|  |
| --- |
| Hawaii Pacific University |
| ACM Algorithms |
| Algorithm Handbook |

|  |
| --- |
| Jiru Xu (Jazz)  11-5-2017 |

ACM ALGORITHM

Basics 2

Useful built-in libraries 3

bitwise operation 4

Non-built-in Data structure 5

Segment Tree 5

Union-Find 10

Binary Index Tree 12

dynamic programming 13

Longest Increasing Subsequence 13

Maximum Range Sum 14

0-1Knapsack 15

Coin Change 16

Travelling Salesman Problem 17

graph 18

Depth First Search/Breadth First Search 18

Tree Diameter 19

Bridge 20

Articulation Points 21

Topological Sort 22

Strongly connected component 24

Single Source Shortest Path 25

All Pair Shortest Path 26

Bipartite Matching 26

Minimum Spanning Tree 27

Maximum Flow 30

Geometry 32

Mathematics 36

Base Conversion 36

Modulo Power 36

Prime Number 36

Extended Euclid Theory 38

Fibonacci Number 38

Linear Recurrence 39

Gaussian Elimination 40

Reference 41

**Basics**

Standard inputs outputs for Java

Scanner in **=** **new** Scanner**(**System**.**in**);**

int a **=** in**.**nextInt**();**

System**.**out**.**println**(**a**);**

In ACM contest, you can only write code in one single file. So, if you want to create another class, do this:

public class Main {

static class C {

int a, b;

}

public static void main(String[] args) {

C something = new C();

}

}

During the contest, in most of the situations, you will get three results when you submit your code:

AC(Accepted), WA(Wrong Answer) and TLE(Time Limit Exceeded)

Well, when you get WA, you either entirely go wrong direction or you miss some boundary cases. For example, let’s say the problem is: given two integers A and B, where -231 ≤ A, B ≤ 231 – 1, output the sum of A and B. Looks easy enough. Well if your code is look like this:

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int A = in.nextInt(), B = in.nextInt();

System.out.println(A + B);

}

Unfortunately, you will get WA. It works well with small numbers, but if A is 2×109 and B is also 2×109 (they are less than 231 - 1), your output will be -294967296. This is because of the integer overflow. To solve this, use long instead of int. Even the simplest problem has boundary cases, so always check the boundary case!

When you get TLE, in most cases, the reason is you are using a wrong algorithm or you didn’t optimize it well (Other rare cases maybe that you write some infinite loops, but you will usually notice it when testing the test cases). So, how do you find out what cause the TLE? Well, you can analyze the runtime very quickly by looking at the input size. Let’s say the problem is sort the list, the size of the list will not exceed 106. If you use the easy sorting algorithm like bubble sort, insertion sort or selection sort, you will get TLE since their complexity are O(n2), and base on the maximum input size, the maximum operations will be 1012 (you will usually get TLE if your operations are over 108). However, if you are using those O(nlogn) approach sorting algorithm like merge sort, the maximum operation will be approximately 2×107, which will be fast enough. So, analyze the runtime of your algorithm before you implement it!

FYI: In real contest, you would do this:

int[] array = new int[]{4, 3, 5, 2, 1};

Arrays.sort(array); //or Collections.sort(array) if array is a List object

rather than implement a sorting algorithm by yourself. :-) Just don’t forget to import java.util.\*;

Those built-in methods are all implemented using O(nlogn) approach.

**Useful built-in libraries and methods**

Before writing any code, import java.util.\*; every time.

Math class is very useful, there are tons of static methods in this class.

double a **=** 3.0**,** b **=** 4.0**;**

double max **=** Math**.**max**(**a**,** b**);**

double min **=** Math**.**min**(**a**,** b**);**

double cos **=** Math**.**cos**(**Math**.**PI **/** 6.0**);** // in radian

double pow **=** Math**.**pow**(**a**,** b**);** // a^b

double sqrt **=** Math**.**sqrt**(**b**);**

double absdiff **=** Math**.**abs**(**a **-** b**);** // difference between two number

There are some useful static field of Integer and Double class

int maxInt **=** Integer**.**MAX\_VALUE**;**

int minInt **=** Integer**.**MIN\_VALUE**;**

double INF **=** Double**.**POSITIVE\_INFINITY**;**

double NINF **=** Double**.**NEGATIVE\_INFINITY**;**

You must be careful when using Integer.MAX\_VALUE and the min value. If you add one to Integer.MAX\_VALUE, you will get a negative number. Similarly, if you subtract one from min value, you will get a positive number. luckily, the infinity value in the double class won’t have this problem.

Here are some useful codes:

int**[]** array **=** **{**5**,** **-**1**,** **-**6**,** 4**,** 2**,** 9**};**

//find max value in the array

int max **=** Integer**.**MIN\_VALUE**;**

**for(**int e **:** array**)**

max **=** Math**.**max**(**max**,** e**);**

//find min value in the array

int min **=** Integer**.**MAX\_VALUE**;**

**for(**int e **:** array**)**

min **=** Math**.**min**(**min**,** e**);**

//fill a 1D array with a value

Arrays**.**fill**(**array**,** **-**1**);**

//fill a multi-dimension array with a value. Took 3D array as an example

int**[][][]** a **=** **new** int**[**10**][**10**][**10**];**

**for(**int**[][]** x **:** a**)** **for(**int**[]** y **:** x**)** Arrays**.**fill**(**y**,** **-**1**);**

Methods:

//take two integer a, b, return the greatest common divider

static int gcd**(**int a**,** int b**)** **{**

**if(**b **==** 0**)** **return** a**;**

**return** gcd**(**b**,** a **%** b**);**

**}**

//take two points on a plane, return the distance between two points

static double dist**(**double x1**,** double y1**,** double x2**,** double y2**)** **{**

**return** Math**.**sqrt**(**Math**.**pow**(**x1 **-** x2**,** 2**)** **+** Math**.**pow**(**y1 **-** y2**,** 2**));**

**}**

**Bitwise operations**

commonly used in problems include bitmasks.

Some problems may need to record the states. (TSP) This is where bitwise operations are useful. You can use a single integer to record 232 states instead of a boolean array.

public class BitOperation **{**

public static void main**(**String**[]** args**)** **{**

/\*

\* &: bitwise and

\* |: bitwise or

\* ^: bitwise XOR

\* ~: bitwise not

\* <</>>/>>>: bit shifting

\*/

//diffrence between >> and >>>

int x **=** **-**2**;** // 11111.....1110

int x1 **=** x **>>** 1**;** // 11111.....1111 fill the leading bits with sign bit

int x2 **=** x **>>>** 1**;** // 01111.....1111 fill the leading bits with 0s

int a **=** 127**;** // 000...001111111

int n **=** 3**;**

/\*

\* check if the n-th bit is turned on

\* if ans is 0, then n-th bit is turned off

\* otherwise n-th bit is turned on

\*/

int ans **=** a **&** **(**1 **<<** n**);**

// turn off the n-th bit

a **&=** **~(**1 **<<** n**);** // a: 000...001110111

// turn on the n-th bit

a **|=** **(**1 **<<** n**);** // a: 000...001111111

//flip the n-th bit

a **^=** **(**1 **<<** n**);** // a: 000...001110111

//get the value of least significant bit that is on

a **=** 40**;** // a: 000...000101000

ans **=** a **&** **(-**a**);** // ans: 000...000001000

//turn on all bits in a set of size n

ans **=** **(**1 **<<** n**)** **-** 1**;** //ans: 000...0000000111

//count the number of bits that is on. aka the number of 1s

int num **=** 0xF0F0F0F0**;** //16 bits are on

int count **=** Integer**.**bitCount**(**num**);** //16

**}**

**}**

**Non-Built-In Data Structure**

**Segment Tree (RMQ, RSQ)**

Solve problems that have a lot of range minimum/maximum/sum query.

e.g. Give a list of number {4, 3, 5, 2, 3, 6, 1}, what is the maximum number from index 2 to 5.

Following code include RMQ with lazy update, if the number list is static, then ignore the lazy update code.

This code is for Range Minimum Query. Need a little change to apply for Range Maximum Query and Range Sum Query.

**import** java**.**util**.\*;**

//Ranged Minimum/Maximum/Sum Query problem (RMQ/RSQ)

public class SegmentTree **{**

private int**[]** st**,** input**,** lazy**;** //lazy is for lazy propagation

public SegmentTree**(**int**[]** in**)** **{**

**this.**input **=** in**;**

**this.**st **=** **new** int**[**input**.**length **\*** 4**];**

**this.**lazy **=** **new** int**[**st**.**length**];** //this is only for lazy propagation

//fill with Integer.MIN\_VALUE if is range maximum query, 0 if RSQ

Arrays**.**fill**(**st**,** Integer**.**MAX\_VALUE**);**

construct**(**0**,** input**.**length **-** 1**,** 0**);**

**}**

public int rmq**(**int low**,** int high**)** **{**

**return** rmq**(**low**,** high**,** 0**,** input**.**length **-** 1**,** 0**);**

**}**

private void construct**(**int low**,** int high**,** int pos**)** **{**

**if(**low **==** high**)** **{**

st**[**pos**]** **=** input**[**low**];**

**return;**

**}**

int mid **=** **(**low **+** high**)** **/** 2**;**

construct**(**low**,** mid**,** pos **\*** 2 **+** 1**);**

construct**(**mid **+** 1**,** high**,** pos **\*** 2 **+** 2**);**

//use Math.max if is range maximum query, add those two if is RSQ

st**[**pos**]** **=** Math**.**min**(**st**[**pos **\*** 2 **+** 1**],** st**[**pos **\*** 2 **+** 2**]);**

**}**

private int rmq**(**int qlow**,** int qhigh**,** int low**,** int high**,** int pos**)** **{**

**if(**qlow **>** high **||** qhigh **<** low**)**

**return** Integer**.**MAX\_VALUE**;** //return Integer.MIN\_VALUE if is range maximum query, return 0 if is rsq

**if(**qlow **<=** low **&&** qhigh **>=** high**)**

**return** st**[**pos**];**

int mid **=** **(**low **+** high**)** **/** 2**;**

//use Math.max if is range maximum query

//return sum if is RSQ

**return** Math**.**min**(**rmq**(**qlow**,** qhigh**,** low**,** mid**,** pos **\*** 2 **+** 1**),**

rmq**(**qlow**,** qhigh**,** mid **+** 1**,** high**,** pos**\*** 2 **+** 2**));**

**}**

/\*

\* This is for updating one value of the data

\*/

private void updatePoint**(**int index**,** int delta**,** int low**,** int high**,** int pos**)** **{**

**if(**index **==** low **&&** index **==** high**)** **{**

input**[**index**]** **+=** delta**;**

st**[**pos**]** **=** input**[**index**];**

**}**

**if(**index **>** high **||** index **<** low**)** **return;**

int mid **=** **(**low **+** high**)** **/** 2**;**

updatePoint**(**index**,** delta**,** low**,** mid**,** pos **\*** 2 **+** 1**);**

updatePoint**(**index**,** delta**,** mid **+** 1**,** high**,** pos **\*** 2 **+** 2**);**

//use max if is maximum query, sum if is RSQ

st**[**pos**]** **=** Math**.**min**(**st**[**pos **\*** 2 **+** 1**],** st**[**pos **\*** 2 **+** 2**]);**

**}**

public void updatePoint**(**int index**,** int delta**)** **{**

updatePoint**(**index**,** delta**,** 0**,** input**.**length **-** 1**,** 0**);**

