<https://github.com/BrockCSC/acm-icpc/wiki/Online-Resources>

<https://www-s.acm.illinois.edu/archives/icpc-l/msg00375.html>

<http://codeforces.com/blog/entry/23054>

<https://www.quora.com/Can-anyone-provide-me-a-list-of-all-algorithms-needed-for-ACM-ICPC>

[***https://www.youtube.com/watch?v=7mdw94MZ5wI***](https://www.youtube.com/watch?v=7mdw94MZ5wI)

[***https://www.topcoder.com/login/?next=https:%2F%2Farena.topcoder.com%2Findex.html***](https://www.topcoder.com/login/?next=https:%2F%2Farena.topcoder.com%2Findex.html)

***Questions to ask***

Can I use my own keyboard at the competition?

How to decide when to use bigInteger and when not? How to use the BigInteger class efficiently without messy code?

Is it a good idea to always use the same modelling part of the program for graph problems even if the nodes are as simple as just alphabet letters?

**Arrays**

Arrays.binarySearch()

Arrays.copyOf()

Arrays.copyOfRange()

Arrays.deepEquals()

Arrays.deepHashCode()

Arrays.deepToString()

Arrays.equals()

Arrays.fill()

Arrays.hashCode()

Arrays.sort()

Arrays.toString()

**Collections**

addAll()

binarySearch()

copy()

disjoint()

enumeration()

fill()

frequency()

indexOfSubList()

max()

min()

nCopies()

newSetFromMap()

reverse()

rotate()

shuffle()

sort()

swap()

**LinkedLists**

add()

addAll()

addFirst()

addLast()

clear()

clone()

contains()

element()

get()

getLast()

getFirst()

indexOf()

lastIndesOf()

offer()

offerFirst()

offerLast()

peek()

peekFirst()

peekLast()

poll()

pollFirst()

pollLast()

pop()

push()

remove()

removeFirst()

removeFirstOccurance()

removeLast()

removeLastOccurence()

set()

size()

toArray()

**Stack**

empty()

peek()

pop()

push()

search()

**Priority Queue**

add()

clear()

contains()

offer()

peek()

poll()

remove()

size()

toArray()

**Set**

**HashSet**

add()

clear()

clone()

contains()

isEmpty()

remove()

size()

**TreeSet**

add()

addAll()

ceiling()

clear()

clone()

contains()

first()

floor()

higher()

isEmpty()

last()

lower()

pollFirst()

pollLast()

remove()

size()

**Map**

**HashMap**

clear()

clone()

containsKey()

containsValue()

get()

isEmpty()

put()

putAll()

remove()

size()

**TreeMap**

ceilingKey()

clear()

clone()

containsKey()

containsValue()

firstEntry()

floorKey()

floorEntry()

get()

higherEntry()

higherKey()

lastEntry()

lastKey()

lowerEntry()

lowerKey()

pollFirstEntry()

pollLastEntry()

put()

putAll()

remove()

size()

**ArrayList**

add()

addAll()

clear()

clone()

contains()

get()

indexOf()

ensureCapacity()

indexOf()

isEmpty()

lastIndexOf()

remove()

set()

size()

toArray()

trimToSize()

**ArrayDeque**

add()

addFirst()

addLast()

clear()

clone()

contains()

element()

getFirst()

getLast()

isEmpty()

Offer()

offerFirst()

offerLast()

peek()

peekFirst()

peekLast()

poll()

pollFirst()

pollLast()

pop()

push()

remove()

removeFirst()

removeFirstOccurence()

removeLast()

removeLastOccurence()

size()

toArray()

**BitSet**

and()

andNot()

cardinality()

clear()

clone()

equals()

flip()

get()

hashCode()

intersects()

isEmpty()

length()

nextClearBit()

nextSetBit()

or()

set()

size()

toString()

xor()

**Dictionary**

get()

isEmpty()

keys()

put()

remove()

size()

