ICPC for Schools – Team Cookbook

Team name: import_solution

Team members: Aadarsh A (Std 11)

Team institution: The PSBB Millennium School, Gerugambakkam

Contest Template:

```
import atexit
import io
import sys

_INPUT_LINES = sys.stdin.read().splitlines()

input = iter(_INPUT_LINES).__next__

_OUTPUT_BUFFER = io.StringIO()

sys.stdout = _OUTPUT_BUFFER

@atexit.register
def write():
	sys.__stdout__.write(_OUTPUT_BUFFE
R.getvalue())

def main():
	#do something

if __name__==”__main__”:
	main()
```

Data Structures:

1. **Binary Search Tree** –

```
class Node(object):
 def __init__(self, d):
  self.data = d
  self.left = None
  self.right = None
 def insert(self, d):
  if self.data == d:
    return False
  elif d < self.data:
   if self.left:
    return self.left.insert(d)
   else:
    self.left = Node(d)
    return True
  else:
    if self.right:
     return self.right.insert(d)
    else:
     self.right = Node(d)
     return True
 def find(self, d):
   if self.data == d:
    return True
   elif d < self.data and self.left:
    return self.left.find(d)
   elif d > self.data and self.right:
    return self.right.find(d)
   return False
 def preorder(self, l):
   l.append(self.data)
   if self.left:
    self.left.preorder(l)
   if self.right:
    self.right.preorder(l)
   return l
 def postorder(self, l):
   if self.left:
    self.left.postorder(l)
   if self.right:
    self.right.postorder(l)
   l.append(self.data)
   return l
 def inorder(self, l):
   if self.left:
    self.left.inorder(l)
   l.append(self.data)
   if self.right:
    self.right.inorder(l)
   return l

class BST(object):
 def __init__(self):
  self.root = None
 def insert(self, d):
  if self.root:
   return self.root.insert(d)
  else:
   self.root = Node(d)
   return True
 def find(self, d):
  if self.root:
   return self.root.find(d)
  else:
   return False
 def remove(self, d):
  if self.root == None:
   return False
  if self.root.data == d:
   if self.root.left is None and self.root.right is None:
    self.root = None
    return True
   elif self.root.left and self.root.right is None:
    self.root = self.root.left
    return True
   elif self.root.left is None and self.root.right:
    self.root = self.root.right
    return True
   else:
    moveNode = self.root.right
    moveNodeParent = None
    while moveNode.left:
     moveNodeParent = moveNode
     moveNode = moveNode.left
    self.root.data = moveNode.data
    if moveNode.data < moveNodeParent.data:
     moveNodeParent.left = None
    else:
     moveNodeParent.right = None
    return True
  parent = None
  node = self.root
  while node and node.data != d:
   parent = node
   if d < node.data:
    node = node.left
   elif d > node.data:
    node = node.right
  if node == None or node.data != d:
   return False
```

```
  elif node.left is None and node.right is None:
    if d < parent.data:
      parent.left = None
    else:
      parent.right = None
    return True
  elif node.left and node.right is None:
    if d < parent.data:
      parent.left = node.left
    else:
      parent.right = node.left
    return True
  elif node.left is None and node.right:
    if d < parent.data:
      parent.left = node.right
    else:
      parent.right = node.right
    return True
  else:
    moveNodeParent = node
    moveNode = node.right
    while moveNode.left:
      moveNodeParent = moveNode
      moveNode = moveNode.left
    node.data = moveNode.data
    if moveNode.right:
      if moveNode.data < moveNodeParent.data:
        moveNodeParent.left = moveNode.right
      else:
        moveNodeParent.right = moveNode.right
    else:
      if moveNode.data < moveNodeParent.data:
        moveNodeParent.left = None
      else:
        moveNodeParent.right = None
    return True
def preorder(self):
  if self.root:
    return self.root.preorder([])
  else:
    return []
def postorder(self):
  if self.root:
    return self.root.postorder([])
  else:
    return []
def inorder(self):
  if self.root:
    return self.root.inorder([])
  else:
    return []
```

2. **Linked List (Insert and delete)** –

```
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None

class LinkedList:
  def __init__(self):
    self.head = None
  def printList(self):
    temp = self.head
    while (temp):
      print (temp.data,)
      temp = temp.next
  def insertFirst(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node
  def insertAfter(self, prev_node, new_data):
    if prev_node is None:
      print "The given previous node must in
LinkedList."
      return
    new_node = Node(new_data)
    new_node.next = prev_node.next
    prev_node.next = new_node
  def append(self, new_data):
      new_node = Node(new_data)
      if self.head is None:
        self.head = new_node
        return
      last = self.head
      while (last.next):
        last = last.next
      last.next =  new_node
  def deleteNode(self, key):
    temp = self.head
    if (temp is not None):
      if (temp.data == key):
        self.head = temp.next
        temp = None
        return
    while(temp is not None):
      if temp.data == key:
          break
      prev = temp
      temp = temp.next
    if(temp == None):
      return
    prev.next = temp.next
    temp = None
```