**}**

/\*

\* previous code is standard rmq segment tree

\*

\* next is lazy updates (used when the problem requires dynamic updates)

\*/

//update data from sPos to ePos by delta

private void updateRange**(**int sPos**,** int ePos**,** int delta**,** int low**,** int high**,** int pos**)** **{**

**if(**low **>** high**)**

**return;**

**if(**lazy**[**pos**]** **!=** 0**)** **{**

st**[**pos**]** **+=** lazy**[**pos**];** // += lazy[pos] \* (high - low + 1); if is rsq

**if(**low **!=** high**)** **{**

lazy**[**pos **\*** 2 **+** 1**]** **+=** lazy**[**pos**];**

lazy**[**pos **\*** 2 **+** 2**]** **+=** lazy**[**pos**];**

**}**

lazy**[**pos**]** **=** 0**;**

**}**

**if(**sPos **>** high **||** ePos **<** low**)**

**return;**

**if(**sPos **<=** low **&&** ePos **>=** high**)** **{**

st**[**pos**]** **+=** delta**;** // += delta \* (high - low + 1); if is rsq

**if(**low **!=** high**)** **{**

lazy**[**pos **\*** 2 **+** 1**]** **+=** delta**;**

lazy**[**pos **\*** 2 **+** 2**]** **+=** delta**;**

**}**

**return;**

**}**

int mid **=** **(**low **+** high**)** **/** 2**;**

updateRange**(**sPos**,** ePos**,** delta**,** low**,** mid**,** pos **\*** 2 **+** 1**);**

updateRange**(**sPos**,** ePos**,** delta**,** mid **+** 1**,** high**,** pos **\*** 2 **+** 2**);**

//use Math.max if is maximum query, take the sum if is rsq

st**[**pos**]** **=** Math**.**min**(**st**[**pos **\*** 2 **+** 1**],** st**[**pos **\*** 2 **+** 2**]);**

**}**

public void updateRange**(**int sPos**,** int ePos**,** int delta**)** **{**

updateRange**(**sPos**,** ePos**,** delta**,** 0**,** input**.**length **-** 1**,** 0**);**

**}**

//lazy version of RMQ

private int rmqLazy**(**int qLow**,** int qHigh**,** int low**,** int high**,** int pos**)** **{**

**if(**low **>** high**)**

**return** Integer**.**MAX\_VALUE**;** //return MIN if maximum query, return 0 if is rsq

**if(**lazy**[**pos**]** **!=** 0**)** **{**

st**[**pos**]** **+=** lazy**[**pos**];** // += lazy[pos] \* (high - low + 1); if is rsq

**if(**low **!=** high**)** **{**

lazy**[**pos **\*** 2 **+** 1**]** **+=** lazy**[**pos**];**

lazy**[**pos **\*** 2 **+** 2**]** **+=** lazy**[**pos**];**

**}**

lazy**[**pos**]** **=** 0**;**

**}**

**if(**qLow **>** high **||** qHigh **<** low**)**

**return** Integer**.**MAX\_VALUE**;** //return MIN if maximum query, return 0 if is rsq

**if(**qLow **<=** low **&&** qHigh **>=** high**)**

**return** st**[**pos**];**

int mid **=** **(**low **+** high**)** **/** 2**;**

//return max if is maximum query, return sum if is rsq

**return** Math**.**min**(**rmqLazy**(**qLow**,** qHigh**,** low**,** mid**,** 2 **\*** pos **+** 1**),**

rmqLazy**(**qLow**,** qHigh**,** mid **+** 1**,** high**,** 2 **\*** pos **+** 2**));**

**}**

public int rmqLazy**(**int qLow**,** int qHigh**)** **{**

**return** rmqLazy**(**qLow**,** qHigh**,** 0**,** input**.**length **-** 1**,** 0**);**

**}**

**}**

**Segment Tree V2 (Shorter code with only single point update)**

static class SegTree **{**

int**[]** st**;**

int n**;**

public SegTree**(**int**[]** array**)** **{**

n **=** array**.**length**;**

st **=** **new** int**[**2 **\*** n**];**

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

st**[**n **+** i**]** **=** array**[**i**];**

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

st**[**i**]** **=** Math**.**min**(**st**[**i**<<**1**],** st**[**i**<<**1**|**1**]);**

**}**

//range minimum query in range [l, r). NOT including r

public int rmq**(**int l**,** int r**)** **{**

int min **=** Integer**.**MAX\_VALUE**;**

**for(**l **+=** n**,** r **+=** n**;** l **<** r**;** l **>>=** 1**,** r **>>=** 1**)** **{**

**if((**l **&** 1**)** **==** 1**)** min **=** Math**.**min**(**min**,** st**[**l**++]);**

**if((**r **&** 1**)** **==** 1**)** min **=** Math**.**min**(**st**[--**r**],** min**);**

**}**

**return** min**;**

**}**

public void update**(**int p**,** int c**)** **{**

st**[**p **+=** n**]** **=** c**;**

**for(**p **>>=** 1**;** p **>** 0**;** p **>>=** 1**)**

st**[**p**]** **=** Math**.**min**(**st**[**p **<<** 1**],** st**[**p **<<** 1 **|** 1**]);**

**}**

**}**

public static void main**(**String**[]** args**)** **{**

SegTree st **=** **new** SegTree**(new** int**[]** **{**6**,** **-**3**,** 4**,** 6**,** 2**,** **-**1**,** 10**});**

st**.**update**(**2**,** 1**);** // {6, -3, 1, 6, 2, -1, 10}

System**.**out**.**println**(**st**.**rmq**(**2**,** 5**));** // will output 1

**}**

**Segment Tree V3 (Maximum Subvector Sum)**

Giving a list of numbers, what is the maximum subvector sum in range [a, b].

For example, array [-1, 3, -4, 3, 1, -3, 2], the maximum subvector sum in range [1, 6] is 4 (3 + 1).

// this struct contains 4 things:

// sum, max prefix sum, max suffix sum, and max subvector sum

static class X **{**

long sum**,** prefix**,** suffix**,** subvec**;**

X**(**long a**,** long b**,** long c**,** long d**)** **{**

sum **=** a**;** prefix **=** b**;** suffix **=** c**;** subvec **=** d**;**

**}**

X**(**long a**)** **{** sum **=** prefix **=** suffix **=** subvec **=** a**;** **}**

**}**

static X add**(**X x1**,** X x2**)** **{**

**return** **new** X**(**x1**.**sum **+** x2**.**sum**,**

Math**.**max**(**x1**.**prefix**,** x1**.**sum **+** x2**.**prefix**),**

Math**.**max**(**x2**.**suffix**,** x2**.**sum **+** x1**.**suffix**),**

Math**.**max**(**x1**.**suffix **+** x2**.**prefix**,** Math**.**max**(**x1**.**subvec**,** x2**.**subvec**)));**

**}**

static class SegmentTree **{**

X**[]** st**;**

int n**;**

SegmentTree**(**long**[]** input**)** **{**

n **=** input**.**length**;**

st **=** **new** X**[**2 **\*** n**];**

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

st**[**n **+** i**]** **=** **new** X**(**input**[**i**]);**

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

st**[**i**]** **=** add**(**st**[**i **<<** 1**],** st**[**i **<<** 1 **|** 1**]);**

**}**

// find out the max subvector in range [l, r). NOT including r

X query**(**int l**,** int r**)** **{**

X retl **=** **new** X**(**0**),** retr **=** **new** X**(**0**);**

**for(**l **+=** n**,** r **+=** n**;** l **<** r**;** l **>>=** 1**,** r **>>=** 1**)** **{**

//note that add(a,b) is NOT equals to add(b,a)

**if((**l **&** 1**)** **==** 1**)** retl **=** add**(**retl**,** st**[**l**++]);** //order matters!

**if((**r **&** 1**)** **==** 1**)** retr **=** add**(**st**[--**r**],** retr**);** //order matters!

**}**

**return** add**(**retl**,** retr**);** //order matters

**}**

void update**(**int p**,** long v**)** **{**

st**[**p **+=** n**]** **=** **new** X**(**v**);**

**for(**p **>>=** 1**;** p **>** 0**;** p **>>=** 1**)**

st**[**p**]** **=** add**(**st**[**p **<<** 1**],** st**[**p **<<** 1 **|** 1**]);**

**}**

**}**

public static void main**(**String**[]** args**)** **{**

SegmentTree st **=** **new** SegmentTree**(new** long**[]** **{** **-**1**,** 3**,** **-**4**,** 3**,** 1**,** **-**3**,** 2 **}** **);**

st**.**update**(**5**,** 0**);** // { -1, 3, -4, 3, 1, 0, 2 }

System**.**out**.**println**(**st**.**query**(**1**,** 7**).**subvec**);** //will output 6 (3 + 1 + 0 + 2)

**}**

**Union-Find Disjoint Set**

This type of problems involving commands like union two sets multiple times, and then have multiple queries about the size of the set, total number of sets, or are two elements in the same set. (If there is only one query, then DFS is enough: by looking each element as a vertex, then union two element is simply add an edge between two vertices. This is still slower then Union-Find data structure, but easier to implement)

**import** java**.**util**.\*;**

//union-find disjoint set

public class UnionFind **{**

static class UF **{**

private Vector**<**Integer**>** p**,** rank**,** setSize**;**

private int numSets**;**

public UF**(**int N**)** **{**

p **=** **new** Vector**<**Integer**>(**N**);**

rank **=** **new** Vector**<**Integer**>(**N**);**

setSize **=** **new** Vector**<**Integer**>(**N**);**

numSets **=** N**;**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

p**.**add**(**i**);**

rank**.**add**(**0**);**

setSize**.**add**(**1**);**

**}**

**}**

public int findSet**(**int i**)** **{**

**if** **(**p**.**get**(**i**)** **==** i**) return** i**;**

**else** **{**

int ret **=** findSet**(**p**.**get**(**i**));**

p**.**set**(**i**,** ret**);**

**return** ret**;**

**}**

**}**

// return true if element i and j are in the same set, return false otherwise

public Boolean isSameSet**(**int i**,** int j**)** **{**

**return** findSet**(**i**)** **==** findSet**(**j**);**

**}**

// union two sets that contain element i and j

public void unionSet**(**int i**,** int j**)** **{**

**if** **(!**isSameSet**(**i**,** j**))** **{**

numSets**--;**

int x **=** findSet**(**i**),** y **=** findSet**(**j**);**

// rank is used to keep the tree short

**if** **(**rank**.**get**(**x**)** **>** rank**.**get**(**y**))** **{**

p**.**set**(**y**,** x**);**

setSize**.**set**(**x**,** setSize**.**get**(**x**)** **+** setSize**.**get**(**y**));**

**}** **else** **{**

p**.**set**(**x**,** y**);**

setSize**.**set**(**y**,** setSize**.**get**(**y**)** **+** setSize**.**get**(**x**));**

**if** **(**rank**.**get**(**x**)** **==** rank**.**get**(**y**))**

rank**.**set**(**y**,** rank**.**get**(**y**)** **+** 1**);**

**}**

**}**

**}**

// return number of disjoint sets

public int numDisjointSets**()** **{**

**return** numSets**;**

**}**

// return the size of the set that contain element i

public int sizeOfSet**(**int i**)** **{**

**return** setSize**.**get**(**findSet**(**i**));**

**}**

**}**

**}**

**Binary Index Tree (BIT)**

If you only looking for Range Sum Query, this is a shorter algorithm for it.