**Vector**

add()

addAll()

addElement()

capacity()

clear()

clone()

contains()

containsAll()

copyInto()

elementAt()

ensureCapacity()

equals()

firstElement()

get()

hashCode()

indexOf()

insertElement()

isEmpty()

lastElement()

lastIndexOf()

remove()

removeAll()

removeAllElements()

removeElementAt()

removeRange()

removeRange()

retainAll()

set()

setSize()

size()

toArray()

toString()

trimToSize()

***Java.Lang***

**Integer**

bitCount()

byteValue()

compareTo()

decode()

doubleValue()

equals()

getInteger()

intValue()

longValue()

numberOfLeadingZeroes()

numberOfTrailingZeroes()

parseInt()

reverse()

reverseBytes()

rotateRight()

rotateLeft()

shortValue()

signum()

toBinaryString()

toHexString()

toOctalString()

toString()

valueOf()

**Math**

abs()

acos()

asin()

atan()

atan2()

cbrt()

ceil()

copySign()

cos()

cosh()

exp()

floor()

getExponent()

hypot()

log()

log10()

max()

min()

nextAfter()

nextUp()

pow()

rint()

round()

scalb()

signum()

sin()

sinh()

sqrt()

tan()

tanh()

toDegrees()

toRadians()

**String**

charAt()

compareTo()

compareToIgnoreCase()

concat()

contains()

contentEquals()

copyValueOf()

endsWith()

equals()

equalsIgnoreCase()

format()

getBytes()

hashCode()

indexOf()

isEmpty()

lastIndexOf()

length()

matches()

regionMatches()

replace()

replaceAll()

replaceFirst()

split()

subSequence()

substring()

toCharArray()

toLowerCase()

toString()

toUpperCase()

trim()

valueOf()

**StringBuilder()**

append()

capacity()

charAt()

delete()

deleteCharAt()

ensureCapacity()

getChars()

indexOf()

insert()

lastIndexOf()

length()

replace()

reverse()

setCharAt()

setLength()

subSequence()

substring()

toString()

trimToSize()

**java.awt.geom**

**EnumMap**

**EnumSet**

**Formatter**

**GregorianCalendar**

**Random**

**Timer**

**Calendar**

**Currency**

**Date**

Making change for a customer How do you make 86 cents for a customer?Greedy Algorithm, recursively solve in an optimized way Locally optimal solution to a subproblem They appear very disguised in the competition. There need to be substructures in the problem. If the subproblems can be solved optimally then the initial problem can also be optimally solved. They can be solved iteratively.Make the best choice at any moment.identify the greedy property

Programming Exercises solvable using Greedy (hints omitted):

1. UVa 410 - Station Balance (elaborated in this section)

2. UVa 10020 - Minimal Coverage

3. UVa 10340 - All in All

4. UVa 10440 - Ferry Loading II

5. UVa 10670 - Work Reduction

6. UVa 10763 - Foreign Exchange

7. UVa 11054 - Wine Trading in Gergovia

8. UVa 11292 - Dragon of Loowater

9. UVa 11369 - Shopaholic

What do we do with the left over part after making the first initial choice

Dynamic Programming

Knapsack Problem

When you have to make a choice between which path to choose for a optimization

These problems will be set up with test cases to make you solve it using Greedy algorithm when it is not the right solution

Differences between Greedy and Dynamic

What is the least number of changes required to change an initial word to a target word?

Longiest common subsequence

C++ Sorting

<https://en.wikipedia.org/wiki/Sort_(C%2B%2B)>

What do we use to sort in java quickly?

**Data Structures**

**Static Array in C/C++ and in Java**

Resizeable Array a.k.a. Vector: C++ STL <vector> (Java ArrayList)

Using vector over array is better if array size is unknown beforehand, i.e. before running the program. Usually, we initialize the size with some guess value for better performance. Typical operations are: push back(), at(), [] operator, erase(), and typically use iterator to scan the content of the vector.

**Efficient Sorting and Searching in Static/Resize-able Array**

**O(nlogn) comparison-based sorting algorithms [4]: Merge/Heap/Random Quick Sort.**

We can use **C++ STL sort, partial sort, stable sort, in<algorithm>** to achieve this purpose (**Java Collections.sort)**. We only need to specify the required comparison function and these library routines will handle the rest.

