;
  for(int i = 1 ; i<n ; i++) {
    max = Math.max(distance[i] + D[i]  , max) ;
  }
  return max;
}
```

Master

```java
public static int longestCycle(int n, int[] U, int[] V, int[] C, int[] D) {
  // For your convenience, we consutrct G and H such that:
  //   G.get(i).get(j) = the j-th child of vertex i
  //   H.get(i).get(j) = cost of the edge between vertex i and the j-th child of vertex i
  // The number of elements of G.get(i) is given by G.get(i).size().
  ArrayList<ArrayList<Integer>> G = new ArrayList<ArrayList<Integer>>(n);
  ArrayList<ArrayList<Integer>> H = new ArrayList<ArrayList<Integer>>(n);
  for (int i = 0; i < n; i++) { // initialization with empty arrays
    G.add(new ArrayList<Integer>());
    H.add(new ArrayList<Integer>());
  }
  for (int i = 0; i < n - 1; i++) { // adding information from (U, V, C) to (G, H)
    G.get(U[i]).add(V[i]);
    H.get(U[i]).add(C[i]);
  }

  int[] DP = new int[n];
  sumTo(0, DP, G, H);
  int result = 0;
  for (int i = 1; i < n; i++) {
    result = Math.max(result, DP[i] + D[i]);
  }
  return result;
}

public static int minMST(int n, int[] U, int[] V, int[] C, int[] D) {
  // For your convenience, we consutrct G and H such that:
  //   G.get(i).get(j) = the j-th child of vertex i
  //   H.get(i).get(j) = cost of the edge between vertex i and the j-th child of vertex i
  // The number of elements of G.get(i) is given by G.get(i).size().
  ArrayList<ArrayList<Integer>> G = new ArrayList<ArrayList<Integer>>(n);
  ArrayList<ArrayList<Integer>> H = new ArrayList<ArrayList<Integer>>(n);
  for (int i = 0; i < n; i++) { // initialization with empty arrays
    G.add(new ArrayList<Integer>());
    H.add(new ArrayList<Integer>());
  }
  for (int i = 0; i < n - 1; i++) { // adding information from (U, V, C) to (G, H)
    G.get(U[i]).add(V[i]);
    H.get(U[i]).add(C[i]);
  }

  int[] DP = new int[n];
  maxTo(0, DP, G, H);
  int sumEdges = 0, result = 0;
  for (int i = 0; i < n - 1; i++) {
    sumEdges += C[i];
  }
  result = sumEdges;
  for (int i = 1; i < n; i++) {
    result = Math.min(result, sumEdges - DP[i] + D[i]);
  }
  return result;
}
```

```java
private static void sumTo(int x, int[] D, ArrayList<ArrayList<Integer>> G, ArrayList<ArrayList<Integer>> H) {
  for (int i = 0; i < G.get(x).size(); i++) {
    D[G.get(x).get(i)] = D[x] + H.get(x).get(i);
    sumTo(G.get(x).get(i), D, G, H);
  }
}

private static void maxTo(int x, int[] D, ArrayList<ArrayList<Integer>> G, ArrayList<ArrayList<Integer>> H) {
  for (int i = 0; i < G.get(x).size(); i++) {
    D[G.get(x).get(i)] = Math.max(D[x], H.get(x).get(i));
    maxTo(G.get(x).get(i), D, G, H);
  }
}
```

```java
public static int minMST(int n, int[] U, int[] V, int[] C, int[] D) {
  // For your convenience, we consutrct G and H such that:
  //   G.get(i).get(j) = the j-th child of vertex i
  //   H.get(i).get(j) = cost of the edge between vertex i and the j-th child of vertex i
  // The number of elements of G.get(i) is given by G.get(i).size().
  ArrayList<ArrayList<Integer>> G = new ArrayList<ArrayList<Integer>>(n);
  ArrayList<ArrayList<Integer>> H = new ArrayList<ArrayList<Integer>>(n);
  for (int i = 0; i < n; i++) { // initialization with empty arrays
    G.add(new ArrayList<Integer>());
    H.add(new ArrayList<Integer>());
  }
  for (int i = 0; i < n - 1; i++) { // adding information from (U, V, C) to (G, H)
    G.get(U[i]).add(V[i]);
    H.get(U[i]).add(C[i]);
  }


  LinkedList<Integer> Q = new LinkedList<> () ;
  Q.addFirst(0) ;
  int[] distance = new int [n] ;
  int[] maxEdgeOnPath = new int[n] ;

  while(!Q.isEmpty()) {
    int i = Q.removeFirst() ;
    for(int j = 0 ; j<G.get(i).size() ; j++) {
      if(distance[G.get(i).get(j) ] == 0 ) {
        distance[G.get(i).get(j) ] = distance[i] + H.get(i).get(j) ;

        maxEdgeOnPath[G.get(i).get(j)] = Math.max(H.get(i).get(j) , maxEdgeOnPath[i] ) ;

        Q.addLast(G.get(i).get(j)) ;
      }
    }
  }

  int maxdiff = 0 ;
  for(int i = 1 ; i<n ; i++) {
    maxdiff = Math.max( maxEdgeOnPath[i] - D[i]  , maxdiff) ;
  }

  int sum = 0 ;
  for(int i = 0 ; i<n ; i ++) {
    sum+= C[i] ;
  }

  if(maxdiff == 0) return sum ;
  return sum-maxdiff;
}
```

An Undirected, Unweighted Graph

During the semester, you have learned basic graph theory and many graph algorithms. This assignment considers an **undirected**, **unweighted** graph $G = (V, E)$ with adjacency lists. You need to implement the following tasks:

1. **IsPath()**: Check if G is a ***path*** provided that G is connected, i.e., G contains a path connecting all the vertices in V, but G does not conain any other edge.
2. **EdgeOfTriangle(u, v)**: For a given edge $e = (u, v)$, check if G contains a ***triangle*** that includes e, i.e., V contains a vertex w such that E contains all the three edges (u, w), (u, v) and (v, w).
3. **NumberOfComponents()**: Calculate the number of connected components in G.
4. **LargestPerimeter(v)**: Given a vertex v, calculate the ***largest perimeter*** from v in G provided that G is connected, i.e., the largest number of vertices equidistant from v in G.

Grading (24 points):

1. **IsPath()** (4 points): An $O(|V| + |E|)$-time implementation gets 4 points.
2. **EdgeOfTriangle(u, v)** (6 points): An $O(|V|)$-time implementation gets 6 points, while an $O(|V|^2)$-time implementation only gets 3 points.
3. **NumberOfComponents()** (6 points): An $O(|V| + |E|)$-time implementation gets 6 points.
4. **LargestPerimeter(v)** (8 points): An $O(|E|)$-time implementation gets 8 points, while an $O(|V| \cdot |E|)$-time implementation only gets 4 points.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You see the **percentage** of your submission in the CodeExpert system instead of points.