**import** java**.**util**.\*;**

public class BinaryIndexTree **{**

private Vector**<**Integer**>** ft**;**

private int LSOne**(**int S**)** **{**

**return** **(**S **&** **(-**S**));**

**}**

public BinaryIndexTree**()** **{}**

// initialization: n + 1 zeroes, ignore index 0

public BinaryIndexTree**(**int n**)** **{**

ft **=** **new** Vector**<**Integer**>();**

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

ft**.**add**(**0**);**

**}**

public int rsq**(**int b**)** **{** // returns RSQ(1, b)

int sum **=** 0**;**

**for** **(;** b **>** 0**;** b **-=** LSOne**(**b**))**

sum **+=** ft**.**get**(**b**);**

**return** sum**;**

**}**

public int rsq**(**int a**,** int b**)** **{** // returns RSQ(a, b)

**return** rsq**(**b**)** **-** **(**a **==** 1 **?** 0 **:** rsq**(**a **-** 1**));**

**}**

// adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)

void adjust**(**int k**,** int v**)** **{** // note: n = ft.size() - 1

**for** **(;** k **<** **(**int**)** ft**.**size**();** k **+=** LSOne**(**k**))**

ft**.**set**(**k**,** ft**.**get**(**k**)** **+** v**);**

**}**

**}**

**Dynamic Programming**

**Longest Increasing Subsequence (LIS)**

Finding the length of longest increasing subsequence or the subsequence itself.

e.g. given a list {3, 8, 4, 6, 2, 9} the length of LIS is 4, which is {3, 4, 6, 9}

There are two approaches, the first one is slower, O(n2) time complexity.

public static int LIS**(**int**[]** array**)** **{**

int**[]** dp **=** **new** int**[**array**.**length**];**

int max **=** 0**;**

**for(**int i **=** 0**;** i **<** array**.**length**;** i**++)** **{**

**for(**int j **=** i **+** 1**;** j **<** array**.**length**;** j**++)** **{**

**if(**array**[**j**]** **>** array**[**i**])** **{**

dp**[**j**]** **=** Math**.**max**(**dp**[**j**],** dp**[**i**]** **+** 1**);**

**}**

**}**

max **=** Math**.**max**(**max**,** dp**[**i**]);**

**}**

**return** max **+** 1**;**

**}**

The second approach is faster, O(nlogn) time complexity. This approach also have the ability to return the actual subsequence.

public static int**[]** LISFaster**(**int**[]** array**)** **{**

int n **=** array**.**length**;**

int**[]** lid **=** **new** int**[**n**];**

int**[]** p **=** **new** int**[**n**];**

List**<**Integer**>** l **=** **new** ArrayList**<>();**

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

int pos **=** Collections**.**binarySearch**(**l**,** array**[**i**]);**

**if(**pos **<** 0**)** pos **=** **-(**pos **+** 1**);**

**if(**pos **>=** l**.**size**())** l**.**add**(**array**[**i**]);**

**else** l**.**set**(**pos**,** array**[**i**]);**

lid**[**pos**]** **=** i**;**

p**[**i**]** **=** pos **>** 0 **?** lid**[**pos **-** 1**]** **:** **-**1**;**

**}**

//l.size() is the length of LIS, directly return it if only need the length

//following is constructing the LIS.

int**[]** lis **=** **new** int**[**l**.**size**()];**

int i **=** lid**[**l**.**size**()** **-** 1**];**

**for(**int index **=** l**.**size**()** **-** 1**;** p**[**i**]** **!=** **-**1**;** i **=** p**[**i**],** index**--)**

lis**[**index**]** **=** array**[**i**];**

lis**[**0**]** **=** array**[**i**];**

**return** lis**;**

**}**

Same approach, an alternative way to write it following. (This only return the length of the LIS)

public static int LISFaster**(**double**[]** array**)** **{**

double**[]** dp **=** **new** double**[**array**.**length**];**

dp**[**0**]** **=** array**[**0**];**

int ans **=** 1**;**

**for(**int i **=** 1**;** i **<** array**.**length**;** i**++)** **{**

int low **=** 0**,** high **=** ans **-** 1**;**

**while(**low **<=** high**)** **{**

int mid **=** **(**low **+** high**)** **/** 2**;**

**if(**dp**[**mid**]** **<** array**[**i**])** low **=** mid **+** 1**;**

**else** high **=** mid **-** 1**;**

**}**

dp**[**low**]** **=** array**[**i**];**

ans **=** Math**.**max**(**ans**,** low **+** 1**);**

**}**

**return** ans**;**

**}**

**Maximum Range Sum (MRS)**

Finding the maximum range sum in an array.

e.g. given a list {4, 2, -7, 4, -3, 1}. The max range sum is 6. (from index 0 to 1)

MRS for 1D array is O(n), for 2D array is O(n4)

public static int max1DRangeSum**(**int**[]** a**)** **{**

int sum **=** 0**,** ans **=** 0**;**

**for(**int i **=** 0**;** i **<** a**.**length**;** i**++)** **{**

sum **+=** a**[**i**];**

ans **=** Math**.**max**(**ans**,** sum**);**

**if(**sum **<** 0**)** sum **=** 0**;**

**}**

**return** ans**;**

**}**

public static int max2DRangeSum**(**int**[][]** a**)** **{**

**for(**int i **=** 0**;** i **<** a**.**length**;** i**++)** **for(**int j **=** 0**;** j **<** a**[**i**].**length**;** j**++)** **{**

**if(**i **>** 0**)** a**[**i**][**j**]** **+=** a**[**i **-** 1**][**j**];**

**if(**j **>** 0**)** a**[**i**][**j**]** **+=** a**[**i**][**j **-** 1**];**

**if(**i **>** 0 **&&** j **>** 0**)** a**[**i**][**j**]** **-=** a**[**i **-** 1**][**j **-** 1**];**

**}**

int max **=** Integer**.**MIN\_VALUE**;**

**for(**int i **=** 0**;** i **<** a**.**length**;** i**++)** **for(**int j **=** 0**;** j **<** a**[**i**].**length**;** j**++)** **{**

**for(**int k **=** i**;** k **<** a**.**length**;** k**++)** **for(**int l **=** j**;** l **<** a**[**k**].**length**;** l**++)** **{**

int subRect **=** a**[**k**][**l**];**

**if(**i **>** 0**)** subRect **-=** a**[**i **-** 1**][**l**];**

**if(**j **>** 0**)** subRect **-=** a**[**k**][**j **-** 1**];**

**if(**i **>** 0 **&&** j **>** 8**)** subRect **+=** a**[**i **-** 1**][**j **-** 1**];**

max **=** Math**.**max**(**max**,** subRect**);**

**}**

**}**

**return** max**;**

**}**

**0-1Knapsack**

Given a set of items with values and weights, you want to maximize the total value but the total weight must be less than or equal to the threshold.

e.g. suppose there are 4 items. value = {100, 70, 50, 10}, weight = {10, 4, 6, 12}, the bag size is 12. The maximum value you can get is 120, by taking the second and the third item.

This DP approach is O(nS), where n is the number of items, S is the size of the bag.

**import** java**.**util**.**Arrays**;**

public class Knapsack01 **{**

static int**[]** value **=** **new** int**[]{**100**,** 70**,** 50**,** 10**};**

static int**[]** weight **=** **new** int**[]{**10**,** 4**,** 6**,** 12**};**

static int**[][]** memo**;**

public static void main**(**String**[]** args**)** **{**

int size **=** 12**;** //Bag size

//bottom-up approach

int len **=** value**.**length**;**

int**[][]** dp **=** **new** int**[**len**][**size **+** 1**];**

**for(**int i **=** 0**;** i **<** len**;** i**++)** dp**[**i**][**0**]** **=** 0**;**

**for(**int i **=** 0**;** i **<=** size**;** i**++)** dp**[**0**][**i**]** **=** 0**;**

**for(**int i **=** 1**;** i **<** len**;** i**++)** **{**

**for(**int j **=** 1**;** j **<=** size**;** j**++)** **{**

**if(**weight**[**i**]** **>** j**)** dp**[**i**][**j**]** **=** dp**[**i **-** 1**][**j**];**

**else** dp**[**i**][**j**]** **=** Math**.**max**(**dp**[**i **-** 1**][**j**],**

value**[**i**]** **+** dp**[**i **-** 1**][**j **-** weight**[**i**]]);**

**}**

**}**

System**.**out**.**println**(**dp**[**len **-** 1**][**size**]);**

//bottom up done. Following is top-down approach.

//Note that these two are separate approaches, only need to implement one of them

//top-down approach

memo **=** **new** int**[**value**.**length**][**size **+** 1**];**

**for(**int i **=** 0**;** i **<** memo**.**length**;** i**++)**

Arrays**.**fill**(**memo**[**i**],** **-**1**);**

System**.**out**.**println**(**val**(**0**,** size**));**

**}**

//top-down

static int val**(**int id**,** int remSize**)** **{**

**if(**id **==** value**.**length **||** remSize **==** 0**)** **return** 0**;**

**if(**memo**[**id**][**remSize**]** **!=** **-**1**)** **return** memo**[**id**][**remSize**];**

**if(**weight**[**id**]** **>** remSize**)** **return** memo**[**id**][**remSize**]** **=** val**(**id **+** 1**,** remSize**);**

**return** memo**[**id**][**remSize**]** **=** Math**.**max**(**val**(**id **+** 1**,** remSize**),**

value**[**id**]** **+** val**(**id **+** 1**,** remSize **-** weight**[**id**]));**

**}**

**}**

**Coin Change (CC)**

Given a set of coin values and the target amount V, what is the minimum number of coins that we must use to represent V? Assume we have infinite amount of coins for each value.

e.g. coinValue = {1, 5, 10, 15}, V = 12. Well we need 3 coins: one 10 and two 1’s.

Complexity O(nV), where n is the number of coin values, V is the target value.

There are similar problems like how many ways are there to achieve the target value. What is the minimum number of coins that we must use to represent a target value if there are limited amount of coins? (Both included in following code)

**import** java**.**util**.**Arrays**;**

public class CoinChange **{**

// 2 is size of coinValue, 100 is the target value

static int**[][]** ways **=** **new** int**[**2**][**101**];**

static int**[]** coinValue **=** **{**1**,** 5**};**

static int N **=** 2**;** // size of coinValue

public static void main**(**String**[]** args**)** **{**

//min CC, unlimited amount of coins

double**[]** dp **=** **new** double**[**11**];**

Arrays**.**fill**(**dp**,** Double**.**POSITIVE\_INFINITY**);**

dp**[**0**]** **=** 0**;**

**for(**int i **=** 0**;** i **<** dp**.**length**;** i**++)** **{**

**for(**int j **=** 0**;** j **<** N**;** j**++)** **{**

**if(**i **+** coinValue**[**j**]** **<** dp**.**length**)**

dp**[**i **+** coinValue**[**j**]]** **=** Math**.**min**(**dp**[**i **+** coinValue**[**j**]],** dp**[**i**]** **+** 1**);**

**}**

**}**

//Ways: number of ways to get total value 100 with different coins

//Ways top-down example

**for(**int**[]** a **:** ways**)** Arrays**.**fill**(**a**,** **-**1**);**

System**.**out**.**println**(**way**(**0**,** 100**));**

//Ways bottom-up example

int**[]** dp1 **=** way**(**100**);**

System**.**out**.**println**(**dp1**[**100**]);**

**}**

//Ways, top-down, may stack overflow if value is too large,

//then use bottom-up approach

public static int way**(**int type**,** int value**)** **{**

**if(**value **==** 0**)** **return** 1**;**

**if(**value **<** 0 **||** type **==** N**)** **return** 0**;**

**if(**ways**[**type**][**value**]** **!=** **-**1**)** **return** ways**[**type**][**value**];**

**return** ways**[**type**][**value**]** **=** way**(**type **+** 1**,** value**)** **+** way**(**type**,** value **-**

coinValue**[**type**]);**

**}**

/\* bottom-up, build the table only once!, let value be the largest possible value.

