|  |
| --- |
|  |
| NOTES |
| MD. SHAHADAT HOSSAIN SHAHIN |
|  |
| UNIVERSITY OF DHAKA |
|  |

Table of Contents

[Template 2](#_Toc520941338)

[Miscellaneous Formulas 3](#_Toc520941339)

[Maximum Flow Notes 7](#_Toc520941340)

[BPM Notes 9](#_Toc520941341)

[Posets (Partially Ordered Sets) 9](#_Toc520941342)

[Antichain 9](#_Toc520941343)

[Chain 9](#_Toc520941344)

[Mirsky’s Theorem 9](#_Toc520941345)

[Dilworth’s Theorem 9](#_Toc520941346)

[Konig’s Theorem 9](#_Toc520941347)

[Finding Minimum Vertex Cover 10](#_Toc520941348)

[Node Disjoint Path Cover 10](#_Toc520941349)

[General Path Covers 10](#_Toc520941350)

[Weighted Minimum Vertex Cover 10](#_Toc520941351)

[Sorting an Array with Minimum Moves 12](#_Toc520941352)

[Type 1 12](#_Toc520941353)

[Moves 12](#_Toc520941354)

[Solution 12](#_Toc520941355)

[Type 2 12](#_Toc520941356)

[Moves 12](#_Toc520941357)

[Solution 12](#_Toc520941358)

[Type 3 12](#_Toc520941359)

[Move 12](#_Toc520941360)

[Solution 12](#_Toc520941361)

[Game Theory 12](#_Toc520941362)

[Impartial Game 12](#_Toc520941363)

[Nim Game 13](#_Toc520941364)

[Turning Turtles 13](#_Toc520941365)

[Grundy Number 13](#_Toc520941366)

[Colon principle 13](#_Toc520941367)

[Fusion Principle 14](#_Toc520941368)

Miscellaneous Formulas

* If

Then,

* If

Then,

* Number of ways of forming teams from members where each consists of members () is

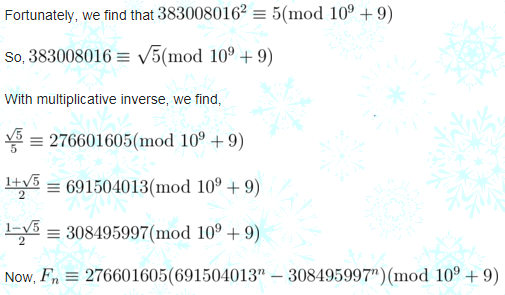
Ways of choosing members from and making one of them captain

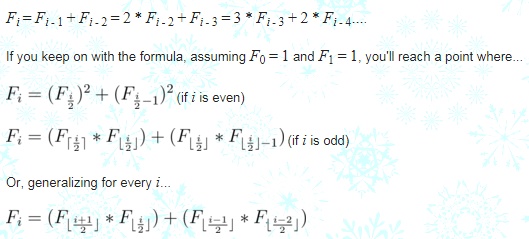
Ways of choosing members from and making one of them captain(k is not fixed here)

* Number of ways to tile a grid with two types of tiles() is

Hockey Stick Pattern

* is always a multiple of
* GCD() =

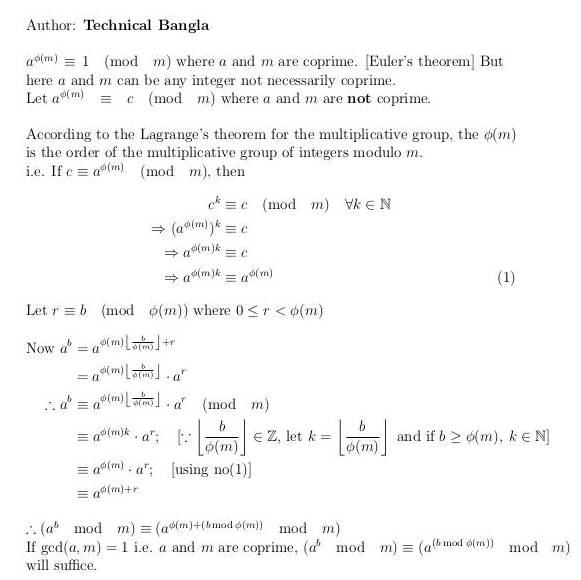




* because is a multiple of
* where and are pair wise co prime
* if is a prime
* Modular inverse of with respect to is
* If then has a square root modulo