```java
class Graph{
  private int n;              // number of vertices
  private int m;              // number of edges
  private int[] degrees;      // degrees[i]: the degree of vertex i
  private int[][] edges;      // edges[i][j]: the endpoint of the j-th edge of vertex i
  private boolean[] visited;  // visited[i]: whether vertex i has been visited
```

```java
public boolean IsPath() {
  // TODO: implement
  int count = 0 ;
  for(int i = 0 ; i<n ; i++) {
    if(degrees[i] >2) return false;
    if(degrees[i] == 1) count ++ ;
  }
  if(count>2 ) return false;
  return true;
}
```

```java
public boolean EdgeOfTriangle(int u, int v) {
  // TODO: implement
  int[]edgesofu = new int[n] ;
  for(int i = 0 ; i<degrees[u] ; i++) {
    if(edges[u][i] != v) edgesofu[edges[u][i]] = 1 ;
  }

  for(int i = 0 ; i<degrees[v] ; i++) {
    if(edgesofu[edges[v][i]] == 1 ) return true ;
  }
  return false;
}
```

```java
public int NumberOfComponents() {
  // TODO: implement
  int count =0 ;
  for(int i = 0 ; i<n ; i++) {
    if(visited[i] == false) {
      count ++ ;
      DFS(i) ;
    }
  }
  // Note that we provide a method DFS(x), which you may find useful.
  // You are also free to modify the DFS(x) method.
  return count;
}
```

```java
private void DFS(int v) {
  visited[v]=true;
  for (int i = 0; i < degrees[v]; i++) {
    if(visited[edges[v][i]] == false) {
      DFS(edges[v][i]);
    }
  }
}
```

```java
public int LargestPerimeter(int v) {
  // TODO: implement

  // If you want to use a queue, the following code declares one:
  for(int i = 0 ; i<n ; i++) {
    visited[i] = false ;
  }
   LinkedList<Integer> Q = new LinkedList<Integer>();
   Q.addFirst(v) ;
   visited[v] = true ;
   int[] distance = new int[n] ;

   while(!Q.isEmpty()) {
     int curr = Q.removeFirst() ;

     for(int i = 0 ; i<degrees[curr] ; i++ ) {
       if(visited[edges[curr][i]] == false) {

         visited[edges[curr][i]] = true ;
         distance[edges[curr][i]] = distance[curr] + 1;
         Q.addLast(edges[curr][i]) ;

       }
     }
   }

   int [] distancenum = new int[n] ;
   for(int i = 0 ; i<n; i++) {
     distancenum[ distance[i] ] ++ ;
   }

   int max = 0;
   for(int i = 0 ; i<n ; i++) {
     max = Math.max(max , distancenum[i]) ;
   }
  return max;
}
```

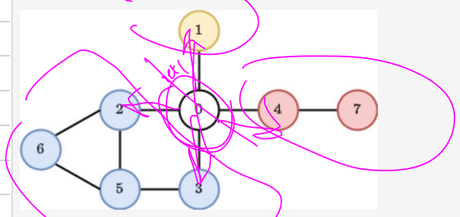
2021 Winter 1

Graph Sets

You are given an undirected connected graph with n nodes numbered from 0 to $n-1$ and m edges between them.

The nodes of the graph are partitioned into sets in the following way. Node 0 forms a set of its own. Suppose that node 0 is removed from the graph. Then, the nodes in each connected subgraph of the resulting graph forms another set.

An example is shown below, in which the nodes are colored according to the set they are in. Specifically, the sets in this example are $\{0\}$, $\{1\}$, $\{2,3,5,6\}$, and $\{4,7\}$.



Given such a graph, you have to answer queries of the following type:

1. **hasCycle()**: Return 1 if the graph has a cycle, and 0 otherwise.
2. **hasCycleWithoutNodeZero()**: Suppose that node 0 is removed from the graph. Return 1 if the resulting graph has a cycle, and 0 otherwise.
3. **isSameSet(x, y)**: Given two nodes x and y, return 1 if they are in the same set, and 0 otherwise.
4. **getShortestPath(x, y)**: Given two nodes x and y **that are in different sets**, return the shortest-path distance between them.

You have to implement the four methods described above. In addition, we provide an **intialize()** method which we guarantee to run before any of the queries. Feel free to use this method, for example, to initialize information that you want available in all of the queries (of course, the time consumed by **intialize()** counts toward the time limit of the problem).

For an easier implementation of the queries, **recall and make use of the fact** that the graph is connected.

Grading (24 points):

1. **hasCycle()** (4 points): This query will be run at most once. An $O(n+m)$-time implementation gets 4 points.
2. **hasCycleWithoutNodeZero()** (4 points): This query will be run at most once. An $O(n+m)$-time implementation gets 4 points.
3. **isSameSet(x, y)** (8 points): If q is the number of queries of this type, an $O(n+m+q)$-time implementation gets 8 points and an $O(q(n+m))$-time implementation gets 4 points.
4. **getShortestPath(x, y)** (8 points): If q is the number of queries of this type, an $O(n+m+q)$-time implementation gets 8 points and an $O(q(n+m))$-time implementation gets 4 points.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You see the **percentage** of your submission in the CodeExpert system instead of points.

```
class Graph {
  private int n;              // number of nodes
  private int m;              // number of edges
  private int[] degree;       // degrees[i]: the degree of vertex i
  private int[][] edges;      // edges[i][j]: the endpoint of the j-th edge of vertex i
  private boolean[] visited;  // visited[i]: whether node i has been visited

  public int [] set ;
  public int setnum ;
  public int [] distancetozero ;
  public boolean bhascycle ;

  Graph(int n, int m, int[][] edgeArray){
    this.n = n;
    this.m = m;
    degree = new int[n];
    edges = new int[n][];
    visited = new boolean[n];
    this.set = new int [n] ;
    this.distancetozero = new int [n] ;
    this.setnum = 0 ;
    this.bhascycle = false;

    for (int i = 0; i < m; i++) {
      degree[edgeArray[i][0]]++;
      degree[edgeArray[i][1]]++;
    }
    for (int i = 0; i < n; i++) {
      edges[i] = new int[degree[i]];
      degree[i] = 0;
    }
    for (int i = 0; i < m; i++) {
      edges[edgeArray[i][0]][degree[edgeArray[i][0]]++] = edgeArray[i][1];
      edges[edgeArray[i][1]][degree[edgeArray[i][1]]++] = edgeArray[i][0];
    }
  }
```

added

```
  public void initialize() {
    // TODO: implement
    for(int i = 0 ; i<n ;i++) {
      set[i] = -1 ;
    }
    set[0] = 0 ;

    for(int i = 0 ; i<degree[0] ; i++ ) {
      if(visited[edges[0][i]] == false) {
        setnum ++;
        set[edges[0][i]] = setnum ;
        dfsmodified(edges[0][i], set) ;
      }
    }

    for(int i = 0 ; i<n ; i++) {
      visited[i] = false;
    }

    LinkedList<Integer> Q = new LinkedList<Integer>();
    Q.addFirst(0) ;
    distancetozero[0] = 0 ;
    while(! Q.isEmpty()) {
      int curr = Q.removeFirst() ;
      for(int i = 0 ; i<degree[curr]; i++) {
        if(visited[edges[curr][i]] == false) {
          visited[edges[curr][i]] = true ;
          distancetozero[edges[curr][i] ] = distancetozero[curr] + 1 ;
          Q.addLast(edges[curr][i]) ;
        }

      }

    }

    // Note that we provide a method DFS(x), which you may find useful.
    // You are also free to modify the DFS(x) method.