**O(logn) Binary Search: use lower bound in C++ STL <algorithm> (or Java Collections.binarySearch).**

If the input is unsorted, it is fruitful to sort it just once using an O(nlogn) sorting algorithm above in order to use Binary Search many times.

**Linked Lists**

**Linked List: C++ STL <list> (Java LinkedList)**

Although this data structure almost always appears in data structure & algorithm textbooks, Linked List is usually avoided in typical contest problems. Reasons: it involves pointers and theoretically slow for accessing data as it has to be performed from the head or tail of a list.

**Stack: C++ STL <stack> (Java Stack)**

This data structure is used as part of algorithm to solve a certain problem (e.g. Postﬁx calculation, Graham’s scan in Section 7.3). Stack only allows insertion (push) and deletion (pop) from the top only. This behavior is called Last In First Out (LIFO) as with normal stack in the real world. Typical operations are push()/pop() (insert/remove from top of stack), top() (obtain content from the top of stack), empty()

**Queue: C++ STL (Java Queue)**

This data structure is used in algorithms like Breadth First Search (BFS) (Section 4.3). A queue only allows insertion (enqueue) from the back (rear), and only allows deletion (dequeue) from the head (front). This behavior is called First In First Out (FIFO), similar to normal queue in the real world. Typical operations are push()/pop() (insert from back/take out from front of queue), front()/back() (obtain content from the front/back of queue), empty()

**• Balanced Binary Search Tree (BST): C++ STL / (Java TreeMap/TreeSet)**

[**https://www.youtube.com/watch?v=m7s6ulOJOAM**](https://www.youtube.com/watch?v=m7s6ulOJOAM)

BST is a way to organize data as a tree-structure. In each subtree rooted at x, this BST property holds: items on the left subtree of x are smaller than x and items on the right subtree of x are greater (or equal) than x. Organizing the data like this (see Figure 2.1, left) allows O(log n) insertion, search, and deletion as only O(log n) worst case root-to-leaf scan is needed to perform those actions (details in [4]) – but this only works if the BST is balanced.

Implementing a bug-free balanced BST like AVL2 Tree or Red-Black (RB) Tree is tedious and hard to do under time constrained contest environment. Fortunately, C++ STL has and which are usually the implementation of RB Tree, thus all operations are in O(log n). Mastery of these two STL templates can save a lot of precious coding time during contests! The difference is simple: stores (key → data) pair whereas only stores the key. •

**Heap: C++ STL : priority queue (Java PriorityQueue)**

Heap is another way to organize data as a tree-structure. Heap is also a binary tree like BST but it must be complete. Instead of enforcing BST property, Heap enforces Heap property: In each subtree rooted at x, items on the left and the right subtrees of x are smaller than x (see Figure 2.1, right). This property guarantees that the top of the heap is the maximum element. There is usually no notion of ‘search’ in Heap, but only insertion and deletion, which can be easily done by traversing a O(log n) leaf-to-root or root-to-leaf path [4].

**Graphs**

A Adjacency Matrix, usually in form of 2-D array. In contest problems involving graph, usually V is known, thus we can build a ‘connectivity table’ by setting up a 2-D, O(V 2) static array: int AdjMat[V ][V ]. For an unweighted graph, we set AdjMat[i][j] = 1 if there is an edge between vertex i-j and set 0 otherwise. For a weighted graph, we set AdjMat[i][j] = weight(i, j) if there is an edge between vertex i-j with weight(i, j) and set 0 otherwise. Adjacency Matrix is good if the connectivity between two vertices in a small dense graph is frequently asked, but it is not good for large sparse graph as there will be too many cells in the 2-D array that are blank (contain zeroes). An adjacency Matrix requires exactly O(V ) to enumerate the list of neighbors of a vertex v – an operation commonly used in many graph algorithms – even if vertex v only has a handful of neighbors. A more compact and efficient form of graph representation is Adjacency List.