Else and has no square root modulo

* where
* If then where is a divisor of
* Let and . If then
* If then
* implies that



Maximum Flow Notes

**There might be both penalties and rewards in same problem. Try to convert everything into penalties and find a min cut solution. Example : Image Segmentation Problem.**

#### Maximum Flow With Demands

Suppose we have multiple sources and multiple sinks. Each sink wants to get a certain amount of ow

(its demand). Each source has a certain amount of ow to give (its supply). We can represent supply as negative demand.

* Capacity constraints: For each e ϵ E, 0 ≤ f(e) ≤ ce
* For each v ϵ V, fin(v) – fout(v) = dv

Let S be the set of nodes with negative demands (supply). Let T be the set of nodes with positive demands (demand). In order for there to be a feasible ow, we must have:

How can we turn the circulation with demands problem into the maximum ow problem?

* Add a new source s\* with an edge (s\*, s) from s\* to every node s ϵ S.
* Add a new sink t\* with an edge (t, t\*) from every node t ϵ T to t\*.
* cap(s\*, s) = -ds
* cap(t, t\*) = dt

#### Finding Feasible Flow With Lower Bound on Edges

* Create a super source(s’), a super sink(t’). If edge u🡪v has a lower bound of LB,

1. Give an edge from s’ to v with capacity LB.
2. Give an edge from u to t’ with capacity LB.
3. Change the capacity of edge u🡪v to (C – LB)

* Give an edge from normal sink to normal source with capacity infinity (If source and sink are specified in the original network, otherwise ignore that edge)

If maxflow is equal to sum of LBs , then the lower bound can be satisfied.

Actual flow trough edge u🡪v is = LB + Flow through u🡪v in the modified network. (Add edge id to max flow edge structure, that will help to keep track of the flow through any edge)

#### Minimum Flow That Satisfies Lower Bound

Do a binary search on the capacity of the edge from actual sink to actual source.

#### Maximum Flow That Satisfies Lower Bound

Let the previous souce node be X. Let’s build a new graph with one extra node. Let it be node Y(new source). Add an edge from Y to X. Let capacity of the edge be INF and lower bound be **d**. Do a binary search on the maximum possible value of **d** that satisfies all the lower bounds.  
While trying to satisfy lower bounds, give an infinite edge from the actual source to node **Y**(not X).

#### Finding the Edges of Minimum Cut

After running max flow, do a DFS from source on the residual graph. Then an edge from u to v will be in the minimum cut if and only if

* u is visited and v is unvisited
* u is unvisited and v is visited

#### Finding Minimum Cut with minimum Number of Edges

Multiply the capacity of each edge by some constant T. Increase all the capacities by one. The minimum cut int the new graph is the minimum cut of the original graph with least number of edges.  
Keeping T greater than the number of edges in the graph is safe enough.

#### Project Selection Problem

* Maximize total profit.
* Doing i'th project profits you by P[i].
* Doing i'th project requires you to buy a list of instruments each of which has different cost.
* Different projects may require the same instrument in which case, buying one instrument is ok.
* Make a flow graph with projects on the left and instruments on the right.
* cap[source][i'th project] = P[i]
* cap[j'th instrument][sink] = Cost[j]
* cap[i'th project][j'th instrument] = inf ( if i'th project requires j'th instrument )

here mincut will minimize this function ( sacrifice profits of projects + cost of instruments to be bought )  
So ans is = Total profit of all projects - mincut.