    // If you want to use a queue, the following code declares one:
    // Queue<Integer> Q = new LinkedList<Integer>();
  }

  public void dfsmodified(int v , int [] set) {

    visited[v] = true ;

    for(int i = 0 ; i<degree[v] ; i++) {
      int curr = edges [v][i] ;
      if(visited[curr] == false && curr != 0 ) {
        set[curr] = set[v] ;
        dfsmodified(curr , set) ;
      }
    }

  }
```

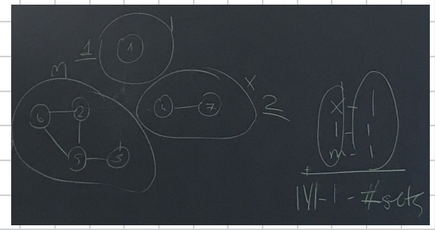
finding sets

find distance to zero

```
  public int hasCycle() {
    // TODO: implement
    if(m>=n) return 1 ;
    else return 0 ;
  }

  public int hasCycleWithoutNodeZero() {
    // TODO: implement
    //number of sets :
    if(m - degree[0] >=  n - setnum ) return 1 ;

    return 0;
  }

  public int isSameSet(int x, int y) {
    // TODO: implement
    if(set[x] == set[y]) return 1 ;
    return 0 ;
  }

  public int getShortestPath(int x, int y) {
    // TODO: implement
    return distancetozero[x] + distancetozero[y];
  }
```

m=n-1 tree has
m > n-1 : cycle

2021 Winter 2



provided DFS (not used)

```
  private void DFS(int x) {
    visited[x] = true;
    for (int i = 0; i < degree[x]; i++) {
      if (!visited[edges[x][i]]) {
        DFS(edges[x][i]);
      }
    }
  }
}
```

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;


class Main {
  public static void main(String[] args) {
    // Uncomment the following two lines if you want to read from a file.
    // Also make sure that the lines are commented when testing with the
    // "Test" button or when submitting. Otherwise your code may exceed the
    // time liimt.
    // In.open("public/1-cycle.in");
     //Out.compareTo("public/1-cycle.out");

    int tests = In.readInt(); // number of tests
    for (int t = 0; t < tests; t++) {
      int n = In.readInt(); // number of nodes
      int m = In.readInt(); // number of edges
      int q = In.readInt(); // number of queries

      int[][] edgeArray = new int[m][2]; // array of edges
      for (int i = 0; i < m; i++) {
        edgeArray[i][0] = In.readInt();
        edgeArray[i][1] = In.readInt();
      }

      Graph G = new Graph(n, m, edgeArray); // graph
      G.initialize();

      // queries
      for (int i = 0; i < q; i++) {
        int type = In.readInt();
        if (type == 1) {
          // hasCycle
          Out.println(G.hasCycle());
        } else if (type == 2) {
          // hasCycleWithoutNodeZero
          Out.println(G.hasCycleWithoutNodeZero());
        } else if (type == 3) {
          // isSameSet
          int x = In.readInt();
          int y = In.readInt();
          Out.println(G.isSameSet(x, y));
        } else if (type == 4) {
          // getShortestPath
          int x = In.readInt();
          int y = In.readInt();
          Out.println(G.getShortestPath(x, y));
        }
      }
    }
    // Uncomment this line if you want to read from a file
    // In.close();
  }
}
```

2020 Summer

An Undirected, Unweighted Graph

During the semester, you have learned basic graph theory and many graph algorithms. This assignment considers an **undirected**, **unweighted** graph $G = (V, E)$ with adjacency lists. You need to implement the following tasks:

1. **Two_Induced_Path(u, v, w)**: For three given vertices v, u and w, check if they form a **2-induced path**, i.e., if **exactly** two of the three edges (v, u), (u, w) and (v, w) belong to E.
2. **Exists_Euler_Cycle()**: Check if there exists an **Euler cycle** in G provided that G is **connected**. (A Euler cycle is a cycle that visits every edge exactly once.)
3. **Two_Colorable()**: Check if G is **2-colorable**. (A graph is *2-colorable* if its vertices can be assigned with 2 colors such that no two adjacent vertices share the same color).
4. **Max_Distance(v)**: Given a vertex v, provided that G is connected, compute the **maximum distance** from v to another vertex in V where the distance between two vertices is the minimum number of edges along a path between them.

Grading (24 points):

1. **Two_Induced_Path(u, v, w)** (4 points): An $O(|V|)$-time implementation gets 4 points, while an $O(|V|^2)$-time implementation only gets 2 points.
2. **Exists_Euler_Cycle()** (6 points): An $O(|E|)$-time implementation gets 6 points. If border cases are treated incorrectly, you will instead get 3 points.
3. **Two_Colorable()** (6 points): An $O(|E|)$-time implementation gets 6 points. If border cases are treated incorrectly, you will instead get 3 points.
4. **Max_Distance(v)** (8 points): An $O(|E|)$-time implementation gets 8 points, while an $O(|V^2|)$-time implementation only gets 4 points.

Comments for the code template:

- The Class Graph
  - **n** is the number of vertices in the graph.
  - **m** is the number of edges in the graph.
  - **degree[i]** is the degree of the i-th vertex in the graph.
  - **edges[i][j]** indicates the endpoint of the j-th edge of the i-th vertex in the graph.
  - **visited[i]** indicates if the i-th vertex has been visited, which is useful for a depth-first search or a breadth-first search.
- We provide template functions to show how to do a depth-first search.
  - **DFS_Initialization()** sets all the n vertices as **unvisited**.
  - **DFS(v)** conducts a depth-first search from the vertex v.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You see the **percentage** of your submission in the CodeExpert system instead of points.

```java
public boolean Exists_Euler_Cycle()
{
  for(int i = 0 ; i<n ; i++) {
    if(degree[i] % 2 ==1) return false;
  }
  return true;
}


public boolean Two_Induced_Path(int u,int v, int w)
{
  int count = 0 ;
  for(int i = 0 ; i< degree[u] ; i++) {
    if(edges[u][i] == w) count ++ ;
  }
  for(int i = 0 ; i< degree[v] ; i++) {
    if(edges[v][i] == u) count ++ ;
  }
  for(int i = 0 ; i< degree[v] ; i++) {
    if(edges[v][i] == w) count ++ ;
  }
  if(count == 2) return true;
  return false;
}
```

```java
public int Max_Distance(int v)
{
  for(int i = 0 ; i<n ; i++) {
    visited[i] = false ;
  }
  int [] distance = new int [n] ;
  LinkedList<Integer> Q = new LinkedList<> () ;
  Q.addFirst(v) ;

  while(!Q.isEmpty()) {
    int i = Q.removeFirst() ;
    for(int j =0 ; j<degree[i] ; j++) {
      if(visited[edges[i][j]] == false) {
        visited[edges[i][j]] = true ;
        distance[edges[i][j]] = distance[i] + 1 ;
        Q.addLast(edges[i][j]) ;
      }
    }
  }

  int max = 0 ;

  for(int i = 0 ; i<n ; i++) {
    max = Math.max(max , distance[i]) ;
  }

  return max;
}
```

+AW.get(i).get(j)

My Sol. BFS

```java
public boolean Two_Colorable()
{
  for(int i = 0 ; i<n ; i++) {
    visited[i] = false ;
  }
  int [] color = new int [n] ;
  LinkedList<Integer> Q = new LinkedList<> () ;
  Q.addFirst(0) ;
  color[0] = 1 ;

  while(!Q.isEmpty()) {
    int i = Q.removeFirst() ;

    for(int j =0 ; j<degree[i] ; j++) {
      if(visited[edges[i][j]] == true) {
        if(color[edges[i][j]] == color[i]) return false;
      }
      if(visited[edges[i][j]] == false) {
        visited[edges[i][j]] = true ;
        if(color[i] == 1) color[edges[i][j]] = 2;
        if(color[i] == 2) color[edges[i][j]] = 1 ;
        Q.addLast(edges[i][j]) ;
      }
    }
  }

  return true ;
}
```