B Adjacency List, usually in form of C++ STL vector AdjList, with vii defined as: typedef pair ii; typedef vector vii; // our data type shortcuts In Adjacency List, we have a vector of V vertices and for each vertex v, we store another vector that contains pairs of (neighboring vertex and it’s edge weight) that have connection to v. If the graph is unweighted, simply store weight = 0 or drop this second attribute. With Adjacency List, we can enumerate the list of neighbors of a vertex v efficiently. If there are k neighbors of v, this enumeration is O(k). As this is one of the most common operations in most graph algorithms, it is advisable to stick with Adjacency List as your default choice.

C Edge List, usually in form of C++ STL priority queue > EdgeList. In Edge List, we store the list of edges, usually in some order. This structure is very useful for Kruskal’s algorithm for MST (Section 4.4) where the collection of edges are sorted by their length from shortest to longest.

D Parent-Child Tree Structure, usually in form of int parent & C++ STL vector child. If the graph is a tree (connected graph with no cycle and E = V − 1), like a directory/folder structure, then there exists another form of data structure. For each vertex, we only store two attributes: the parent (NULL for root vertex) and the list of children (NULL for leaves).

**Dynamic Programming**

1. Find out the recurrence relation such as shop(a,b ) = max [shop(a-price[i][],b+1 )]
2. Construct the 2 dimensional table for the recurrence relation

Programming Exercises for problems solvable using Dynamic Programming:

• Longest Increasing Subsequence (LIS) – Classical

1. UVa 103 - Stacking Boxes (Longest Path in DAG ≈ LIS)

2. UVa 111 - History Grading (straight-forward)

3. UVa 231 - Testing the Catcher (straight-forward)

4. UVa 481 - What Goes Up? (must use O(nlogk) LIS)

5. UVa 497 - Strategic Defense Initiative (solution must be printed)

6. UVa 10051 - Tower of Cubes (can be modeled as LIS)

7. UVa 10534 - Wavio Sequence (must use O(nlogk) LIS twice)

8. UVa 11790 - Murcia’s Skyline (combination of classical LIS+LDS, weighted)

9. UVa 11003 - Boxes

10. UVa 11456 - Trainsorting (get max(LIS(i) + LDS(i) - 1), ∀i ∈[0...N-1])

11. LA 2815 - Tiling Up Blocks (Kaohsiung03)

• Coin Change – Classical

1. UVa 147 - Dollars (similar to UVa 357 and UVa 674)

2. UVa 166 - Making Change

3. UVa 357 - Let Me Count The Ways (a variant of the coin change problem

4. UVa 674 - Coin Change

55

3.4. DYNAMIC PROGRAMMING c Steven & Felix, NUS

5. UVa 10306 - e-Coins

6. UVa 10313 - Pay the Price

7. UVa 11137 - Ingenuous Cubrency (use long long)

8. UVa 11517 - Exact Change

• Maximum Sum

1. UVa 108 - Maximum Sum (maximum 2-D sum, elaborated in this section)

2. UVa 836 - Largest Submatrix (maximum 2-D sum)

3. UVa 10074 - Take the Land (maximum 2-D sum)

4. UVa 10667 - Largest Block (maximum 2-D sum)

5. UVa 10827 - Maximum Sum on a Torus (maximum 2-D sum)

6. UVa 507 - Jill Rides Again (maximum 1-D sum/maximum consecutive subsequence)

7. UVa 10684 - The Jackpot (maximum 1-D sum/maximum consecutive subsequence)

• 0-1 Knapsack – Classical

1. UVa 562 - Dividing Coins

2. UVa 990 - Diving For Gold

3. UVa 10130 - SuperSale

4. LA 3619 - Sum of Diﬀerent Primes (Yokohama06)

• String Edit (Alignment) Distance – Classical (see Section 6.3)

• Longest Common Subsequence – Classical (see Section 6.3)

• Non Classical (medium diﬃculty)