#### Image Segmentation Problem

There are n pixels. Each pixel i can be assigned a foreground value f[i] or a background value b[i]. There is a penalty of p[i][j] if pixels i, j are adjacent and have different assignments. The problem is to assign pixels to foreground or background such that the sum of their values minus the penalties is maximum.

Let P be the set of pixels assigned to foreground and Q be the set of points assigned to background, then the problem can be formulated as

* maximize ( totalF + totalB - sacrificeFore for Q - sacrificeBack for P - penalty pij)
* or, minimize( sacrificeFore for Q + sacrificeBack for P + penalty pij)

The above minimization problem can be formulated as a minimum-cut problem by constructing a network,

source 🡪 node i with capacity f[i]  
node i 🡪 sink with capacity b[i]  
Undirected edge between node i and node j with capacity p[i][j]

BPM Notes

### BPM Notes

#### Posets (Partially Ordered Sets)

A DAG basically. If there is an edge from node u to node v, then node u and node v are related (For example u <= v )

#### Antichain

An **antichain** is a set of nodes of a graph such that there is no path from any node to another node using the edges of the graph. (Independent set)

#### Chain

Any path in the DAG.

#### Mirsky’s Theorem

For the nodes of A DAG,  
Minimum number of partitions into antichains = Maximum Chain (Longest path in the DAG)

##### Construction

Let’s suppose , L = number of nodes in the longest path.  
Let lp[u] denote the length of the longest path that ends at node u. For a pair of nodes u and v, if lp[u] = lp[v] then we can easily prove that there is no edge between them. There are atmost L distinct values of lp, so we need at most L partitions.

#### Dilworth’s Theorem

For the nodes of A DAG,  
Minimum partition into chain = Maximum length anti chain(Maximum Independent Set of a DAG)  
or, Minimum General Path Cover = Maximum length anti chain(Maximum Independent Set of a DAG)

#### Konig’s Theorem

A minimum vertex cover of a graph is a minimum set of nodes **S** such that each edge og the graph has at least one end point in **S**.  
Minimum Vertex Cover = Maximum Matching

**Maximum independent set** of a graph is a maximum set of nodes such that there is no path from one node to another of that set (Maximum Length Antichain)

Maximum Independent Set = Total Nodes – Minimum Vertex Cover

#### Finding Minimum Vertex Cover

See the min cut solution of the weighted version or do the following.  
Run BPM. After that do a dfs from every **unmatched** node in the left side. Go from u to v if and only if :

* u is on the left side and matchL[u] != v or
* u is on the right side and matchR[u] == v

After this, all unvisited nodes in the left side and the visited nodes in the right side will form a MVC.

See the proof of Konig’s theorem.

The rest of the nodes will form Maximum Independent Set then.

#### Node Disjoint Path Cover

Each node belongs to exactly one path  
We can find a minimum node-disjoint path cover by constructing a matching graph where each node of the original graph is represented by two nodes: a left node and a right node. There is an edge from a left node to a right node if there is a such an edge in the original graph.

Minimum node disjoint path cover = Total nodes(N) – maximum matching

Let’s assume we need N paths initially. Each match reduces the number of needed paths by one.

#### General Path Covers

A **general path cover** is a path cover where a node can belong to more than one path.

A minimum general path cover can be found almost like a minimum nodedisjoint path cover. It suffices to add some new edges to the matching graph so that there is an edge a! b always when there is a path from a to b in the original graph (possibly through several edges).

#### Weighted Minimum Vertex Cover

Sum of the weights of the vertices in the cover should be minimum.  
In the bipartite graph, make all the edges directed from left to right and set their capacity to INF. Define a source node src and a sink node snk.  
Give an edge from src every node in the left side where the capacity of that edge is the weight of that node.  
Give an edge from every node in the right side to snk in the same way.  
Find Min cut(Max Flow) of the graph. This will be the minimum vertex cover weight.