My Sol. DFS

```java
public boolean Two_Colorable()
{
  for(int i = 0 ; i<n ; i++) {
    visited[i] = false ;
  }

  int [] color = new int [n] ;

  for(int i = 0; i<n ; i++) {
    if(visited[i] == false) {
      color[i] = 1 ;

      dfsmodified(i , color) ;
    }
  }

  for(int i = 0 ; i<n ; i++) {
    if(color[i] == 3) return false ;
  }
  return true ;
}

public void dfsmodified(int v , int[] color ){
  visited[v] = true ;
  for(int i = 0 ; i<degree[v] ; i++) {
    if(visited[edges[v][i]] == true) {
      if(color[edges[v][i]] == color[v]) color[edges[v][i]] = 3 ;
    }
    else {
      if(color[v] == 1 ) color[edges[v][i]] =2 ;
      if(color[v] == 2 ) color[edges[v][i]] =1 ;
      dfsmodified(edges[v][i]  , color) ;
    }
  }
```

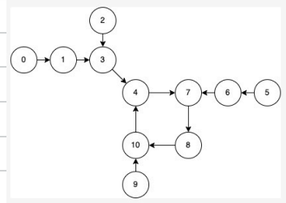
Master DFS

```java
public boolean Two_Colorable()
{
    char[] color_array=new char[n];

    for(int i=0;i<n;i++)
      color_array[i]=0;

    for(int i=0;i<n;i++){
      if(color_array[i]==0)
      {
        color_array[i]=1;
        if(DFS_Colorable(i,color_array)==false)
          return false;
      }

    }

    return true;
}
```

```java
private boolean DFS_Colorable(int index, char[] color_array)
{

    for(int i=0;i< degrees[index];i++)
    {
        if(color_array[edges[index][i]]==0)
        {
            if(color_array[index]==1)
                color_array[edges[index][i]]=2;
            else
                color_array[edges[index][i]]=1;
            if(DFS_Colorable(edges[index][i],color_array)==false)
              return false;
        }
        else if(color_array[edges[index][i]]==color_array[index])
            return false;
    }

    return true;
}
```

Players on a Graph

You are given a directed graph with n nodes and n edges, such that the out-degree of each node is 1. You are guaranteed that, viewing the edges as *undirected*, the graph is connected. Such a graph always contains exactly one cycle. Here is an example for $n = 11$, in which the nodes 4, 7, 8, and 10 form the cycle:



In the beginning, each node of the graph contains a player. Then a timer starts, and at each time step, each players moves to the node indicated by the edge of its current node. The timer continues forever. Note that multiple players may arrive at the same time at the same node -- that is allowed.

Given such a graph, you have to answer queries of the following type:

1. **cycleLength()**: Return the length of the cycle of the graph.
2. **distanceToCycle(x)**: For the player that starts at node x, return the number of edges that it has to traverse to reach the cycle. (If x is already in the cycle, the answer is 0.)
3. **firstNodeInCycle(x)**: After some time steps, the player that starts at node x reaches the cycle. Return the first node in the cycle that this player reaches. (If x is already in the cycle, the answer is x.)
4. **distanceInCycle(x, y)**: After some time steps, both the player that starts at node x and the player that starts at node y have reached the cycle. Return the distance between them on the cycle (this distance is calculated viewing the edges of the cycle as *undirected*; that is, the minimum of the clockwise distance and counterclockwise distance on the cycle).

   (Note that the two players may reach the cycle at different time steps.)

You have to implement the four methods described above. In addition, we provide an **intialize()** method which we guarantee to run exactly once, before the queries. Feel free to use this method, for example, to initialize information that you want available in all of the queries.

Grading (24 points):

1. **cycleLength()** (4 points): This query will be run at most once. An $O(n)$-time implementation gets 4 points.
2. **distanceToCycle(x)** (6 points): If q is the number of queries of this type, an $O(n + q)$-time implementation gets 6 points and an $O(nq)$-time implementation gets 3 points.
3. **firstNodeInCycle(x)** (6 points): If q is the number of queries of this type, an $O(n + q)$-time implementation gets 6 points and an $O(nq)$-time implementation gets 3 points.
4. **distanceInCycle(x, y)** (8 points): If q is the number of queries of this type, an $O(n + q)$-time implementation gets 8 points and an $O(nq)$-time implementation gets 4 points.

Attention:

- You are **NOT** allowed to use imports explicitly or implicitly, other than the imports already included in the code template.
- When you submit the code, you see your score as a **percentage**, not as a number of points.

Binary Search Tree

During the semester, you have learned binary search trees and also implemented a corresponding programming exercise. In this exercise, your task is to implement the following four kinds of queries:

1. **min()**: If the current binary search tree is not empty, return the **minimum key** in the binary search tree; otherwise, return -1.
2. **depth(k)**: Given an integer k, if the current binary search tree contains a node whose key is k, return the **depth** of the node; otherwise, return -1. Note that the depth of the *root* is defined to be **0**.
3. **children_of_node(k)**: Given an integer k, if the current binary search tree contains a node whose key is k, return **the number of children** of the node; otherwise, return -1.
4. **key_of_rank(r)**: Given an integer r, if the number of nodes in the current binary tree is at least r, return **the r-th smallest key** in the binary search tree; otherwise, return -1.

Note:

- For the last kind of queries, *key_of_rank(r)*, you have to augment the binary search tree.
- You can safely assume that all keys in the binary search tree are distinct and nonnegative.

Grading (24 points): Let h be the height of the current binary search tree.

1. **min()** (5 points): An $O(h)$-time implementation gets 5 points. If border cases are treated incorrectly, you will instead get 4 points.
2. **depth(k)** (5 points): An $O(h)$-time implementation gets 5 points. If border cases are treated incorrectly, you will instead get 4 points.
3. **children_of_node(k)** (5 points): An $O(h)$-time implementation gets 5 points. If border cases are treated incorrectly, you will instead get 4 points.
4. **key_of_rank(r)** (9 points): An $O(h)$-time implementation gets 9 points, while an $O(n)$-time implementation gets 5 points, where n is the number of nodes in the current binary search tree. If border cases are treated incorrectly, you will instead get 8 points and 4 points, respectively.

Attention:

- You are NOT allowed to use imports explicitly or implicitly other than the imports already in the code template.
- You only can see the **percentage** of your submission in the CodeExpert system instead of points.

2020 winter

```java
public int min(){
  if(this.root == null) return -1 ;
  TreeNode ans = root ;
  while(ans.left != null ) {
    ans = ans.left ;
  }

  return ans.key ;
}
```

```java
public int depth(int key){
  int depth = 0 ;
  if(this.root == null) return -1 ;
  TreeNode curr = root ;
  while(curr != null) {
    if(curr.key == key ) return depth ;
    if(curr.key < key){
      curr = curr.right ;
      depth ++;
    }
    if(curr.key > key) {
      curr = curr.left ;
      depth ++ ;
    }

  }

  return -1 ;
}
```

```java
public int children_of_node(int key){
  TreeNode searchedNode = search(key) ;
  if(searchedNode == null) return -1 ;

  if(searchedNode.left != null && searchedNode.right != null) return 2 ;
  if(searchedNode.left != null && searchedNode.right == null) return 1 ;
  if(searchedNode.left == null && searchedNode.right != null) return 1 ;
  if(searchedNode.left == null && searchedNode.right == null) return 0 ;

  return 0 ;
}
```