\* once get the array, use dp[value] to get the ways

\* May use long[] if the value and the size of the coin value is large.

\* complexity nV, n is the value, V is the size of coinValue

\*/

public static int**[]** way**(**int value**)** **{**

int**[]** dp **=** **new** int**[**value **+** 1**];**

dp**[**0**]** **=** 1**;**

**for(**int i **=** 0**;** i **<** N**;** i**++)** **for(**int j **=** coinValue**[**i**];** j **<=** value**;** j**++)**

dp**[**j**]** **+=** dp**[**j **-** coinValue**[**i**]];**

**return** dp**;**

**}**

// Coin Change with limited amount of coins

public static void cc**()** **{**

int V **=** 11**;** // V is the target amount

int**[]** coins **=** **{**1**,** 1**,** 2**,** 3**,** 4**};** //coin values. Two 1s, one 2, one 3, and one 4

int**[]** dp **=** **new** int**[**V **+** 1**];**

Arrays**.**fill**(**dp**,** Integer**.**MAX\_VALUE**);**

dp**[**0**]** **=** 0**;**

**for(**int c **:** coins**)** **for(**int i **=** dp**.**length **-** 1 **-** c**;** i **>=** 0**;** i**--)**

**if(**dp**[**i**]** **<** Integer**.**MAX\_VALUE**)** dp**[**i **+** c**]** **=** Math**.**min**(**dp**[**i **+** c**],** dp**[**i**]** **+** 1**);**

dp**[**V**];** // is the minimum number of coins we must use to get V

**}**

**}**

**Travelling Salesman Problem (TSP)**

Given n cities and their pairwise distances in the form of a matrix dist of size n\*n, compute the cost of making a tour that starts from any city s, go through all the other n – 1 cities exactly once, and return s to the starting city s.

**import** java**.**util**.**Arrays**;**

public class TravelingSalesmanProblem **{**

//TSP

//Won't TLE only if the input size is <= 16

static int**[][]** dp **=** **new** int**[**4**][**1 **<<** 4**];** //4 is the number of vertex

static int**[][]** graph **=** **new** int**[][]{{**0**,** 20**,** 42**,** 35**},**

**{**20**,** 0**,** 30**,** 34**},**

**{**42**,** 30**,** 0**,** 12**},**

**{**35**,** 34**,** 12**,** 0**}};**

public static void main**(**String**[]** args**)** **{**

**for(**int**[]** a **:** dp**)** Arrays**.**fill**(**a**,** **-**1**);**

System**.**out**.**println**(**tsp**(**0**,** 1**));** //always begin with 0, 1

**}**

public static int tsp**(**int pos**,** int mask**)** **{**

**if(**mask **==** **(**1 **<<** graph**.**length**)** **-** 1**)** **return** graph**[**pos**][**0**];**

**if(**dp**[**pos**][**mask**]** **!=** **-**1**)** **return** dp**[**pos**][**mask**];**

int min **=** Integer**.**MAX\_VALUE**;**

**for(**int i **=** 0**;** i **<** graph**.**length**;** i**++)**

**if(**i **!=** pos **&&** **(**mask **&** **(**1 **<<** i**))** **==** 0**)**

min **=** Math**.**min**(**min**,** graph**[**pos**][**i**]** **+** tsp**(**i**,** mask **|** **(**1 **<<** i**)));**

**return** dp**[**pos**][**mask**]** **=** min**;**

**}**

**}**

**Graph**

**Depth First Search (DFS) / Breadth First Search (BFS)**

Graph traverse

**import** java**.**util**.\*;**

public class DFS **{**

static class Node **{**

int id**;**

List**<**Node**>** adjacent **=** **new** ArrayList**<>();**

Node**(**int id**)** **{this.**id **=** id**;}**

**}**

static boolean**[]** visited**;**

public static void main**(**String**[]** args**)** **{**

int N **=** 4**;** // 4 nodes;

List**<**Node**>** nodes **=** **new** ArrayList**<>();**

**for(**int i **=** 0**;** i **<** N**;** i**++)** nodes**.**add**(new** Node**(**i**));**

nodes**.**get**(**0**).**adjacent**.**add**(**nodes**.**get**(**1**));**

nodes**.**get**(**1**).**adjacent**.**add**(**nodes**.**get**(**0**));** // 0-1 connected

nodes**.**get**(**0**).**adjacent**.**add**(**nodes**.**get**(**2**));**

nodes**.**get**(**2**).**adjacent**.**add**(**nodes**.**get**(**0**));** // 0-2 connected

nodes**.**get**(**2**).**adjacent**.**add**(**nodes**.**get**(**1**));**

nodes**.**get**(**1**).**adjacent**.**add**(**nodes**.**get**(**2**));** // 2-1 connected

nodes**.**get**(**0**).**adjacent**.**add**(**nodes**.**get**(**3**));**

nodes**.**get**(**3**).**adjacent**.**add**(**nodes**.**get**(**0**));** // 0-3 connected

visited **=** **new** boolean**[**N**];**

**}**

public static void dfs**(**Node n**)** **{**

visited**[**n**.**id**]** **=** **true;**

**for(**Node adj **:** n**.**adjacent**)**

**if(!**visited**[**adj**.**id**])**

dfs**(**adj**);**

**}**

public static void bfs**(**Node n**)** **{**

visited**[**n**.**id**]** **=** **true;**

Queue**<**Node**>** q **=** **new** LinkedList**<>();**

q**.**add**(**n**);**

**while(!**q**.**isEmpty**())** **{**

Node u **=** q**.**poll**();**

**for(**Node adj **:** n**.**adjacent**)** **{**

**if(!**visited**[**adj**.**id**])** **{**

q**.**add**(**adj**);**

visited**[**adj**.**id**]** **=** **true;**

**}**

**}**

**}**

**}**

**}**

**Tree Diameter**

Given a tree (has to be a tree), find out the diameter of the tree. aka the longest simple path in a tree.

//O(n)

//acyclic graph

static List**<**List**<**Integer**>>** adj **=** **new** ArrayList**<>();**

static int mx**,** mxn**;**

static void dfs**(**int n**,** int p**,** int l**)** **{**

**if(**l **>** mx**)** **{**

mx **=** l**;**

mxn **=** n**;**

**}**

**for(**int v **:** adj**.**get**(**n**))**

dfs**(**v**,** n**,** l **+** 1**);**

**}**

public static void main**(**String**[]** args**)** **{**

//some graph setup

// .........

//store the graph in adj: adjacency list

dfs**(**0**,** 0**,** 0**);**

mx **=** 0**;**

dfs**(**mxn**,** mxn**,** 0**);**

//diameter now stored in mx

**}**

**Bridge**

A bridge in a graph is an edge that if the edge was removed, the graph will be disconnected.

//store graph as Nodes with adjacent list

static class Node **{**

int id**;**

List**<**Node**>** adjacent **=** **new** ArrayList**<>();**

Node**(**int dd**)** **{** id **=** dd**;** **}**

**}**

static List**<**Node**>** nodes**;**

static int**[]** pre**,** low**;**

static int bridgeNum**;** // store number of bridges

static int cnt**;**

static void findBridge**()** **{**

low **=** **new** int**[**nodes**.**size**()];**

pre **=** **new** int**[**nodes**.**size**()];**

Arrays**.**fill**(**low**,** **-**1**);**

Arrays**.**fill**(**pre**,** **-**1**);**

**for(**Node v **:** nodes**)**

**if(**pre**[**v**.**id**]** **==** **-**1**)**

dfs**(**v**,** v**);**

**}**

static void dfs**(**Node u**,** Node v**)** **{**

pre**[**v**.**id**]** **=** cnt**++;**

low**[**v**.**id**]** **=** pre**[**v**.**id**];**

**for(**Node w **:** v**.**adjacent**)** **{**

**if(**pre**[**w**.**id**]** **==** **-**1**)** **{**

dfs**(**v**,** w**);**

low**[**v**.**id**]** **=** Math**.**min**(**low**[**v**.**id**],** low**[**w**.**id**]);**

**if(**low**[**w**.**id**]** **==** pre**[**w**.**id**])** **{**

//bridge found! v-w edge is a bridge

System**.**out**.**println**(**v**.**id **+** "-" **+** w**.**id**);**

bridgeNum**++;**

**}**

**}** **else** **if(**w**.**id **!=** u**.**id**)** **{**

low**[**v**.**id**]** **=** Math**.**min**(**low**[**v**.**id**],** pre**[**w**.**id**]);**

**}**

**}**

**}**

public static void main**(**String**[]** args**)** **{**

//example

nodes **=** **new** ArrayList**<>();**

**for(**int i **=** 0**;** i **<** 7**;** i**++)** nodes**.**add**(new** Node**(**i**));**

int**[][]** graph **=** **{{**1**,** 2**},** /\* \*/

**{**0**,** 2**},** /\* 0 4 \*/

**{**0**,** 1**,** 3**},** /\* / \ / \ \*/

**{**2**,** 4**,** 5**},** /\* / \ / \ \*/

**{**3**,** 5**},** /\* 1 --- 2 --- 3 --- 5 --- 6 \*/

**{**3**,** 4**,** 6**},** /\* \*/

**{**5**}};** /\* \*/

**for(**int i **=** 0**;** i **<** graph**.**length**;** i**++)** **{**

**for(**int j **=** 0**;** j **<** graph**[**i**].**length**;** j**++)** **{**

nodes**.**get**(**i**).**adjacent**.**add**(**nodes**.**get**(**graph**[**i**][**j**]));**

**}**

**}**

findBridge**();** //bridgeNum = 2, bridge: 2-3, 5-6

**}**

**Articulation Points**

Like bridge, an articulation point is a vertex that if it was removed, the graph will be disconnected.