Sum of weights of all the nodes – Min vertex cover = Maximum Independent set with maximum weight

**If all the nodes have weight 1, then this finds the MVC for unweighted version**

**For any graph,**  
**∑Weights of the nodes in MVC + ∑Weights of the nodes in Maximum Independent Set  
= ∑Weights of all the nodes**

Sorting an Array with Minimum Moves

## Type 1

### Moves

Taking an element and inserting it in the **front** of the array.

### Solution

The idea is to traverse array from end. We expect n at the end, so we initialize expectedItem as n. All the items which are between actual position of expectedItem and current position must be moved to front. So, we calculate the number of items between current item and expected item. Once we find expectedItem, we look for next expectedItem by reducing expectedITem by one.

## Type 2

### Moves

Taking an element and inserting it in **any position** of the array.

### Solution

Number of elements in the array – LIS

## Type 3

### Move

Swapping two elements

### Solution

This can be easily done by visualizing the problem as a graph. We will have **n** nodes and an edge directed from node **i** to node **j** if the element at i’th index must be present at j’th index in the sorted array. The graph will now contain many non-intersecting cycles. Now a cycle with 2 nodes will only require 1 swap to reach the correct ordering, similarly a cycle with 3 nodes will only require 2 swaps to do so.

Hence,

ans = **Σ** cycle\_size – number of cycles

Game Theory

## Impartial Game

An [impartial game](https://en.wikipedia.org/wiki/Impartial_game) is one such as [nim](https://en.wikipedia.org/wiki/Nim), in which each player has exactly the same available moves as the other player in any position.

## Nim Game

N piles each containing some stones. He who removes the last stones wins.

Solution : If xor sum of the number of stones in the piles>0, the first player will win, otherwise the second player.

## Turning Turtles

Given a horizontal line of N coins with some coins showing heads and some tails. Each turn, a player has to flip one coin from head to tail, and in the same time (if he/she wants), flip one more coin to the left of it.

Solution : This game is equivalent to Nim Game with each coin showing head in kth position equals to a pile of k stones.

## Grundy Number

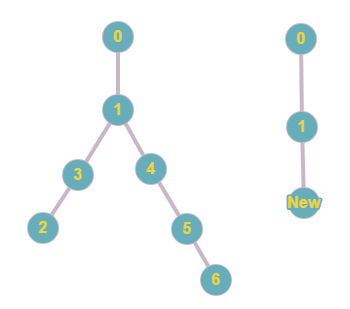
Grundy number of a losing state is 0.

Grundy number of a state A is the mex of the following set :

{ x : x is the grundy number of some state which is reachable from the state A by a single move }

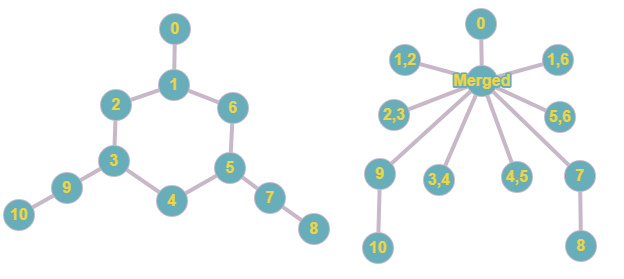
If we know the grundy number for every part of a composite game, the winner will be the first player if the xor sum of the grundy numbers is greater than 0.