Layout :

```java
class TreeNode{
    public int key;
    public TreeNode parent;
    public TreeNode left;
    public TreeNode right;

    TreeNode(int key){
        this.key=key;
        this.parent=null;
        this.left=null;
        this.right=null;
    }
}
```

```java
class TreeNode{
    public int key;
    public TreeNode parent;
    public TreeNode left;
    public TreeNode right;
    public int size;          //the size of the subtree

    TreeNode(int key){
        this.key=key;
        this.parent=null;
        this.left=null;
        this.right=null;
        this.size=1;
    }
}
```

```java
class BinaryTree{

    TreeNode root;

    BinaryTree(){
      this.root=null;
    }
```

```java
public void insert(int key){

  if(root==null){
    root=new TreeNode(key);
  }
  else{
    insert(root,key);
  }
}


public void insert(TreeNode node,int key){

  if(key<node.key){
    if(node.left!=null){
      insert(node.left,key);
    }
    else{
      node.left=new TreeNode(key);
      node.left.parent=node;
    }
  }
  else{
    if(node.right!=null){
      insert(node.right,key);
    }
    else{
      node.right=new TreeNode(key);
      node.right.parent=node;
    }
  }
}
```

```java
public void insert(int key){

  if(root==null){
    root=new TreeNode(key);
  }
  else{
    insert(root,key);
  }
}


public void insert(TreeNode node,int key){

  node.size++;

  if(key<node.key){
    if(node.left!=null){
      insert(node.left,key);
    }
    else{
      node.left=new TreeNode(key);
      node.left.parent=node;
    }
  }
  else{
    if(node.right!=null){
      insert(node.right,key);
    }
    else{
      node.right=new TreeNode(key);
      node.right.parent=node;
    }
  }
}
```

```java
public TreeNode search(int key){

  if(root==null){
    return null;
  }
  else{
    return search(root,key);
  }
}

public TreeNode search(TreeNode node,int key){
  if(node==null){
    return null;
  }
  else if(key==node.key){
    return node;
  }
  else{
    if(key<node.key){
      return search(node.left,key);
    }
    else{
      return search(node.right,key);
    }
  }
}
```

```java
public void delete(int key){

  // find the node to be deleted
  TreeNode node=search(key);

  if(node==null){
    return;
  }

  if(node.left==null){
    if(node.right==null){
      // no child

      if(node==root){
        root=null;
      }
      else{
        if(node.parent.left==node){
          node.parent.left=null;
        }
        else{
          node.parent.right=null;
        }
      }
    }
    else{
      // only right child

      if(node==root){
        root=node.right;
        root.parent=null;
      }
      else{
        if(node.parent.left==node){
          node.parent.left=node.right;
          node.right.parent=node.parent;
        }
        else{
          node.parent.right=node.right;
          node.right.parent=node.parent;
        }
      }
    }

  }
  else{
    if(node.right==null){
      //only left child

      if(node==root){
        root=node.left;
        root.parent=null;
      }
      else{
        if(node.parent.left==node)
        {
          node.parent.left=node.left;
          node.left.parent=node.parent;
        }
        else
        {
          node.parent.right=node.left;
          node.left.parent=node.parent;
        }
      }
    }
    else{
      // two children

      TreeNode swap_node=node.left;

      while(swap_node.right!=null){
        swap_node=swap_node.right;
      }

      node.key=swap_node.key;

      if(swap_node.left!=null){
        // swap_node has left Child

        if(swap_node.parent.left==swap_node){
          swap_node.parent.left=swap_node.left;
          swap_node.left.parent=swap_node.parent;
        }
        else{
          swap_node.parent.right=swap_node.left;
          swap_node.left.parent=swap_node.parent;
        }
      }
      else{
        // swap_node has no child

        if(swap_node.parent.left==swap_node){
          swap_node.parent.left=null;
        }
        else{
          swap_node.parent.right=null;
        }
      }
    }
  }

}
```

```java
public void delete(int key){

  // find the node to be deleted
  TreeNode node=search(key);

  if(node==null){
    return;
  }

  TreeNode update_node=null;  // The node that we bactrack from to update the sizes

  if(node.left==null){
    if(node.right==null){
      // no child

      if(node==root){
        root=null;
      }
      else{
        if(node.parent.left==node){
          node.parent.left=null;
        }
        else{
          node.parent.right=null;
        }
      }

      update_node=node.parent;
    }
    else{
      // only right child

      if(node==root){
        root=node.right;
        root.parent=null;
      }
      else{
        if(node.parent.left==node){
          node.parent.left=node.right;
          node.right.parent=node.parent;
        }
        else{
          node.parent.right=node.right;
          node.right.parent=node.parent;
        }
      }

      update_node=node.parent;
    }

  }
  else{
    if(node.right==null){
      //only left child

      if(node==root){
        root=node.left;
        root.parent=null;
      }
      else{
        if(node.parent.left==node)
        {
          node.parent.left=node.left;
          node.left.parent=node.parent;
        }
        else
        {
          node.parent.right=node.left;
          node.left.parent=node.parent;
        }
      }

      update_node=node.parent;
    }
    else{
      // two children

      TreeNode swap_node=node.left;

      while(swap_node.right!=null){
        swap_node=swap_node.right;
      }

      node.key=swap_node.key;

      if(swap_node.left!=null){
        // swap_node has left Child

        if(swap_node.parent.left==swap_node){
          swap_node.parent.left=swap_node.left;
          swap_node.left.parent=swap_node.parent;
        }
        else{
          swap_node.parent.right=swap_node.left;
          swap_node.left.parent=swap_node.parent;
        }
      }
      else{
        // swap_node has no child

        if(swap_node.parent.left==swap_node){
          swap_node.parent.left=null;
        }
        else{
          swap_node.parent.right=null;
        }
      }

      update_node=swap_node.parent;
    }
  }

  while(update_node!=null){
    update_node.size--;
    update_node=update_node.parent;
  }
}
```

```java
public int key_of_rank(int rank)
{
    if(rank <=0 || root==null || root.size <rank){
        return -1;
    }
    else{
        return key_of_rank(root,rank);
    }

}

int key_of_rank(TreeNode node, int rank){

    if(node.left!=null){
        if(node.left.size >= rank){
            return key_of_rank(node.left,rank);
        }
        else if(rank == node.left.size+1){
            return node.key;
        }
        else{
            return key_of_rank(node.right, rank - node.left.size - 1);
        }

    }
    else{
        if(rank==1){
            return node.key;
        }
        else {
            return key_of_rank(node.right, rank-1);
        }

    }

}
```

left < rank → go to right sub

left is 0 ✓

this go to right

Binary Search Tree

In the file Main.java you are given an implementation of a binary search tree in which pairs $(\text{key}, \text{value})$ can be inserted. The nodes of the tree are ordered by the key, but each node also stores the value associated with the key. We did not implement a deletion operation, and there will be no deletion operations in this task.

Your task is to augment this binary search tree so that it supports queries of the following type: given an integer x, return the maximum value associated with a key such that the key is less than or equal to x. If there is no key less than or equal to x, you need to return 0.

Specifically, your task is to implement the method **query(x)** in the class **BinaryTree**. To do so, you may need to add auxiliary variables and methods to the classes **TreeNode** and **BinaryTree**, and also to modify some of the existing methods.