//need some global variables

static boolean**[]** visited**;**

static int**[]** parent**,** lowTime**,** visitedTime**;**

static int time**;**

//graph - adjacent list

//return a set of articulation point

public static Set**<**Integer**>** findArticulationPoints**(**List**<**List**<**Integer**>>** graph**)** **{**

time **=** 0**;**

visited **=** **new** boolean**[**graph**.**size**()];**

parent **=** **new** int**[**graph**.**size**()];**

lowTime **=** **new** int**[**graph**.**size**()];**

visitedTime **=** **new** int**[**graph**.**size**()];**

Arrays**.**fill**(**parent**,** **-**1**);**

Set**<**Integer**>** artiPoint **=** **new** HashSet**<>();**

dfs**(**0**,** graph**,** artiPoint**);** // assume the graph initially is all connected

**return** artiPoint**;**

**}**

public static void dfs**(**int v**,** List**<**List**<**Integer**>>** graph**,** Set**<**Integer**>** artiPoint**)** **{**

visited**[**v**]** **=** **true;**

visitedTime**[**v**]** **=** time**;**

lowTime**[**v**]** **=** time**;**

time**++;**

int childCount **=** 0**;**

boolean isArtiPoint **=** **false;**

**for(**int adj **:** graph**.**get**(**v**))** **{**

**if(**adj **==** parent**[**v**])** **continue;**

**if(!**visited**[**adj**])** **{**

parent**[**adj**]** **=** v**;**

childCount**++;**

dfs**(**adj**,** graph**,** artiPoint**);**

**if(**visitedTime**[**v**]** **<=** lowTime**[**adj**])** isArtiPoint **=** **true;**

**else** lowTime**[**v**]** **=** Math**.**min**(**lowTime**[**v**],** lowTime**[**adj**]);**

**}** **else** **{**

lowTime**[**v**]** **=** Math**.**min**(**lowTime**[**v**],** lowTime**[**adj**]);**

**}**

**}**

**if((**parent**[**v**]** **==** **-**1 **&&** childCount **>=** 2**)** **||** **(**parent**[**v**]** **!=** **-**1 **&&** isArtiPoint**))**

artiPoint**.**add**(**v**);**

**}**

//example on next page ->

public static void main**(**String**[]** args**)** **{**

//example

int**[][]** adjList **=** **{{**1**,** 2**,** 3**},** /\*\*/

**{**0**,** 2**},** /\* \*/

**{**0**,** 1**},** /\* 1 6 \*/

**{**0**,** 4**},** /\* / \ / \ \*/

**{**3**,** 5**,** 6**},** /\* / \ / \ \*/

**{**4**,** 6**,** 7**},** /\* 2 --- 0 --- 3 --- 4 --- 5 --- 7 \*/

**{**4**,** 5**},** /\* \*/

**{**5**}};** /\*\*/

List**<**List**<**Integer**>>** adj **=** **new** ArrayList**<>();**

**for(**int i **=** 0**;** i **<** adjList**.**length**;** i**++)** adj**.**add**(new** ArrayList**<>());**

**for(**int i **=** 0**;** i **<** adjList**.**length**;** i**++)** **for(**int j **:** adjList**[**i**])**

adj**.**get**(**i**).**add**(**j**);**

Set**<**Integer**>** articulationPoints **=** findArticulationPoints**(**adj**);**

//output is {0, 3, 4, 5}

articulationPoints**.**forEach**(**p **->** System**.**out**.**print**(**p **+** " "**));**

**}**

**Topological Sort**

Topological sort only applies on direct acyclic graph (DAG). The algorithm will sort the vertices such that a child node will never appears before its parent. Note that there exist multiple results of topological sort depend on witch vertex you start with.

//take adjacent list as argument, return the sorted list

public static Deque**<**Integer**>** topSort**(**List**<**List**<**Integer**>>** adj**)** **{**

Deque**<**Integer**>** stack **=** **new** ArrayDeque**<>();**

Set**<**Integer**>** visited **=** **new** HashSet**<>();**

**for(**int i **=** 0**;** i **<** adj**.**size**();** i**++)** **{**

**if(!**visited**.**contains**(**i**))**

topSortUntil**(**i**,** stack**,** visited**,** adj**);**

**}**

**return** stack**;**

**}**

public static void topSortUntil**(**int v**,** Deque**<**Integer**>** stack**,**

Set**<**Integer**>** visited**,** List**<**List**<**Integer**>>** adj**)** **{**

visited**.**add**(**v**);**

**for(**int child **:** adj**.**get**(**v**))** **{**

**if(!**visited**.**contains**(**child**))**

topSortUntil**(**child**,** stack**,** visited**,** adj**);**

**}**

stack**.**offerFirst**(**v**);**

**}**

//example on next page ->

public static void main**(**String**[]** args**)** **{**

List**<**List**<**Integer**>>** adj **=** **new** ArrayList**<>();**

**for(**int i **=** 0**;** i **<** 8**;** i**++)** adj**.**add**(new** ArrayList**<>());**

adj**.**get**(**0**).**add**(**1**);**

adj**.**get**(**0**).**add**(**2**);**

adj**.**get**(**2**).**add**(**3**);**

adj**.**get**(**2**).**add**(**6**);**

adj**.**get**(**4**).**add**(**5**);**

adj**.**get**(**5**).**add**(**2**);**

adj**.**get**(**6**).**add**(**7**);**

Deque**<**Integer**>** result **=** topSort**(**adj**);**

//output - 4 5 0 2 6 7 3 1

**while(!**result**.**isEmpty**())** System**.**out**.**print**(**result**.**poll**()** **+** " "**);**

**}**

7

3

6

4

5

2

1

0

**Strongly Connected Component**

In a directed unweighted graph, two vertices A, B are in the same strongly connected component if there exist a path from A to B and there exist a path from B to A.

static List**<**List**<**Integer**>>** adj**;**

static int**[]** dfs\_num**,** dfs\_low**,** label**;**

static boolean**[]** visited**;**

static Stack**<**Integer**>** s**;**

static int dfsNumberCount**,** SCCCount**;**

public static void scc**(**int u**)** **{**

dfs\_low**[**u**]** **=** dfs\_num**[**u**]** **=** dfsNumberCount**++;**

s**.**push**(**u**);**

visited**[**u**]** **=** **true;**

**for(**int v **:** adj**.**get**(**u**))** **{**

**if(**dfs\_num**[**v**]** **==** **-**1**)** scc**(**v**);**

**if(**visited**[**v**])** dfs\_low**[**u**]** **=** Math**.**min**(**dfs\_low**[**u**],** dfs\_low**[**v**]);**

**}**

**if(**dfs\_low**[**u**]** **==** dfs\_num**[**u**])** **{**

SCCCount**++;**

**for(;;)** **{**

int v **=** s**.**pop**();**

visited**[**v**]** **=** **false;**

label**[**v**]** **=** SCCCount**;**

**if(**u **==** v**)** **break;**

**}**

**}**

**}**

public static void main**(**String**[]** args**)** **{**

//graph setup

int n **=** 7**;** // number of vertices

adj **=** **new** ArrayList**<>();**

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

4

1

adj**.**add**(new** ArrayList**<>());**

adj**.**get**(**0**).**add**(**1**);** adj**.**get**(**1**).**add**(**2**);**

adj**.**get**(**2**).**add**(**0**);** adj**.**get**(**2**).**add**(**3**);**

0

adj**.**get**(**3**).**add**(**4**);** adj**.**get**(**4**).**add**(**5**);**

66

5

3

2

adj**.**get**(**5**).**add**(**3**);** adj**.**get**(**6**).**add**(**5**);**

//initializing data.

label **=** **new** int**[**n**];**

In this case, Vertex 0, 1, 2 are in one strongly connected component and 3, 4, 5 are in another strongly connected component. The result of the algorithm is stored in “label” array, where index is the node id, value is which strongly connected component it belongs to.

dfs\_num **=** **new** int**[**n**];**

dfs\_low **=** **new** int**[**n**];**

visited **=** **new** boolean**[**n**];**

s **=** **new** Stack**<>();**

dfsNumberCount **=** SCCCount **=** 0**;**

Arrays**.**fill**(**dfs\_num**,** **-**1**);**

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

**if(**dfs\_num**[**i**]** **==** **-**1**)** scc**(**i**);**

System**.**out**.**println**(**Arrays**.**toString**(**label**));** //[2, 2, 2, 1, 1, 1, 3]

**}**

**Single Source Shortest Path (SSSP Dijkstra’s Algorithm)**

**import** java**.**util**.\*;**

public class Dijkstra **{**

public static class Vertex **implements** Comparable**<**Vertex**>** **{**

public final int name**;**

public ArrayList**<**Edge**>** adjacent**;**

public double minDis **=** Double**.**POSITIVE\_INFINITY**;**

public Vertex**(**int name**)** **{**

**this.**name **=** name**;**

adjacent **=** **new** ArrayList**<>();**

**}**

@Override

public int compareTo**(**Vertex other**)** **{**

**return** Double**.**compare**(this.**minDis**,** other**.**minDis**);**

**}**

**}**

public static class Edge **{**

public Vertex target**;**

public double weight**;**

public Edge**(**Vertex target**,** double weight**)** **{**

**this.**target **=** target**;**

**this.**weight **=** weight**;**

**}**

**}**

public static void dijkstra**(**Vertex source**,** Vertex dest**)** **{**

source**.**minDis **=** 0**;**

PriorityQueue**<**Vertex**>** vertexQueue **=** **new** PriorityQueue**<>();**

vertexQueue**.**add**(**source**);**

**while(!**vertexQueue**.**isEmpty**())** **{**

Vertex u **=** vertexQueue**.**poll**();**

**if(**u **==** dest**)**

**return;**

**for(**Edge e **:** u**.**adjacent**)** **{**

Vertex v **=** e**.**target**;**

double disThroughU **=** u**.**minDis **+** e**.**weight**;**

**if(**disThroughU **<** v**.**minDis**)** **{**

v**.**minDis **=** disThroughU**;**

vertexQueue**.**add**(**v**);**

**}**

**}**

**}**

**}**

**}**

**All Pair Shortest Path (APSP Floyd Warshall’s Algorithm)**

public static int**[][]** APSP**(**int**[][]** dist**)** **{**

int n **=** dist**.**length**;**

**for(**int k **=** 0**;** k **<** n**;** k**++)** **for(**int i **=** 0**;** i **<** n**;** i**++)** **for(**int j **=** 0**;** j **<** n**;** j**++)**

**if(**dist**[**i**][**k**]** **!=** Integer**.**MAX\_VALUE **&&** dist**[**k**][**j**]** **!=** Integer**.**MAX\_VALUE**)**

dist**[**i**][**j**]** **=** Math**.**min**(**dist**[**i**][**j**],** dist**[**i**][**k**]** **+** dist**[**k**][**j**]);**

**return** dist**;**

**}**

**Maximum Candidate Bipartite Matching (MCBM)**

**import** java**.**util**.\*;**

public class MaxBipartiteMatching **{**

static Vector**<**Vector**<**Integer**>>** adjList **=** **new** Vector**<>();**

static boolean**[]** visited**;**

static int**[]** match**;**

private static int Aug**(**int left**)** **{**

**if(**visited**[**left**])** **return** 0**;**

visited**[**left**]** **=** **true;**

Iterator it **=** adjList**.**get**(**left**).**iterator**();**

**while(**it**.**hasNext**())** **{**

Integer right **=** **(**Integer**)**it**.**next**();**

**if(**match**[**right**]** **==** **-**1 **||** Aug**(**match**[**right**])** **==** 1**)** **{**

match**[**right**]** **=** left**;**

**return** 1**;**

**}**

**}**

**return** 0**;**

**}**

public static void main**(**String**[]** args**)** **{**

int V **=** 5**,** V\_l **=** 3**;** // V: total vertex, V\_l: vertex on left side

**for(**int i **=** 0**;** i **<** V**;** i**++)** adjList**.**add**(new** Vector**<>());**

//just ignore vertex 0

adjList**.**get**(**1**).**add**(**3**);** // 1 -- 3

adjList**.**get**(**1**).**add**(**4**);** // \

adjList**.**get**(**2**).**add**(**3**);** // \

// 2 -- 4

int MCBM **=** 0**;**

match **=** **new** int**[**V**];**

Arrays**.**fill**(**match**,** **-**1**);**

**for(**int i **=** 0**;** i **<** V\_l**;** i**++)** **{**

visited **=** **new** boolean**[**V\_l**];**

MCBM **+=** Aug**(**i**);**

**}**

System**.**out**.**println**(**MCBM**);**

**}**

**}**

**Minimum Spanning Tree (MST)**

Two approaches. Kruskal’s version:

//MST Kruskal's

//you can find the Union-Find code on Page 10

static class UF **{** /\* code... \*/ **}**

static class Edge **implements** Comparable**<**Edge**>** **{**

int v1**,** v2**,** weight**;**

Edge**(**int vv1**,** int vv2**,** int w**)** **{**

v1 **=** vv1**;**

v2 **=** vv2**;**

weight **=** w**;**

**}**

public int compareTo**(**Edge e**)** **{return** Integer**.**compare**(**weight**,** e**.**weight**);}**

**}**

static int mstWeight**;**

//edges is the graph, n is the number of vertex

//return the MST

public static List**<**Edge**>** getMST**(**List**<**Edge**>** edges**,** int n**)** **{**

Collections**.**sort**(**edges**);**

UF uf **=** **new** UF**(**n**);**

mstWeight **=** 0**;**

List**<**Edge**>** result **=** **new** ArrayList**<>();**

**for(**Edge e **:** edges**)** **{**

**if(!**uf**.**isSameSet**(**e**.**v1**,** e**.**v2**))** **{**

result**.**add**(**e**);**

uf**.**unionSet**(**e**.**v1**,** e**.**v2**);**

mstWeight **+=** e**.**weight**;**

**}**

**}**

**return** result**;**

**}**

//example of MST

public static void main**(**String**[]** args**)** **{**

List**<**Edge**>** e **=** **new** ArrayList**<>();**

e**.**add**(new** Edge**(**0**,** 1**,** 3**));** // 1 6

e**.**add**(new** Edge**(**0**,** 3**,** 1**));** // 0 ---- 3 ---- 4

e**.**add**(new** Edge**(**1**,** 3**,** 3**));** // | / | / |

e**.**add**(new** Edge**(**1**,** 2**,** 1**));** // 3| / |1 / |2

e**.**add**(new** Edge**(**3**,** 2**,** 1**));** // | /3 | /5 |

e**.**add**(new** Edge**(**3**,** 4**,** 6**));** // | / | / |

e**.**add**(new** Edge**(**2**,** 4**,** 5**));** // 1 ---- 2 ---- 5

e**.**add**(new** Edge**(**2**,** 5**,** 4**));** // 1 4

e**.**add**(new** Edge**(**4**,** 5**,** 2**));**

List**<**Edge**>** result **=** getMST**(**e**,** 6**);**

//output: mstWeight = 9, result = {0-3, 1-2, 3-2, 4-5, 2-5}

**}**

Prim’s MST

//Prim's

static class Vertex **{**

int id**;**

List**<**Edge**>** adj **=** **new** ArrayList**<>();**

Vertex**(**int a**)** **{**id **=** a**;}**

**}**

static class Edge **implements** Comparable**<**Edge**>** **{**

int weight**;**

Vertex target**;**

Vertex src**;**

Edge**(**Vertex s**,** Vertex t**,** int w**)** **{**weight **=** w**;** target **=** t**;** src **=** s**;}**

public int compareTo**(**Edge e**)** **{return** Integer**.**compare**(**weight**,** e**.**weight**);}**

**}**

//this class is to store the edges in MST

//if only need the MST weight, then ignore this

static class IntPair **{**

int v1**,** v2**;**

IntPair**(**int a**,** int b**)** **{**v1 **=** a**;** v2 **=** b**;}**

**}**

static int mstWeight**;**

//took a graph and return the MST, or don't return anything if just need the min cost

public static List**<**IntPair**>** mst**(**List**<**Vertex**>** vertices**)** **{**

mstWeight **=** 0**;**

List**<**IntPair**>** result **=** **new** ArrayList**<>();** //store the MST

boolean**[]** visited **=** **new** boolean**[**vertices**.**size**()];**

visited**[**0**]** **=** **true;**

PriorityQueue**<**Edge**>** pq **=** **new** PriorityQueue**<>();**

**for(**Edge e **:** vertices**.**get**(**0**).**adj**)**

pq**.**offer**(**e**);**

**while(!**pq**.**isEmpty**())** **{**

Edge u **=** pq**.**poll**();**

**if(!**visited**[**u**.**target**.**id**])** **{**

mstWeight **+=** u**.**weight**;**

result**.**add**(new** IntPair**(**u**.**src**.**id**,** u**.**target**.**id**));** //store the MST

visited**[**u**.**target**.**id**]** **=** **true;**

**for(**Edge e **:** u**.**target**.**adj**)** **{**

**if(!**visited**[**e**.**target**.**id**])** pq**.**offer**(**e**);**

**}**

**}**

**}**

**return** result**;** //return MST

**}**

public static void main**(**String**[]** args**)** **{**

//example:

List**<**Vertex**>** vertices **=** **new** ArrayList**<>();**

**for(**int i **=** 0**;** i **<** 6**;** i**++)** vertices**.**add**(new** Vertex**(**i**));**

/\*

\* input:

\*

\* 0 3 1 // 1 6

\* 0 1 3 // 0 ---- 3 ---- 4

\* 1 2 1 // | / | / |

\* 1 3 3 // 3| / |1 / |2

\* 2 3 1 // | /3 | /5 |

\* 2 4 5 // | / | / |

\* 2 5 4 // 1 ---- 2 ---- 5

\* 3 4 6 // 1 4

\* 4 5 2

\*

\*/

Scanner in **=** **new** Scanner**(**System**.**in**);**

**for(**int i **=** 0**;** i **<** 9**;** i**++)** **{**

int a **=** in**.**nextInt**(),** b **=** in**.**nextInt**(),** w **=** in**.**nextInt**();**

vertices**.**get**(**a**).**adj**.**add**(new** Edge**(**vertices**.**get**(**a**),** vertices**.**get**(**b**),** w**));**

vertices**.**get**(**b**).**adj**.**add**(new** Edge**(**vertices**.**get**(**b**),** vertices**.**get**(**a**),** w**));**

**}**

List**<**IntPair**>** result **=** **(**mst**(**vertices**));**

// mstWeight = 9, result = {0-3, 3-2, 2-1, 2-5, 5-4}

**}**

**Maximum Flow (Ford-Fulkerson Algorithm) O(VE2)**

BFS Version:

public static int maxFlow**(**int cap**[][],** int src**,** int dest**)** **{**

int**[][]** resiCap **=** **new** int**[**cap**.**length**][**cap**[**0**].**length**];**

**for(**int i **=** 0**;** i **<** cap**.**length**;** i**++)** **for(**int j **=** 0**;** j **<** cap**[**0**].**length**;** j**++)**

resiCap**[**i**][**j**]** **=** cap**[**i**][**j**];**

Map**<**Integer**,** Integer**>** parent **=** **new** HashMap**<>();**

List**<**List**<**Integer**>>** augPath **=** **new** ArrayList**<>();** //Paths are stored here

int maxFlow **=** 0**;**

**while(**bfs**(**resiCap**,** parent**,** src**,** dest**))** **{**

List**<**Integer**>** augmentedPath **=** **new** ArrayList**<>();**

int flow **=** Integer**.**MAX\_VALUE**;**

int v **=** dest**;**

**while(**v **!=** src**)** **{**

augmentedPath**.**add**(**v**);**

int u **=** parent**.**get**(**v**);**

flow **=** Math**.**min**(**flow**,** resiCap**[**u**][**v**]);**

v **=** u**;**

**}**

augmentedPath**.**add**(**src**);**

Collections**.**reverse**(**augmentedPath**);**

augPath**.**add**(**augmentedPath**);**

maxFlow **+=** flow**;**

v **=** dest**;**

**while(**v **!=** src**)** **{**

int u **=** parent**.**get**(**v**);**

resiCap**[**u**][**v**]** **-=** flow**;**

resiCap**[**v**][**u**]** **+=** flow**;**

v **=** u**;**

**}**

**}**

**return** maxFlow**;**

**}**

public static boolean bfs**(**int**[][]** resiCap**,** Map**<**Integer**,** Integer**>** parent**,**

int src**,** int dest**)** **{**

Set**<**Integer**>** visited **=** **new** HashSet**<>();**

Queue**<**Integer**>** queue **=** **new** LinkedList**<>();**

queue**.**add**(**src**);**

visited**.**add**(**src**);**

**while(!**queue**.**isEmpty**())** **{**

int u **=** queue**.**poll**();**

**for(**int v **=** 0**;** v **<** resiCap**.**length**;** v**++)** **{**

**if(!**visited**.**contains**(**v**)** **&&** resiCap**[**u**][**v**]** **>** 0**)** **{**

parent**.**put**(**v**,** u**);**

visited**.**add**(**v**);**

queue**.**add**(**v**);**

**if(**v **==** dest**)** **return** **true;**

**}**

**}**

**}**

**return** **false;**

**}**DFS Version: (This is faster than BFS version if it is used to solve bipartite matching problem)

//C - store the capacity. C[i][j] - capacity i -> j

//F - store the result

static int**[][]** C**,** F**;** //C has to be pre-computed before calling flow method

//adjacent list, this is used when there are large number of vertex,

//but small number of edges connected to each vertex

static List**<**List**<**Integer**>>** adj **=** **new** ArrayList**<>();**

static boolean**[]** visited**;**

//s - source

//t - destination

//return the maximum flow

private static int flow**(**int s**,** int t**)** **{**

// Visited array to perform DFS, initially empty

visited **=** **new** boolean**[**C**.**length**];**

//initialize adjacent list

**for(**int i **=** 0**;** i **<** C**.**length**;** i**++)** adj**.**add**(new** ArrayList**<>());**

**for(**int i **=** 0**;** i **<** C**.**length**;** i**++)** **for(**int j **=** 0**;** j **<** C**.**length**;** j**++)** **{**

**if(**C**[**i**][**j**]** **>** 0**)** **{**

adj**.**get**(**i**).**add**(**j**);**

adj**.**get**(**j**).**add**(**i**);**

**}**

**}**

int maxflow **=** 0**,** flow **=** 0**;**

// Repeat until there is no path

**while** **((**flow **=** dfs**(**s**,** t**,** Integer**.**MAX\_VALUE**))** **>** 0**)** **{**

visited **=** **new** boolean**[**C**.**length**];**

maxflow **+=** flow**;**

**}**

**return** maxflow**;**

**}**

private static int dfs**(**int i**,** int t**,** int min**)** **{**

// If sink has been reached, terminate

**if** **(**i**==**t**)** **return** min**;**

visited**[**i**]** **=** **true;**

**for** **(**int j **:** adj**.**get**(**i**))** **{**

**if** **(**C**[**i**][**j**]** **>** 0 **&&** **!**visited**[**j**])** **{**

int v **=** dfs**(**j**,** t**,** Math**.**min**(**min**,** C**[**i**][**j**]));**

**if** **(**v **>** 0**)** **{**

C**[**i**][**j**]** **=** C**[**i**][**j**]** **-** v**;**

F**[**i**][**j**]** **=** F**[**i**][**j**]** **+** v**;**

C**[**j**][**i**]** **=** C**[**j**][**i**]** **+** v**;**

F**[**j**][**i**]** **=** F**[**j**][**i**]** **-** v**;**

**return** v**;**

**}**

**}**

**}**

// The sink has not been found.

**return** 0**;**

**}**

**Geometry**

Nearly everything you need in geometry is in the following codes.