## Colon principle

****

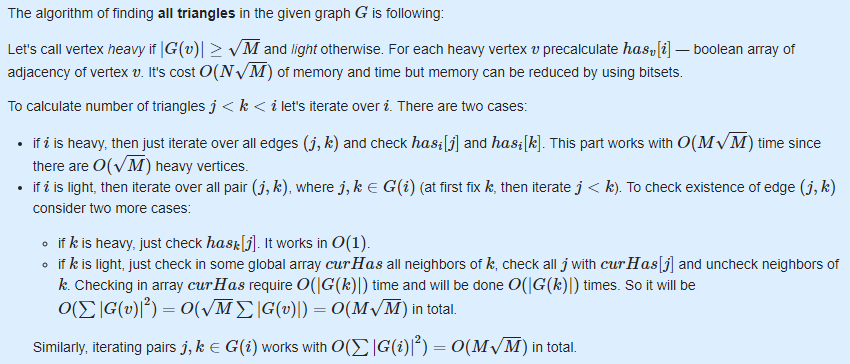
The trees in the image above are similar according to colon principle. Node 2 is 2 edges away from node 1 and node 6 is 3 edges away from node 1. So, we get a new node 2^3 = 1 edges away from node 1.

## Fusion Principle

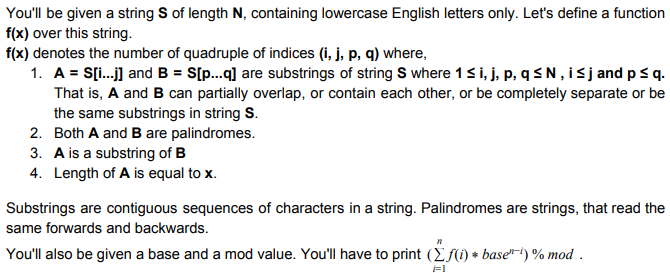
****

Node 1,2,3,4,5,6 are in a cycle cycle, they are merged to a single node. For every edge in the cycle a new node arises. Thus, we can get a tree from any graph and apply colon principle there.

Number of Triangles in a Graph



MIST 2019 F



Editorial:  Firstly let's discuss an O(N2) solution. Build the palindromic tree. Then generate and store the frequencies of all unique palindromes (read nodes) [Section 2.4, Proposition 3 of the original paper]. Now for each node v, traverse to all those nodes u, who are reachable from v via suffix link chains and tree parental chains. Here, obviously, palindrome stored at node u is a substring of palindrome stored at node v. So, we can now add freq[u]\*freq[v] to f(length(u)). Of course, we will also have to add freq[v]\*freq[v]to f(length(v)).

Now, how can we decrease the complexity, and move towards the desired solution? For that let's rewrite the previously discussed O(N2) solution. Previously we iterated over v, and looked for u. Now, we will iterate over u, and look for v. Who are v nodes? The nodes, from which, node u can be reached via suffix link chains and tree parental chains.

We can reach all nodes v, from node u, by checking the nodes in the subtrees of those nodes w, who marked u as their longest suffix palindrome. We will sum up the frequencies of palindromes of each node under all the subtrees beforehand. Then for all those w nodes, we will add freq[u]\*freq[p]to f(length(u)), where p is a node under the subtree of node w. To avoid overcount, we will mark discovery and finishing times of all nodes. We will sort all those w nodes, by discover time. Then we can easily check whether the current w node is already under the subtree of a previously checked w node. In this solution, we will also have to add freq[u]\*freq[u]to f(length(u)).

Let's talk about correctness. Let, x be a node, who marked u as its longest suffix palindrome. Let, y be another node, who marked x as its longest suffix palindrome. Also let x not be reachable from y via parental chain. So are we missing out all those nodes under node y, to mark as node v? No, it can be observed that there exists at least one node p, which can be reached via parental chain from node y, where node p marks node u as its longest suffix palindrome.

**Time Complexity:** As each node marks exactly one node as its longest suffix palindrome, so over all u nodes, the total number of w nodes will be N. We are sorting these w nodes, using discovery time, which costs O(N log N). And that is the complexity of this solution. As you can see, the complexity is mostly dominated by the sorting algorithm used. By using a linear sorting algorithm, we can even reach a linear time solution for this problem.

Our solution runs well under 1s. We offered 10s time limit to encourage any other interesting approaches, with slightly higher complexity. Unfortunately, nobody came close to a valid solution during the onsite contest or the online replay.