You get one point for each passing test set. To pass both test sets, the time complexity per query needs to be $O(h)$, where h is the height of the binary search tree.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

```
class BinaryTree {
  TreeNode root;

  BinaryTree() {
    this.root = null;
  }
```

```
// Idea: Add a max variable to TreeNode, which keeps track of the maximum value
// in the subtree of the node. The max variable must be updated in the insert
// opreation. Then, for the query, we start at the root, and if x is smaller
// than the key, we search only in the left subtree, and if x is larger than or
// equal to the key, we search only in the right subtree but take into
// consideration what the maximum value is in the left subtree.
```

```
// Inserts a node with the given key and value in the bina
// BinaryTree.root.
public void insert(int key, int value) {
  if (root == null) {
    root = new TreeNode(key, value);
  } else {
    insert(root, key, value);
  }
}

// Inserts a node with the given key and value in the bina
// node.
public void insert(TreeNode node, int key, int value) {
  if (key < node.key) { // insert in left subtree
    if (node.left != null) {
      insert(node.left, key, value);
    } else {
      node.left = new TreeNode(key, value);
      node.left.parent = node;
    }
  } else { // insert in right subtree
    if (node.right != null) {
      insert(node.right, key, value);
    } else {
      node.right = new TreeNode(key, value);
      node.right.parent = node;
    }
  }
}
```

```
// Inserts a node with the given key and value in the binary tree rooted at
// BinaryTree.root.
public void insert(int key, int value) {
  if (root == null) {
    root = new TreeNode(key, value);
  } else {
    insert(root, key, value);
  }
}

// Inserts a node with the given key and value in the binary tree rooted at
// node.
public void insert(TreeNode node, int key, int value) {
  if (key < node.key) { // insert in left subtree
    if (node.left != null) {
      insert(node.left, key, value);
      updateMax(node);
    } else {
      node.left = new TreeNode(key, value);
      node.left.parent = node;
      updateMax(node);
    }
  } else { // insert in right subtree
    if (node.right != null) {
      insert(node.right, key, value);
      updateMax(node);
    } else {
      node.right = new TreeNode(key, value);
      node.right.parent = node;
      updateMax(node);
    }
  }
}
```

Semester Exercise

```
class TreeNode {
  public int key;
  public int value;
  public TreeNode parent;
  public TreeNode left;
  public TreeNode right;

  TreeNode(int key, int value) {
    this.key = key;
    this.value = value;
    this.parent = null;
    this.left = null;
    this.right = null;
  }
}
```

```
class TreeNode {
  public int key;
  public int value;
  public TreeNode parent;
  public TreeNode left;
  public TreeNode right;
  public int max;

  TreeNode(int key, int value) {
    this.key = key;
    this.value = value;
    this.parent = null;
    this.left = null;
    this.right = null;
    this.max = value;
  }
}
```

```
// Updates the max of the node based on its children.
private void updateMax(TreeNode node) {
  node.max = node.value;
  if (node.left != null) {
    node.max = Math.max(node.max, node.left.max);
  }
  if (node.right != null) {
    node.max = Math.max(node.max, node.right.max);
  }
}
```

```
// Returns the maximum value associated with a key that is less than or equal
// to x in the binary tree rooted at BinaryTree.root.
public int query(int x) {
  // TODO: implement.
  return 0;
}
```

```
// Returns the maximum value associated with a key that is less than or equal
// to x in the binary tree rooted at BinaryTree.root.
public int query(int x) {
  return query(root, x);
}

// Returns the maximum value associated with a key that is less than or equal
// to x in the binary tree rooted at node.
public int query(TreeNode node, int x) {
  if (node == null) {
    return 0;
  }
  if (node.key <= x) {
    if (node.left != null) {
      return Math.max(node.value, Math.max(node.left.max, query(node.right, x)));
    } else {
      return Math.max(node.value, query(node.right, x));
    }
  } else {
    return query(node.left, x);
  }
}
```

Semester Exercise

Navigation

Imagine that you are travelling in a city with n tram stations and two tram lines A and B. A and B are both arrays of n integers. If you are currently at station i, you can choose to either take tram line A to go to station $A[i]$ or take tram line B to go to station $B[i]$. Now, given a start station s and a target station t, your goal is to figure out whether you can reach station t from station s using the two tram lines.

For example, given $A = [1, 3, 1, 4, 3]$ and $B = [1, 0, 3, 0, 1]$. If $s = 2$ and $t = 4$, the answer is yes, because there is a path $2 \to 3 \to 4$. If $s = 0$ and $t = 2$, the answer is no.

You need to implement your solution as a method **travel(**A**,** B**,** n**,** s**,** t**)** in the file Main.java. You get one point for each passing test set. To pass both test sets, your solution is expected to run in time $O(n)$.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

```java
public static boolean travel(int[] A, int[] B, int n, int s, int t) {
  // TODO
  boolean [] visited = new boolean [n] ;
  visited[s] = true ;

  LinkedList<Integer> Q = new LinkedList<>() ;
  Q.addFirst(s) ;
  while(!Q.isEmpty()) {
    int curr = Q.removeFirst() ;
    if(visited[A[curr]] == false) {
      visited[A[curr]] = true ;
      Q.addLast(A[curr]) ;
    }
    if(visited[B[curr]] == false) {
      visited[B[curr]] = true ;
      Q.addLast(B[curr]) ;
    }
  }

  if(visited[t] == true) return true ;
  return false;
}
```

Lava

You are in a 2-dimentional lava field M of size $n \times m$, where $M[i][j]$ is a $\{0, 1\}$ value denoting whether position (i, j) has lava (1 denotes that it has lava and 0 denotes that it does not have lava). In each step, you can move 1 step horizontally or vertically, but you can only walk on positions with no lava and you cannot move out of the lava field. Your goal is to compute the smallest number of steps that you need to go from a given position $s = (i_1, j_1)$ to a given position $t = (i_2, j_2)$ (you can assume that t is always reachable from s).

For example, given

$$M = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

To get from position $(0, 2)$ to position $(3, 2)$, you can either take $(0, 2) \to (1, 2) \to (1, 1) \to (1, 0) \to (2, 0) \to (3, 0) \to (3, 1) \to (3, 2)$ which has 7 steps, or you can take $(0, 2) \to (1, 2) \to (1, 3) \to (2, 3) \to (3, 3) \to (3, 2)$ which has 5 steps. Hence, the smallest number of steps you need is 5.

You need to implement your solution as a method **steps(**M**,** n**,** m**,** $i1$**,** $j1$**,** $i2$**,** $j2$**)** in the file Main.java. You get one point for each passing test set. To pass both test sets, your solution is expected to run in time $O(n * m)$.

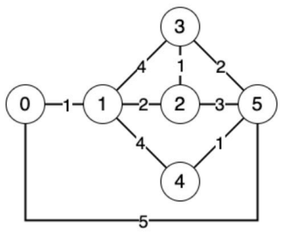
Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

```java
public static int steps(int n, int m, int[][] M, int i1, int j1, int i2, int j2) {
  // TODO
  int[][] distance = new int [n][m] ;
  boolean [][] visited = new boolean[n][m] ;

  for(int i = 0 ; i <n ; i ++) {
    for(int j = 0 ; j<m ; j++) {
      distance[i][j] = Integer.MAX_VALUE ;
    }
  }

  LinkedList <Pair> Q = new LinkedList<> () ;
  Pair s = new Pair(i1 , j1) ;
  visited[i1][j1] = true ;
  distance[i1][j1] = 0 ;
  Q.addFirst(s) ;

  while(!Q.isEmpty()) {
    Pair curr = Q.removeFirst() ;
    int i = curr.x ;
    int j = curr.y;
    if(i+1 < n && visited[i+1][j] == false && M[i+1][j] != 1 ) {
      visited[i+1][j] = true ;
      distance[i+1][j] = Math.min( distance[i+1][j] , distance[i][j] + 1 ) ;
      Q.addLast( new Pair(i+1 , j )) ;
    }
    if(i-1 >= 0 && visited[i-1][j] == false && M[i-1][j] != 1) {
      visited[i-1][j] = true ;
      distance[i-1][j] = Math.min(distance[i-1][j] , distance[i][j] + 1 ) ;
      Q.addLast( new Pair(i-1 , j )) ;
    }
    if(j+1 < m && visited[i][j+1] == false && M[i][j+1] != 1) {
      visited[i][j+1] = true ;
      distance[i][j+1] = Math.min(distance[i][j+1] , distance[i][j] + 1 ) ;
      Q.addLast( new Pair(i , j+1)) ;
    }
    if(j-1 >= 0 && visited[i][j-1] == false && M[i][j-1] != 1) {
      visited[i][j-1] = true ;
      distance[i][j-1] = Math.min(distance[i][j-1] , distance[i][j] + 1 ) ;
      Q.addLast( new Pair(i, j-1 )) ;
    }
  }


  return distance[i2][j2];
}
```

Minimum Cost Edges

You are given an undirected connected graph with n vertices and m weighted edges, where each weight is a positive integer. You are also given two distinct vertices s and t. We call an edge useless if there does not exist any path of minimum cost between s and t that uses this edge. Your task is to return the number of useless edges in the graph.