3. **Fenwick Tree** –

```
def getsum(BITTree,i):
 i = i+1
 while i > 0:
  s += BITTree[i]
  i -= i & (-i)
 return s
def updatebit(BITTree , n , i ,v):
 i += 1
 while i <= n:
  BITTree[i] += v
  i += i & (-i)
def construct(arr, n):
  BITTree = [0]*(n+1)
  for i in range(n):
   updatebit(BITTree, n, i, arr[i])
  return BITTree
```

4. **Graph and functions** –

```
class Graph(object):
 def __init__(self, graph_dict=None):
  #initializes a graph object
  if graph_dict == None:
   graph_dict = {}
  self.__graph_dict = graph_dict

 def vertices(self):
  """ returns the vertices of a graph """
  return list(self.__graph_dict.keys())

 def edges(self):
  """ returns the edges of a graph """
  return self.__generate_edges()

 def add_vertex(self, vertex):
  """If the vertex "vertex" is not in
  self.__graph_dict, a key "vertex" with an empty
  list as a value is added to the dictionary.
  Otherwise nothing has to be done.
  """
  if vertex not in self.__graph_dict:
   self.__graph_dict[vertex] = []

 def add_edge(self, edge):
  """assumes that edge is of type set, tuple or list
  between two vertices can be multiple edges!
  """
  edge = set(edge)
  vertex1 = edge.pop()
  if edge:
   # not a loop
   vertex2 = edge.pop()
  else:
   # a loop
   vertex2 = vertex1
  if vertex1 in self.__graph_dict:
   self.__graph_dict[vertex1].append(vertex2)
  else:
   self.__graph_dict[vertex1] = [vertex2]

 def __generate_edges(self):
  """ A static method generating the edges of the
  graph "graph". Edges are represented as sets
  with one (a loop back to the vertex) or two
  vertices """
  edges = []
  for vertex in self.__graph_dict:
   for neighbour in self.__graph_dict[vertex]:
    if {neighbour, vertex} not in edges:
     edges.append({vertex, neighbour})
  return edges

 def __str__(self):
  res = "vertices: "
  for k in self.__graph_dict:
   res += str(k) + " "
  res += "\nedges: "
  for edge in self.__generate_edges():
   res += str(edge) + " "
  return res

 def find_isolated_vertices(self):
  """ returns a list of isolated vertices. """
  graph = self.__graph_dict
  isolated = []
  for vertex in graph:
   print(isolated, vertex)
   if not graph[vertex]:
    isolated += [vertex]
  return isolated

 def find_path(self, start_vertex, end_vertex,
path=[]):
  """find a path from start_vertex to end_vertex
  in graph """
  graph = self.__graph_dict
  path = path + [start_vertex]
  if start_vertex == end_vertex:
   return path
  if start_vertex not in graph:
   return None
  for vertex in graph[start_vertex]:
```

```
    if vertex not in path:
      extended_path = self.find_path(vertex,
                   end_vertex,
                   path)
      if extended_path:
        return extended_path
    return None

  def find_all_paths(self, start_vertex, end_vertex,
path=[]):
    """ind all paths from start_vertex to
    end_vertex in graph """
    graph = self.__graph_dict
    path = path + [start_vertex]
    if start_vertex == end_vertex:
      return [path]
    if start_vertex not in graph:
      return []
    paths = []
    for vertex in graph[start_vertex]:
      if vertex not in path:
        extended_paths = self.find_all_paths(vertex,
                    end_vertex,
                    path)
        for p in extended_paths:
          paths.append(p)
    return paths

  def is_connected(self,
        vertices_encountered = None,
        start_vertex=None):
    """ determines if the graph is connected """
    if vertices_encountered is None:
      vertices_encountered = set()
    gdict = self.__graph_dict
    vertices = list(gdict.keys()) # "list" necessary in
Python 3
    if not start_vertex:
      # chosse a vertex from graph as a starting
point
      start_vertex = vertices[0]
    vertices_encountered.add(start_vertex)
    if len(vertices_encountered) != len(vertices):
      for vertex in gdict[start_vertex]:
        if vertex not in vertices_encountered:
          if self.is_connected(vertices_encountered,
vertex):
            return True
    else:
      return True
    return False

  def vertex_degree(self, vertex):
    """ The degree of a vertex is the number of
edges connecting
    it, i.e. the number of adjacent vertices. Loops
are counted
    double, i.e. every occurence of vertex in the list
    of adjacent vertices. """
    adj_vertices =  self.__graph_dict[vertex]
    degree = len(adj_vertices) +
adj_vertices.count(vertex)
    return degree

  def degree_sequence(self):
    """ calculates the degree sequence """
    seq = []
    for vertex in self.__graph_dict:
      seq.append(self.vertex_degree(vertex))
    seq.sort(reverse=True)
    return tuple(seq)

  @staticmethod
  def is_degree_sequence(sequence):
    """ Method returns True, if the sequence
"sequence" is a
    degree sequence, i.e. a non-increasing
sequence.
    Otherwise False is returned.
    """
    # check if the sequence sequence is non-
increasing:
    return all( x>=y for x, y in zip(sequence,
sequence[1:]))
  def delta(self):
    """ the minimum degree of the vertices """
    min = 100000000
    for vertex in self.__graph_dict:
      vertex_degree = self.vertex_degree(vertex)
      if vertex_degree < min:
        min = vertex_degree
    return min

  def Delta(self):
    """ the maximum degree of the vertices """
    max = 0
    for vertex in self.__graph_dict:
      vertex_degree = self.vertex_degree(vertex)
      if vertex_degree > max:
        max = vertex_degree
    return max

  def density(self):
```

```
    """ method to calculate the density of a graph
"""
    g = self.__graph_dict
    V = len(g.keys())
    E = len(self.edges())
    return 2.0 * E / (V *(V - 1))

  def diameter(self):
    """ calculates the diameter of the graph """
    v = self.vertices()
    pairs = [ (v[i],v[j]) for i in range(len(v)) for j in
range(i+1, len(v)-1)]
    smallest_paths = []
    for (s,e) in pairs:
      paths = self.find_all_paths(s,e)
      smallest = sorted(paths, key=len)[0]
      smallest_paths.append(smallest)
    smallest_paths.sort(key=len)
    diameter = len(smallest_paths[-1]) - 1
    return diameter
```