1. UVa 116 - Unidirectional TSP (similar to UVa 10337)

2. UVa 473 - Raucuous Rockers

3. UVa 607 - Scheduling Lectures

4. UVa 10003 - Cutting Sticks (discussed in this book)

5. UVa 10337 - Flight Planner (DP solvable with Dijkstra)

6. UVa 10891 - Game of Sum (2 dimensional states)

7. UVa 11450 - Wedding Shopping (discussed in this book)

8. LA 3404 - Atomic Car Race (Tokyo05)

• DP + Bitmasks

1. UVa 10364 - Square (bitmask technique can be used)

2. UVa 10651 - Pebble Solitaire

3. UVa 10908 - Largest Square

4. UVa 10911 - Forming Quiz Teams (elaborated in this section)

5. LA 3136 - Fun Game (Beijing04)

6. PKU 2441 - Arrange the Bulls

• DP on ‘Graph Problem’

1. UVa 590 - Always on the Run

2. UVa 910 - TV Game (straightforward)

3. UVa 10681 - Teobaldo’s Trip

4. UVa 10702 - Traveling Salesman

5. LA 4201 - Switch Bulbs (Dhaka08)

6. SPOJ 101 - Fishmonger

56

3.5. CHAPTER NOTES c Steven & Felix, NUS

• DP with non-trivial states

1. LA 4106 - ACORN (Singapore07) (DP with dimension reduction)

2. LA 4143 - Free Parentheses (Jakarta08) (Problem set by Felix Halim)

3. LA 4146 - ICPC Team Strategy (Jakarta08) (DP with 3 states)

4. LA 4336 - Palindromic paths (Amritapuri08)

5. LA 4337 - Pile it down (Amritapuri08)

6. LA 4525 - Clues (Hsinchu09)

7. LA 4526 - Inventory (Hsinchu09)

8. LA 4643 - Twenty Questions (Tokyo09)

• DP on Tree

1. UVa 10243 - Fire! Fire!! Fire!!! (Min Vertex Cover ≈ Max Independent Set on Tree)

2. UVa 11307 - Alternative Arborescence (Min Chromatic Sum, 6 colors are suﬃcient)

3. LA 3685 - Perfect Service (Kaohsiung06)

4. LA 3794 - Party at Hali-Bula (Tehran06)

5. LA 3797 - Bribing FIPA (Tehran06)

6. LA 3902 - Network (Seoul07)

7. LA 4141 - Disjoint Paths (Jakarta08)

**Collections Framework**

add(), addAll(), remove(), removeAll(), clear(), size(), contains(), containsAll(), isEmpty(), toArray(), clone(), removeAll()

**List Interface**

**Java.math.BigInteger, Java.math.BigDecimal**

BigInteger result = new BigInterger(“String”);

BigInteger result1 = BigInterger.ONE;

reslut1.multiply(result);;

BigInteger.valueOf();

The Java BigInteger (BI) class supports the following basic integer operations: addition – add(BI), subtraction – subtract(BI), multiplication – multiply(BI), division – divide(BI), remainder – remainder(BI), combination of division and remainder – divideAndRemainder(BI), modulo – mod(BI) (slightly diﬀerent to remainder(BI)), and power – pow(int exponent).

***Write code for all kinds of series, sequences, algorithms, data structures, calculations. (With Comments that describe exactly how it should be used)***

***Search for databases that already have these kinds of pre-written functions for Java***

**Fibonacci Series**, Triangle Numbers, **Factorial, sum of squares, sum of cubes,** **Seive of Eratosthenes,** Number of Prime factors of a number, Sum of prime factors of a number, Catalan Numbers, Other useful series and sequences, stars and bars, stars and bars with limitations, number of binary trees of **n** nodes, **Euclid’s algorithm**, **Binomial Coefficients**, Pascal’s Triangle, Generating functions coefficients, Sum of geometric series, arithmetic series, Number of permutations (with and without repetition and limitations), Putting things into groups, Search Algorithms, Sorting Algorithms, Data structure filling up algorithms

**Strategies for combinatorial problems**

* Use alphabets to represent events/variables and use the permutations and combinations of these to find the solution to some other physical problem.
* Use inequalities to represent constraints and shift the number of total objects and then compute the problem to the problem without constraints
* When there are repetitions with combinations, we take ((n+k-1),(k))

Dynamic Programming Generic with memoization(For different types of Dynamic Programming problems)