**import** java**.**util**.\*;**

public class Geometry **{**

//use integer version of points in most of the situations

//if the input data is floating number, let's say up to two decimal places

//just multiply every x and y by 100.

static class Point\_i **implements** Comparable**<**Point\_i**>{**

long x**,** y**;**

Point\_i**(**long xx**,** long yy**)** **{**

x **=** xx**;**

y **=** yy**;**

**}**

Point\_i minus**(**Point\_i a**)** **{** **return** **new** Point\_i**(**x **-** a**.**x**,** y **-** a**.**y**);** **}**

long dot**(**Point\_i a**)** **{** **return** x **\*** a**.**x **+** y **\*** a**.**y**;** **}**

long cross**(**Point\_i a**)** **{** **return** x **\*** a**.**y **-** y **\*** a**.**x**;** **}**

public int compareTo**(**Point\_i p**)** **{**

**if(**x **!=** p**.**x**)** **return** Long**.**compare**(**x**,** p**.**x**);**

**return** Long**.**compare**(**y**,** p**.**y**);**

**}**

**}**

//use this version of point if the answer required is really precise

//something like up to 5 decimal places

static class Point\_f **{**

double x**,** y**;**

Point\_f**(**double xx**,** double yy**)** **{**x **=** round**(**xx**);** y **=** round**(**yy**);}**

//java didn't handle precision well,

//this method is just round the number to 5 decimal places

//change the number if you need more precise number

private double round**(**double a**)** **{return** **(**double**)**Math**.**round**(**a **\*** 100000**)** **/** 100000**;}**

**}**

//rotate point p counter clock wise by rad respect to pivot

//this is the place where you want to use floating points instead of integer points

static Point\_f rotate**(**Point\_f pivot**,** Point\_f p**,** double rad**)** **{**

double x **=** p**.**x **-** pivot**.**x**,** y **=** p**.**y **-** pivot**.**y**;**

double newx **=** x **\*** Math**.**cos**(**rad**)** **-** y **\*** Math**.**sin**(**rad**);**

double newy **=** x **\*** Math**.**sin**(**rad**)** **+** y **\*** Math**.**cos**(**rad**);**

**return** **new** Point\_f**(**newx **+** pivot**.**x**,** newy **+** pivot**.**y**);**

**}**

//return a positive number if a->b->c is a left turn

//negative if a->b->c is a right turn

//the absolute value of the number returned

//is the area of the parallelogram with the side AB and AC

static long ccw**(**Point\_i a**,** Point\_i b**,** Point\_i c**)** **{**

**return** b**.**minus**(**a**).**cross**(**c**.**minus**(**a**));**

**}**

// \*\*\*IMPORTANT\*\*\* return 2x the area

static long triangleArea**(**Point\_i a**,** Point\_i b**,** Point\_i c**)** **{**

**return** **(-**b**.**y**\***c**.**x **+** a**.**y**\*(-**b**.**x **+** c**.**x**)** **+** a**.**x**\*(**b**.**y **-** c**.**y**)** **+** b**.**x**\***c**.**y**);**

**}**

//return true if point C lies on line AB

static boolean inIntSegment**(**Point\_i a**,** Point\_i b**,** Point\_i c**)** **{**

**if(**ccw**(**a**,** b**,** c**)** **!=** 0**)** **return** **false;** //if a->b->c is a left turn or right turn

Point\_i tmp **=** b**.**minus**(**a**);**

long dotProduct **=** c**.**minus**(**a**).**dot**(**tmp**);**

long sqrLength **=** tmp**.**dot**(**tmp**);**

**return** **!(**dotProduct **<** 0 **||** dotProduct **>** sqrLength**);**

**}**

//return true if point P is inside or on the triangle ABC

static boolean inTriangle**(**Point\_i a**,** Point\_i b**,** Point\_i c**,** Point\_i p**)** **{**

//remove this check if only want points that is inside the triangle

**if(**inIntSegment**(**a**,** b**,** p**)** **||** inIntSegment**(**a**,** c**,** p**)** **||** inIntSegment**(**b**,** c**,** p**))**

**return** **true;**

long area **=** triangleArea**(**a**,** b**,** c**);** //2x area works in this case

**if(**area **==** 0**)** **return** **false;**

double s**=** **(**double**)(**a**.**y**\***c**.**x **-** a**.**x**\***c**.**y **+** **(**c**.**y **-** a**.**y**)\***p**.**x **+** **(**a**.**x **-** c**.**x**)\***p**.**y**)** **/** area**;**

**if(**s **<=** 0**)** **return** **false;**

double t**=** **(**double**)(**a**.**x**\***b**.**y **-** a**.**y**\***b**.**x **+** **(**a**.**y **-** b**.**y**)\***p**.**x **+** **(**b**.**x **-** a**.**x**)\***p**.**y**)** **/** area**;**

**return** t **>** 0 **&&** 1 **-** s **-** t **>** 0**;**

**}**

//return true if the point p is inside or on the polygon

//O(logn) approach!! Using the idea of binary search

static boolean inPolygon**(**Point\_i p**,** List**<**Point\_i**>** hull**)** **{**

int left **=** 1**,** right **=** hull**.**size**()** **-** 3**;**

Point\_i first **=** hull**.**get**(**0**);**

**while(**left **<=** right**)** **{**

final int mid **=** **(**left **+** right**)** **/** 2**;**

long cross1 **=** ccw**(**first**,** hull**.**get**(**mid**),** p**);**

long cross2 **=** ccw**(**first**,** p**,** hull**.**get**(**mid **+** 1**));**

**if(**cross1 **>=** 0 **&&** cross2 **>=** 0 **&&**

inTriangle**(**first**,** hull**.**get**(**mid**),** hull**.**get**(**mid **+** 1**),** p**))**

**return** **true;**

**else** **if(**cross1 **<** 0**)**

right **=** mid **-** 1**;**

**else**

left **=** mid **+** 1**;**

**}**

**return** **false;**

**}**

//return the convex hull of a set of points

static List**<**Point\_i**>** convexHull**(**List**<**Point\_i**>** p**)** **{**

int i**,** t**,** k **=** 0**,** n **=** p**.**size**();**

List**<**Point\_i**>** H **=** **new** ArrayList**<>(**2**\***n**);**

Collections**.**sort**(**p**);**

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

**while(**k **>** 1 **&&** ccw**(**H**.**get**(**k **-** 2**),** H**.**get**(**k **-** 1**),** p**.**get**(**i**))** **<=** 0**)** k**--;**

**if(**H**.**size**()** **>** k**)** H**.**set**(**k**,** p**.**get**(**i**));**

**else** H**.**add**(**p**.**get**(**i**));**

k**++;**

**}**

**for(**i **=** n **-** 2**,** t **=** k**;** i **>=** 0**;** i**--)** **{**

**while(**k **>** t **&&** ccw**(**H**.**get**(**k **-** 2**),** H**.**get**(**k **-** 1**),** p**.**get**(**i**))** **<=** 0**)** k**--;**

**if(**H**.**size**()** **>** k**)** H**.**set**(**k**,** p**.**get**(**i**));**

**else** H**.**add**(**p**.**get**(**i**));**

k**++;**

**}**

**return** H**.**subList**(**0**,** k**);**

**}**

//return the area of a polygon

static double polygonArea**(**List**<**Point\_i**>** list**)** **{**

double area **=** 0**;**

**for(**int i **=** 0**;** i **<** list**.**size**();** i**++)** **{**

Point\_i p1 **=** list**.**get**(**i**);**

Point\_i p2 **=** list**.**get**((**i **+** 1**)** **%** list**.**size**());**

area **+=** p1**.**cross**(**p2**);**

**}**

**return** Math**.**abs**(**area**)** **/** 2**;**

**}**

//same as ccw, this just return true or false

static boolean leftTurn**(**Point\_i a**,** Point\_i b**,** Point\_i c**)** **{**

**return** b**.**minus**(**a**).**cross**(**c**.**minus**(**a**))** **>** 0**;**

**}**

//return true if line AB intersect with line CD

//\*\*\*IMPORTANT\*\*\* does not include the case that one end of a line in on another line

static boolean doIntersect**(**Point\_i a**,** Point\_i b**,** Point\_i c**,** Point\_i d**)** **{**

**return** leftTurn**(**a**,** b**,** c**)** **!=** leftTurn**(**a**,** b**,** d**)** **&&**

leftTurn**(**c**,** d**,** a**)** **!=** leftTurn**(**c**,** d**,** b**);**

**}**

//return true if it is a simple polygon

static boolean isSimplePolygon**(**List**<**Point\_i**>** list**)** **{**

**for(**int i **=** 0**;** i **<** list**.**size**();** i**++)** **{**

**for(**int j **=** 0**;** j **<** list**.**size**();** j**++)** **{**

**if(**Math**.**abs**(**i **-** j**)** **<=** 1 **||** Math**.**abs**(**i **-** j**)** **>=** list**.**size**()** **-** 1**)**

**continue;**

**if(**doIntersect**(**list**.**get**(**i**),** list**.**get**((**i **+** 1**)** **%** list**.**size**()),**

list**.**get**(**j**),** list**.**get**((**j **+** 1**)** **%** list**.**size**())))**

**return** **false;**

**}**

**}**

**return** **true;**

**}**

//cut polygon

final static double EPS **=** 1e-9**;**

//cuts polygon Q along the line formed by point a -> point b

//\*\*\*IMPORTANT\*\*\* the last point must be the same as the first point

static List**<**Point\_f**>** cutPolygon**(**Point\_f a**,** Point\_f b**,** List**<**Point\_f**>** Q**)** **{**

List**<**Point\_f**>** P **=** **new** ArrayList**<**Point\_f**>();**

**for** **(**int i **=** 0**;** i **<** **(**int**)**Q**.**size**();** i**++)** **{**

double left1 **=** b**.**minus**(**a**).**cross**(**Q**.**get**(**i**).**minus**(**a**)),** left2 **=** 0**;**

**if** **(**i **!=** **(**int**)**Q**.**size**()-**1**)** left2 **=** b**.**minus**(**a**).**cross**(**Q**.**get**(**i **+** 1**).**minus**(**a**));**

**if** **(**left1 **>** **-**EPS**)** P**.**add**(**Q**.**get**(**i**));** // Q[i] is on the left of ab

**if** **(**left1 **\*** left2 **<** **-**EPS**)** // edge (Q[i], Q[i+1]) crosses line ab

P**.**add**(**lineIntersectSeg**(**Q**.**get**(**i**),** Q**.**get**(**i**+**1**),** a**,** b**));**

**}**

**if** **(!**P**.**isEmpty**()** **&&**

**!(**P**.**get**(**P**.**size**()** **-** 1**).**x **==** P**.**get**(**0**).**x **&&** P**.**get**(**P**.**size**()** **-** 1**).**y **==** P**.**get**(**0**).**y**))**

P**.**add**(**P**.**get**(**0**));** // make P's first point = P's last point

**return** P**;**

**}**

//line segment p-q intersect with line A-B.

static Point\_f lineIntersectSeg**(**Point\_f p**,** Point\_f q**,** Point\_f A**,** Point\_f B**)** **{**

double a **=** B**.**y **-** A**.**y**;**

double b **=** A**.**x **-** B**.**x**;**

double c **=** B**.**x **\*** A**.**y **-** A**.**x **\*** B**.**y**;**

double u **=** Math**.**abs**(**a **\*** p**.**x **+** b **\*** p**.**y **+** c**);**

double v **=** Math**.**abs**(**a **\*** q**.**x **+** b **\*** q**.**y **+** c**);**

**return** **new** Point\_f**((**p**.**x **\*** v **+** q**.**x **\*** u**)** **/** **(**u**+**v**),** **(**p**.**y **\*** v **+** q**.**y **\*** u**)** **/** **(**u**+**v**));**

**}**

**}**

**Mathematics**

**Base Conversion**

There are built-in libraries to do this. Import java.math.\* to use the BigInteger class

//take a string, read as any base

BigInteger a **=** **new** BigInteger**(**"fff"**,** 16**);** //read "fff" as hex.