For example, for the graph in the picture, for $s = 1$ and $t = 5$, the minimum cost is 5, and there are 3 useless edges: $\{0, 1\}$, $\{0, 5\}$, and $\{1, 3\}$.



You need to implement your solution as a method **countEdges** in the file Main.java. You get one point for each passing test set. To pass both test sets correctly, your solution needs to run in $O(m \log m)$ time.

Note: The large.in files for this exercise are very large and it may not be possible to run them with the "run" button (that is, with In.open("public/test.large")). Please test these instead with the "test" (laboratory flask) button, which tests both public tests.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

```
// IDEA: Do Dijkstra two times: once from s and once from t. Then an edge (u, v, w)
// is on a path of minimum cost if and only if sCost[u] + tCost[v] + w = sCost[t]
// or sCost[v] + tCost[u] + w = sCost[t].
```

```
public static void dijkstra ( int n , int source ,  ArrayList<ArrayList<Integer>> Ev,ArrayList<ArrayList<Integer>> Ew , int[] cost ) {
  for(int i = 0 ; i< n ; i ++) {
    cost[i] = -1 ;
  }
  cost[source] = 0 ;
  PriorityQueue <Tuple> Q = new PriorityQueue <> () ;
  Q.add(new Tuple(source , 0)) ;
  while(!Q.isEmpty()) {
    Tuple curr = Q.poll() ;
    if(curr.value != cost[curr.x]) {
      continue ;
    }

    for(int i = 0 ; i<Ev.get(curr.x).size() ; i++) {
      if(cost[Ev.get(curr.x).get(i)] == -1 || cost[curr.x] + Ew.get(curr.x).get(i) < cost[Ev.get(curr.x).get(i)]) {
        cost[Ev.get(curr.x).get(i)] =  cost[curr.x] + Ew.get(curr.x).get(i) ;
        Q.add(new Tuple(Ev.get(curr.x).get(i) , cost[curr.x] + Ew.get(curr.x).get(i)) )  ;
      }
    }

  }
}
```

```
public static int countEdges(int n, int m, int s, int t, ArrayList<ArrayList<Integer>> Ev,
                             ArrayList<ArrayList<Integer>> Ew) {
  // Ev is an adjacency list such that Ev.get(i) is the list of neighbors of
  // vertex i. Specifically:
  //   Ev.get(i).get(j) = the j-th neighbor of vertex i
  //   Ew.get(i).get(j) = the weight of the edge from vertex i to its j-th neighbor
  // The number of elements of Ev.get(i) is given by Ev.get(i).size() (same for Ew).
  int [] fromS = new int [n] ;
  int [] fromT = new int [n] ;
  dijkstra(n , s , Ev , Ew, fromS) ;
  dijkstra(n , t , Ev , Ew, fromT) ;
  int edges = 0 ;
  for(int i = 0 ; i <n ; i ++) {
    for(int j = 0 ; j< Ev.get(i).size() ; j++) {
      if(fromS[i] +Ew.get(i).get(j)  + fromT[Ev.get(i).get(j) ] != fromS[t]  && fromT[i] + Ew.get(i).get(j)  + fromS[Ev.get(i).get(j) ] != fromS[t] ) {
        edges ++ ;
      }
    }
  }

  return edges/2 ;
}
```

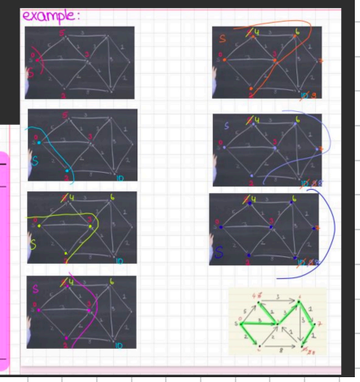
```
public static int[] dijkstra(ArrayList<ArrayList<Integer>> Ev, ArrayList<ArrayList<Integer>> Ew, int n, int s) {
    int[] distance = new int[n];
    Arrays.fill(distance, Integer.MAX_VALUE);
    distance[s] = 0;

    PriorityQueue<Integer> minHeap = new PriorityQueue<>((a, b) -> Integer.compare(distance[a], distance[b]));
    minHeap.offer(s);

    while (!minHeap.isEmpty()) {
        int u = minHeap.poll();
        for (int i = 0; i < Ev.get(u).size(); i++) {
            int v = Ev.get(u).get(i);
            int weight = Ew.get(u).get(i);
            if (distance[v] > distance[u] + weight) {
                distance[v] = distance[u] + weight;
                minHeap.offer(v);
            }
        }
    }

    return distance;
}
}
```

Don't know why, lol
Sorry for the screen :(

Shortest Path

Dijkstra

example:



Runtime: $O((m+n) \cdot \log(n))$

Algorithm 6 Dijkstra(s)

```
1: d[s] ← 0;   d[v] ← ∞  ∀v ∈ V \ {s}
2: S ← ∅
3: H ← make-heap(V); decrease-key(H, s, 0)
4: while S ≠ V do
5:     v* ← extract-min(H)
6:     S ← S ∪ {v*}
7:     for (v*, v) ∈ E, v ∉ S do
8:         d[v] ← min{d[v], d[v*] + c(v*, v)}
9:         decrease-key(H, v, d[v])
```

Railway

You are in charge of a railway system with n stops indexed by $\{0, 1, \ldots, n-1\}$ and a set of m old railway tracks E. You want to reconstruct some of the railway tracks such that the n stops are still connected in the new railway network. For the remaining railway tracks, you plan to sell all of them to other companies. Each railway track i is associated with a non-negative reconstruction cost A_i and a non-negative selling price B_i. If the set of railway tracks to be reconstructed is R, the total cost of the reconstruction is $C(R) = \sum_{i \in R} A_i - \sum_{i \notin R} B_i$.

Given n, m, the old railway tracks E, reconstruction costs A, and selling prices B, your goal is to find the minimum total reconstruction cost $C(R)$ such that the reconstructed railway network R is still connected. Notice that $C(R)$ can be negative here, which means you make more money in selling the railway tracks than reconstruction.

For example, given $n = 4$, $m = 5$, $E = \{(0,1), (0,2), (0,3), (1,2), (1,3)\}$, $A = \{5, 5, 4, 6, 4\}$ and $B = \{2, 1, 4, 2, 3\}$, the minimum total cost is 8 by reconstructing $\{(0,1), (0,2), (1,3)\}$ and selling $\{(0,3), (1,2)\}$.