Djikstra’s Algorithm

Belman - Ford Algorithm

Ford-Fulkerson Algorithm

Floyd - Warshal Algorithm

Deapth First Search

Breadth First Search

Backtracking algorithm

Examples of using all kinds of data structures

**Reccursion for Fibonacci Series**

public class FibonacciWithMemoization {  
  
 public static long fibArray[]=new long[26];  
  
 public static long fibonacci(long n){  
 long fibValue=0;  
 if(n==0 ){  
 return 0;  
 }else if(n==1){  
 return 1;  
 }else if(fibArray[(int)n]!=0){  
 return fibArray[(int)n];  
 }else{  
 fibValue=fibonacci(n-1)+fibonacci(n-2);  
 fibArray[(int) n]=fibValue;  
 return fibValue;  
 }  
 }  
  
 public static void main(String args[]){  
 fibArray[0]=1;  
 fibArray[1]=1;  
 long preTime=System.currentTimeMillis();  
 System.out.println("Value of 25th number in Fibonacci series->"+fibonacci(25));  
 long postTime=System.currentTimeMillis();  
 System.out.println("Time taken to compute in milliseconds->"+(postTime-preTime));  
 }  
}  
Output for the above program:  
Value of 25th number in Fibonacci series->121393  
Time taken to compute in milliseconds->0

* Use the condition in the for loop efficiently
* char letters[] = sc.next().trim().toCharArray();
* Arrays.fill( counts, 0 );
* for( char letter : letters )  
   {  
   ++counts[(int)(letter-'a')];  
   }
* Arrays.sort( counts );
* primes[np++] = 4L;
* Yes, it is a shorthand form of

if (isHere)

count = getHereCount(index);

else

count = getAwayCount(index);

Topological Sort Algorithm

Acyclic Directed Graphs Only

Starting with any node that has not been visited, find out any nodes adjacent to the current node. Call the function recursively for each of the nodes adjacent to hte current node. Keep track of the recursion tree for each of the recursive calls. In a stack go on adding the recursion tree branches from the leaves to the nodes

***List of Graph Algorithm’s that need priority implementation***

Depth First Search

Basic Form and Application

Finding Connected Components in Undirected Graph

Flood Fill - Labeling the Connected Components

It runs like a virus in a body and finishes or makes visited all the nodes where ever it travels. Basic pseudo code involves three parts,

* Check if the node is valid
* Mark the node as visited in a visited[] array
* If it is valid then check all the adjacent nodes to see which ones are valid, and on each one of them call the DFS(x,y) function recursively in a for loop.
* Increase the count of a variable each time DFS(x,y) finishes killing one contigous block valid nodes.

<https://github.com/sfmunera/uva/blob/master/UVa10946_Youwantwhatfilled.java>

<https://github.com/sfmunera/uva/blob/master/UVa00657_Thedieiscast.java>

<https://github.com/sfmunera/uva/blob/master/UVa00469_WetlandsofFlorida.java>

<https://github.com/joker23/ACM/blob/master/uva/352.java>

Graph Edges Property Check via DFS Spanning Tree

Finding Articulation Points and Bridges

Finding Strongly Connected Components in Directed Graph

Topological Sort (on a Directed Acyclic Graph)

Breadth First Search

Basic Form and Application

Single-Source Shortest Paths (SSSP) on Unweighted Graph

Kruskal’s Algorithm (To find Minimal Spanning Tree)

Basic Form and Application

‘Maximum’ Spanning Tree

Partial ‘Minimum’ Spanning Tree

Minimum Spanning ‘Forest’

Second Best Spanning Tree

Dijkstra’s Algorithm (Single Source Shortest Paths)

Bellman-Ford Algorithm (Negative Weighted Graph where Dijkstra’s Alg fails)

For negative weighted graphs

Floyd-Warshall Algorithm

Basic Form and Application

Transitive Closure (Warshall’s algorithm)