String binary **=** a**.**toString**(**2**);** //convert this number to binary.

**Modulo Power**

Calculate an mod b in O(logn) time for large n

//modulo power

BigInteger a **=** **new** BigInteger**(**"2"**);**

BigInteger n **=** **new** BigInteger**(**"1000000000"**);**

BigInteger b **=** **new** BigInteger**(**"1000000007"**);**

//a^n mod b. Calculated in O(logn) time, where n is the exponent

BigInteger ans **=** a**.**modPow**(**n**,** b**);**

**Prime Number**

Here are some useful methods that dealing with primes numbers.

static List**<**Integer**>** primes **=** **new** ArrayList**<>();**

static boolean**[]** bs**;**

//generate all prime number that is less than or equal to upper

//pretty fast if upper <= 10^7

//call this method only once in main method, use 10^7 is enough for most of the cases

//a lot of following methods need prime list

public static void sieve**(**int upper**)** **{**

int size **=** upper **+** 1**;**

bs **=** **new** boolean**[**size**];**

Arrays**.**fill**(**bs**,** **true);**

bs**[**0**]** **=** bs**[**1**]** **=** **false;**

**for(**long i **=** 2**;** i **<** size**;** i**++)** **if(**bs**[(**int**)**i**])** **{**

**for(**long j **=** i **\*** i**;** j **<** size**;** j**+=** i**)** bs**[(**int**)**j**]** **=** **false;**

primes**.**add**((**int**)**i**);**

**}**

**}**

//return true if the number N is a prime number

//\*\*\*IMPORTANT\*\*\* only works if N <= (the largest prime in the prime list)^2

public static boolean isPrime**(**long N**)** **{**

**if(**N **<** bs.length**)** **return** bs**[(**int**)**N**];**

**for(**Integer p **:** primes**)**

**if(**N **%** p **==** 0**)** **return** **false;**

**return** **true;**

**}**

//return a list of prime factors of N

//e.g. N = 1000 => {2, 2, 2, 5, 5, 5}

//use set instead of list if you don't want duplications

public static List**<**Integer**>** primeFactor**(**long N**)** **{**

List**<**Integer**>** factors **=** **new** ArrayList**<>();**

int i **=** 0**;**

long pf **=** primes**.**get**(**i**);**

**while(**N **!=** 1 **&&** **(**pf **\*** pf**)** **<=** N**)** **{**

**while(**N **%** pf **==** 0**)** **{**

N **/=** pf**;**

factors**.**add**((**int**)**pf**);**

**}**

pf **=** primes**.**get**(++**i**);**

**}**

//special case: N is a prime

**if(**N **!=** 1**)** factors**.**add**((**int**)**N**);**

**return** factors**;**

**}**

//return the number of divisors of N

public static long numDiv**(**long N**)** **{**

int i **=** 0**;**

long pf **=** primes**.**get**(**i**),** ans **=** 1**;**

**while(**N **!=** 1 **&&** **(**pf **\*** pf**)** **<=** N**)** **{**

long pwr **=** 0**;**

**while(**N **%** pf **==** 0**)** **{**

N **/=** pf**;**

pwr**++;**

**}**

ans **\*=** **(**pwr **+** 1**);**

pf **=** primes**.**get**(++**i**);**

**}**

**if(**N **!=** 1**)** ans **\*=** 2**;**

**return** ans**;**

**}**

//return the number of positive integer less than N that are relatively prime to N

public static long EulerPhi**(**long N**)** **{**

int i **=** 0**;**

long pf **=** primes**.**get**(**i**),** ans **=** N**;**

**while(**N **!=** 1 **&&** **(**pf **\*** pf**)** **<=** N**)** **{**

**if(**N **%** pf **==** 0**)** ans **-=** ans **/** pf**;**

**while(**N **%** pf **==** 0**)** N **/=** pf**;**

pf **=** primes**.**get**(++**i**);**

**}**

**if(**N **!=** 1**)** ans **-=** ans **/** N**;**

**return** ans**;**

**}**

**Extended Euclid Theory**

This algorithm can solve linear equations with two variables: ax + by = c

Let d = gcd(a, b), if d | c (d divides c), then there exist infinite pairs of integer solutions. Otherwise there is no integer solution. This algorithm will only find the integer solutions. If the first set of solution (x0, y0) can be found, then the rest can be derived from x = x0 + (b / d) \* n, y = y0 + (a / d) \* n, n ∈ Z.

static int x**,** y**,** d**;**

public static void extendedEuclid**(**int a**,** int b**)** **{**

**if(**b **==** 0**)** **{**

x **=** 1**;**

y **=** 0**;**

d **=** a**;**

**return;**

**}**

extendedEuclid**(**b**,** a **%** b**);**

int x1 **=** y**;**

int y1 **=** x **-** **(**a **/** b**)** **\*** y**;**

x **=** x1**;**

y **=** y1**;**

**}**

Once call this method, x0, y0 and d are stored in x, y, d. Then use the formula to produce the solutions.

**Fibonacci Number**

Standard solution for finding the nth Fibonacci number take O(n) time. It will TLE if n is very large.

Here is an algorithm for finding fib(n) mod b with complexity O(logn)

//return the n-th fibonacci number mod b

public static long fib**(**long n**,** long b**)** **{**

long**[][]** f **=** **{{**1**,** 1**},** **{**1**,** 0**}};**

f **=** pow**(**f**,** n**,** b**);**

**return** f**[**0**][**1**];**

**}**

public static long**[][]** mul**(**long**[][]** A**,** long**[][]** B**,** long mod**)** **{**

long**[][]** C **=** **new** long**[**2**][**2**];**

**for(**int i **=** 0**;** i **<** 2**;** i**++)** **for(**int j **=** 0**;** j **<** 2**;** j**++)** **for(**int k **=** 0**;** k **<** 2**;** k**++)**

C**[**i**][**j**]** **=** **(**C**[**i**][**j**]** **+** A**[**i**][**k**]** **\*** B**[**k**][**j**])** **%** mod**;**

**return** C**;**

**}**

public static long**[][]** pow**(**long**[][]** A**,** long n**,** long mod**)** **{**

**if(**n **==** 1**)** **return** A**;**

**if((**n **&** 1**)** **==** 1**)** **return** mul**(**A**,** pow**(**A**,** n **-** 1**,** mod**),** mod**);**

long**[][]** B **=** pow**(**A**,** n **/** 2**,** mod**);**

**return** mul**(**B**,** B**,** mod**);**

**}**

**Linear Recurrence**

To generalize the method to find nth Fibonacci number, you can solve any linear recurrence function in O(K3logn) time.

Let’s say a linear function *f(n) = CKf(n-K) + CK-1f(n-(K-1)) + …… + C2f(n-2) + C1f(n-1)*

Let’s define a matrix F, where

*f(1)*

*f(2)*

*f(3)*

*…*

*f(k)*

F1 =

Note that the first number in Fi is *f(i)*.

Now, we can define a transformation matrix T, such that FiT = Fi+1

0 1 0 0 … 0

0 0 1 0 … 0

0 0 0 1 … 0

… … … … … ...

0 0 0 0 … 1

CK CK-1 CK-2 CK-3 … C1

T **=**

Since F2 = TF1, F3 = TF2 = T2F1, … and so on, therefore, in general, FN = TN-1F1

Here is the code: (Since *f(n)* might be huge, we usually take *f(n) mod p*)

static int MOD **=** 1000000007**;**

public static void main**(**String**[]** args**)** **{**

//f(n) = 3f(n - 2) + 2f(n - 1)

//f(1) = 1, f(2) = 2

long**[][]** T **=** **{{**0**,** 1**},**

**{**3**,** 2**}};**

long**[][]** F **=** **{{**1**},** **{**2**}};**

//lets say you want to find f(10)

T **=** pow**(**T**,** 9L**);**

F **=** mul**(**T**,** F**);**

System**.**out**.**println**(**F**[**0**][**0**]);**

**}**

public static long**[][]** pow**(**long**[][]** b**,** long e**)** **{**

**if(**e **==** 1**)** **return** b**;**

**if(**e **%** 2 **==** 0**)** **{**

long**[][]** x **=** pow**(**b**,** e **/** 2**);**

**return** mul**(**x**,** x**);**

**}**

**return** mul**(**b**,** pow**(**b**,** e **-** 1**));**

**}**

public static long**[][]** mul**(**long**[][]** a**,** long**[][]** b**)** **{**

long**[][]** c **=** **new** long**[**a**.**length**][**b**[**0**].**length**];**

**for(**int i **=** 0**;** i **<** c**.**length**;** i**++)** **{**

**for(**int j **=** 0**;** j **<** c**[**i**].**length**;** j**++)** **{**

**for(**int k **=** 0**;** k **<** a**[**i**].**length**;** k**++)** **{**

c**[**i**][**j**]** **+=** a**[**i**][**k**]** **\*** b**[**k**][**j**];**

c**[**i**][**j**]** **%=** MOD**;**

**}**

**}**

**}**

**return** c**;**

**}**

**Gaussian Elimination**

This method is for solving any linear equation.

static double**[]** gaussianElimination**(**double**[][]** mat**)** **{**

int len **=** mat**.**length**;**

double**[]** x **=** **new** double**[**len**];**

**for(**int j **=** 0**;** j **<** len **-** 1**;** j**++)** **{**

int l **=** j**;**

**for(**int i **=** j **+** 1**;** i **<** len**;** i**++)** **{**

**if(**Math**.**abs**(**mat**[**i**][**j**])** **>** Math**.**abs**(**mat**[**l**][**j**]))**

l **=** i**;**

**}**

**for(**int k **=** j**;** k **<=** len**;** k**++)** **{**

double t **=** mat**[**j**][**k**];**

mat**[**j**][**k**]** **=** mat**[**l**][**k**];**

mat**[**l**][**k**]** **=** t**;**

**}**

**for(**int i **=** j **+** 1**;** i **<** len**;** i**++)** **for(**int k **=** len**;** k **>=** j**;** k**--)** **{**

mat**[**i**][**k**]** **-=** mat**[**j**][**k**]** **\*** mat**[**i**][**j**]** **/** mat**[**j**][**j**];**

**}**

**}**

**for(**int j **=** len **-** 1**;** j **>=** 0**;** j**--)** **{**

double t **=** 0.0**;**

**for(**int k **=** j **+** 1**;** k **<** len**;** k**++)**

t **+=** mat**[**j**][**k**]** **\*** x**[**k**];**

x**[**j**]** **=** **(**mat**[**j**][**len**]** **-** t**)** **/** mat**[**j**][**j**];**

**}**

**return** x**;**

**}**

public static void main**(**String**[]** args**)** **{**

/\*

\* e.g.

\* 1x + 1y + 2z = 9

\* 2x + 4y - 3z = 1

\* 3x + 6y - 5z = 0

\*/

double**[]** ans **=** gaussianElimination**(new** double**[][]** **{{**1**,** 1**,** 2**,** 9**},**

**{**2**,** 4**,** **-**3**,** 1**},**

**{**3**,** 6**,** **-**5**,** 0**}});**

//ans = {1.0, 2.0, 3.0}

// x y z

**}**

**Reference**

Halim, S. Halim, F. (2013). Competitive Programming 3. Singapore.

Tushar Roy, *https://github.com/mission-peace/interview/tree/master/src/com/interview*