You need to implement your solution as a method **minCost(**n, m, E, A, B**)** in the file Main.java. You get one point for each passing test set. To pass both test sets, your solution is expected to run in time $O(m \log m)$.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

```java
public static int minCost(int n, int m, Edge[] E, int[] A, int[] B) {
  // The idea is to use Kruskal's algorithm to construct an MST.
  // The weight of each edge is A + B because, by reconstructing each railway track,
  // you lose the reconstruction cost and the money you could have got for selling the track.
  Tuple[] edges = new Tuple[m];
  for (int i = 0; i < m; i++) {
    edges[i] = new Tuple(E[i].u, E[i].v, A[i] + B[i]);
  }
  Arrays.sort(edges);

  // Initialize cost.
  int cost = 0;
  for (int i=0; i<m; i++) {
    cost -= B[i];
  }

  // Run Kruskal's algorithm.
  UnionFind uf = new UnionFind(n);
  int total = 0;
  int cur = 0;
  while (total < n-1) {
    Tuple edge = edges[cur];
    if (uf.find(edge.u) != uf.find(edge.v)) {
      cost += edge.w;
      total++;
      uf.union(edge.u, edge.v);
    }
    cur++;
  }

  return cost;
}
```

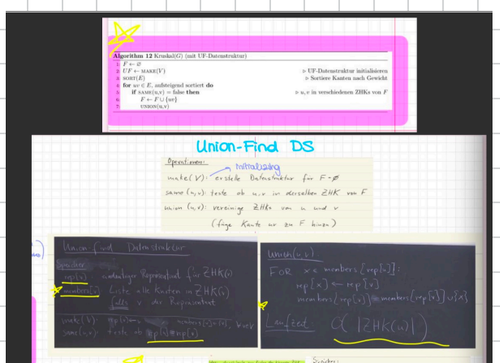
```java
// Implement Union Find data structure here.
static class UnionFind {
  public int[] parent;
  public int[] rank;

  public UnionFind(int n) {
    parent = new int[n];
    rank = new int[n];

    for(int i=0; i<n; i++) {
      parent[i] = i;
      rank[i] = 0;
    }
  }

  public int find(int v) {
    if (parent[v] == v) {
      return v;
    }
    return find(parent[v]);
  }

  public void union(int u, int v) {
    int uroot = find(u);
    int vroot = find(v);

    if (rank[uroot] < rank[vroot]) {
      parent[uroot] = vroot;
    } else if (rank[uroot] > rank[vroot]) {
      parent[vroot] = uroot;
    } else {
      parent[vroot] = uroot;
      rank[uroot]++;
    }
  }
}
```

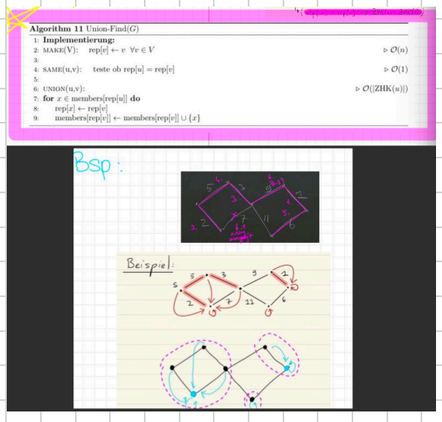
layout:

```java
// Edge class for input edge
static class Edge {
  public int u;
  public int v;

  public Edge (int u, int v) {
    this.u = u;
    this.v = v;
  }
}
```

```java
// You can use the following class we define for you. This defines "Tuple"
// objects that can be sorted according to the compareTo function that you
// define. This allows you, for example, to use Java Arrays.sort to sort a
// collection. Of course, in the current form the class is empty,
// you need to add variables, etc.
/*
static class Tuple implements Comparable<Tuple> {
  public Tuple() {

  }

  // In case you want to sort a custom class you need to have this method
  @Override public int compareTo(Tuple other) {
    // in case this is > than other, return a positive number,
    // otherwise a negative number
    return 0;
  }
}
*/

public static int minCost(int n, int m, Edge[] E, int[] A, int[] B) {
  // TODO
  return 0;
}
```

```java
// Tuple class that implements Java Comparable interface for weighted edge.
static class Tuple implements Comparable<Tuple> {
  int u;
  int v;
  int w;

  public Tuple() {

  }

  public Tuple(int u, int v, int w) {
    this.u = u;
    this.v = v;
    this.w = w;
  }

  @Override public int compareTo(Tuple other) {
    return this.w - other.w;
  }
}
```





Paths with Small Weights

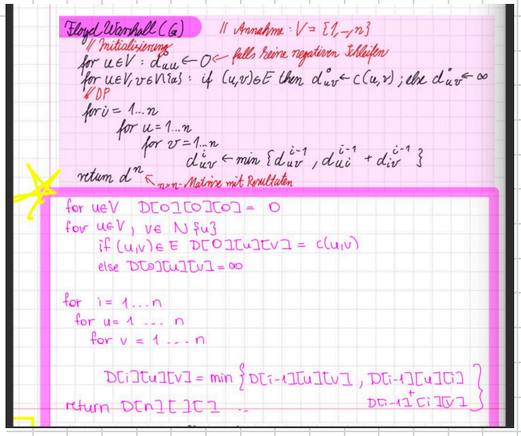
You are given an undirected connected graph with n vertices and m weighted edges, where each weight is a positive integer. You need to answer multiple queries of the following type: given two distinct vertices s and t, what is the minimum positive integer x such that you can travel between s and t on paths that all have weight at most x? (In other words, over all the paths that connect s and t, pick the one for which the maximum weight of an edge is the smallest, and output this weight.)

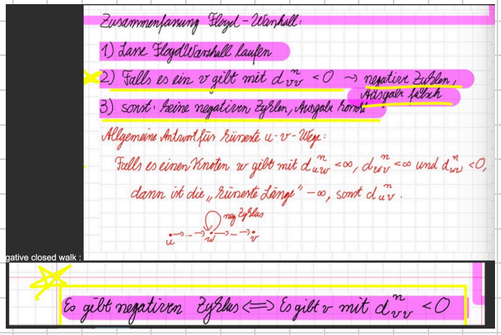
You need to implement your solution as two methods **initialize()** and **query(**s **,** t**)** in the file Main.java. The method **initialize()** is called exactly once, before any of the query methods is called. (Note that there are at most n^2 distinct queries.) You get one point for each passing test set. To pass both test sets, the total time complexity needs to be $O(n^3)$.

Hint: For the easiest solution, try to implement **initialize()** in time $O(n^3)$ and each query in time $O(1)$.

Note: The large.in files for this exercise are very large and it may not be possible to run them with the "run" button (that is, with In.open("public/test.large")). Please test these instead with the "test" (laboratory flask) button, which tests both public tests.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.






```java
public static void initialize(int n, int m, int[] E1, int[] E2, int[] Ew) {
  // E1, E2, Ew are arrays of integers of length m, where the i-th element
  // represents an edge between E1[i] and E2[i] of weight Ew[i]

  // TO DO
}

public static int query(int n, int m, int[] E1, int[] E2, int[] Ew, int x, int y) {
  // E1, E2, Ew are arrays of integers of length m, where the i-th element
  // represents an edge between E1[i] and E2[i] of weight Ew[i]

  // TO DO
  return 0;
}
```

```java
public static int[][] A;
public static int[][] D;

public static void initialize(int n, int m, int[] E1, int[] E2, int[] Ew) {
  // IDEA: Floyd-Warshall computing D[i][j] = the minimum cost of an edge
  // for a path between i and j.

  A = new int[n][n];
  D = new int[n][n];
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      D[i][j] = Integer.MAX_VALUE;
    }
  }
  for (int i = 0; i < m; i++) {
    A[E1[i]][E2[i]] = Ew[i];
    A[E2[i]][E1[i]] = Ew[i];
    D[E1[i]][E2[i]] = Ew[i];
    D[E2[i]][E1[i]] = Ew[i];
  }
  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (i != j && i != k && j != k && Math.max(D[i][k], D[k][j]) < D[i][j]) {
          D[i][j] = Math.max(D[i][k], D[k][j]);
        }
      }
    }
  }
}

public static int query(int n, int m, int[] E1, int[] E2, int[] Ew, int x, int y) {
  return D[x][y];
}
```