Minimax and Maximin

Edmonds Karp’s

Basic Form and Application

Min Cut

Multi-source Multi-sink Max Flow

Max Flow with Vertex Capacities

Max Independent Paths

Max Edge-Disjoint Paths

Min Cost (Max) Flow

Kosaraju’s algorithm for ﬁnding Strongly Connected Component, Prim’s and Boruvka’s algorithms for Minimum Spanning Tree, k-th shortest paths, Euler’s Path/Tour, Fleury’s algorithm, Chinese Postman Problem, Hamiltonian Path/Tour, Bitonic Traveling Salesman Problem, Arborescence, Tarjan’s Oﬄine Lowest Common Ancestor, Dinic’s or Push Relabel algorithms for Max Flow, Circulation Problem, Kuhn Munkres’s (Hungarian) matching algorithm, Edmonds’s Blossom Shrinking, etc.

**The seive of Eratosthenes**

The number of primes≤ M – denoted by π(M) – is bounded by O(M/(ln(M)−1)), so the complexity of this prime testing function is about O(√N/ln(√N)). This algorithm does approximately (N× (1/2 + 1/3 + 1/5 + 1/7 + ... + 1/last prime in range ≤ N)) operations. Using ‘sum of reciprocals of primes up to n’, we end up with the time complexity of roughly O(N loglogN) [44]. Since generating a list of small primes ≤ 10K using the sieve is fast (our library code below can go up to 107 under contest setting), we opt sieve for smaller primes and reserve optimized prime testing function for larger primes – see previous discussion. The combined code is as follows:

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Problems on Graph Theory

UVa 383 - Shipping Routes

UVa 11631 - Dark Roads

. UVa 11513 - 9 Puzzle

UVa 352 - Seasonal War

UVa 11709 - Trust Groups

UVa 872 - Ordering

UVa 341 - Non-Stop Travel

UVa 785 - Grid Colouring

UVa 10034 - Freckles

. UVa 11352 - Crazy King

UVa 10307 - Killing Aliens in Borg Maze

LA 3290 - Invite Your Friends

UVa 10369 - Arctic Networks

. UVa 11730 - Number Transformation

UVa 200 - Rare Order

UVa 336 - A Node Too Far

UVa 10009 - All Roads Lead Where?

UVa 784 - Maze Exploration

UVa 10147 - Highways

UVa 11228 - Transportation System

UVa 928 - Eternal Truths

. UVa 10102 - The Path in the Colored Field

UVa 924 - Spreading the News

UVa 11110 - Equidivisions

UVa 532 - Dungeon Master

UVa 10610 - Gopher and Hawks

UVa 11770 - Lighting Away

UVa 10150 - Doublets

UVa 11101 - Mall Mania

UVa 417 - Word Index

UVa 10336 - Rank the Languages

UVa 852 - Deciding victory in Go

UVa 260 - Il Gioco dell’X

UVa 908 - Re-connecting Computer Sites

UVa 10305 - Ordering Tasks

UVa 10842 - Traﬃc Flow

UVa 469 - Wetlands of Florida

UVa 657 - The Die is Cast

. UVa 11518 - Dominos 2

POJ 1679 - The Unique MST

UVa 11710 - Expensive Subway

LA 3678 - The Bug Sensor Problem

UVa 572 - Oil Deposits

UVa 10946 - You want what ﬁlled?

UVa 11733 - Airports

UVa 11749 - Poor Trade Advisor

UVa 459 - Graph Connectivity

UVa 762 - We Ship Cheap

UVa 11080 - Place the Guards

.UVa 10600 - ACM Contest and Blackout

UVa 315 - Network

. UVa 11049 - Basic Wall Maze

UVa 10926 - How Many Dependencies?

UVa 610 - Street Directions

UVa 10397 - Connect the Campus

UVa 10199 - Tourist Guide

UVa 11635 - Hotel Booking

UVa 534 - Frogger

UVa 10793 - The Orc Attack

UVa 10803 - Thunder Mountain

UVa 753 - A Plug for Unix

UVa 11045 - My T-Shirt Suits Me

UVa 10067 - Playing with Wheels

UVa 796 - Critical Links

UVa 11418 - Clever Naming Patterns

UVa 124 - Following Orders

UVa 429 - Word Transformation

UVa 10422 - Knights in FEN

UVa 627 - The Net (must print the path)

UVa 10731 - Test

UVa 10246 - Asterix and Obelix

UVa 10048 - Audiophobia

UVa 11492 - Babel (model the graph carefully!)

UVa 11747 - Heavy Cycle Edges

LA 4645 - Infected Land

UVa 11624 - Fire!

UVa 186 - Trip Routing

UVa 11377 - Airport Setup (model the graph carefully!)

UVa 11545 - Avoiding Jungle in the Dark

UVa 10044 - Erdos numbers

UVa 10653 - Bombs! NO they are Mines!!

LA 3290 - Invite Your Friends

UVa 782 - Countour Painting

UVa 10959 - The Party, Part I

UVa 11792 - Krochanska is Here!

UVa 11504 - Dominos

UVa 10724 - Road Construction

UVa 10603 - Fill

UVa 10004 - Bicoloring

UVa 10075 - Airlines

UVa 10801 - Lift Hopping (model the graph carefully!)

UVa 567 - Risk

UVa 10986 - Sending email

UVa 334 - Identifying Concurrent Events

UVa 439 - Knight Moves

LA 4408 - Unlock the Lock

UVa 10735 - Euler Circuit

UVa 10557 - XYZZY

UVa 929 - Number Maze

UVa 544 - Heavy Cargo

UVa 563 - Crimewave

UVa 10330 - Power Transmission

UVa 10308 - Roads in the North

UVa 11280 - Flying to Fredericton

UVa 11015 - 05-32 Rendezvous

UVa 341 - Non-Stop Travel

UVa 423 - MPI Maelstrom

UVa 869 - Airline Comparison

UVa 10099 - Tourist Guide

UVa 11506 - Angry Programmer

UVa 11463 - Commandos

UVa 821 - Page Hopping

UVa 558 - Wormholes

LA 2523 - Machine Schedule

UVa 10171 - Meeting Prof. Miguel

UVa 10594 - Data Flow

UVa 112 - Tree Summing

UVa 10000 - Longest Paths

UVa 122 - Trees on the level

UVa 10806 - Dijkstra, Dijkstra.

UVa 615 - Is It A Tree?

UVa 699 - The Falling Leaves

UVa 712 - S-Trees

UVa 536 - Tree Recovery

LA 4138 - Anti Brute Force Lock

LA 4637 - Repeated Substitution with Sed

LA 4110 - RACING

UVa 10701 - Pre, in and post

UVa 10938 - Flea Circus

UVa 11695 - Flight Planning

UVa 10459 - The Tree Root

UVa 10511 - Councilling

UVa 10166 - Travel

UVa 103 - Stacking Boxes

UVa 10029 - Edit Step Ladders

LA 4407 - Gun Fight

UVa 10080 - Gopher II

UVa 10092 - The Problem with the Problemsetter

UVa 10350 - Liftless Eme

UVa 11324 - The Largest Clique

UVa 926 - Walking Around Wisely

LA 2696 - Air Raid

LA 3126 - Taxi Cab Scheme

UVa 670 - The Dog Task

Top Coder Open 2003 Semiﬁnal Round 4 - Division 1, Level 3 - RookAttack

Top Coder Open 2009: Qualifying 1 - Prime Pairs

LA 3294 - The Ultimate Bamboo Eater

UVa 988 - Many paths, one destination

Ural/Timus OJ 1450 - Russian pipelines

UVa 115 - Climbing Trees

UVa 820 - Internet Bandwidth

UVa 10480 - Sabotage

UVa 925 - No more prerequisites, please!

UVa 10779 - Collector’s Problem

UVa 10278 - Fire Station

PKU 3160 - Father Christmas ﬂymouse

UVa 825 - Walking on the Safe Side

UVa 11159 - Factors and Multiples (similar solution with Bipartite Matching above)

UVa 11419 - SAM I AM

PKU 2226 - Muddy Fields

LA 3487 - Duopoly (also in Zhejiang Online Judge problem 2129) • Max Vertex Cover in Bipartite Graph