5. **Trie** –

```
class TrieNode:
  # Trie node class
  def __init__(self):
    self.children = [None]*26
    #EndofWord condition check
    self.isEndOfWord = False

class Trie:
  def __init__(self):
    self.root = self.getNode()

  def getNode(self):
    # Returns new trie node (initialized to NULLs)
    return TrieNode()

  def _charToIndex(self,ch):
    # private helper function
    # Converts key current character into index
    # use only 'a' through 'z' and lower case
    return ord(ch)-ord('a')

  def insert(self,key):
    # If not present, inserts key into trie
    # If the key is prefix of trie node,
    # just marks leaf node
    pCrawl = self.root
    length = len(key)
    for level in range(length):
      index = self._charToIndex(key[level])
      # if current character is not present
      if not pCrawl.children[index]:
        pCrawl.children[index] = self.getNode()
      pCrawl = pCrawl.children[index]
    # mark last node as leaf
    pCrawl.isEndOfWord = True

  def search(self, key):
    # Search key in the trie
    # Returns true if key presents
    # in trie, else false
    pCrawl = self.root
    length = len(key)
    for level in range(length):
      index = self._charToIndex(key[level])
      if not pCrawl.children[index]:
        return False
      pCrawl = pCrawl.children[index]

    return (pCrawl != None and
pCrawl.isEndOfWord)
```

String algorithms:

1. **KMP Algorithm** –

```
def KMPSearch(pat, txt):
  M = len(pat)
  N = len(txt)
  # create lps[] that will hold the longest prefix
suffix
  # values for pattern
  lps = [0]*M
  j = 0 # index for pat[]

  # Preprocess the pattern (calculate lps[] array)
  computeLPSArray(pat, M, lps)
  i = 0 # index for txt[]
  while i < N:
    if pat[j] == txt[i]:
      i += 1
      j += 1
    if j == M:
      print "Found pattern at index " + str(i-j)
      j = lps[j-1]
    # mismatch after j matches
    elif i < N and pat[j] != txt[i]:
      # Do not match lps[0..lps[j-1]] characters,
      # they will match anyway
      if j != 0:
        j = lps[j-1]
      else:
```

```
    i += 1

def computeLPSArray(pat, M, lps):
 len = 0
 lps[0]
 i = 1
 # the loop calculates lps[i] for i = 1 to M-1
 while i < M:
  if pat[i]== pat[len]:
   len += 1
   lps[i] = len
   i += 1
  else:
   if len != 0:
    len = lps[len-1]
   else:
    lps[i] = 0
    i += 1
```

Graph algorithms:

1. **BFS and DFS** –

```
def Graph():
 graph={}
 return(graph)
def addPath(n1,n2,graph):
 try:
  graph[n1].append(n2)
 except:
  graph[n1]=[n2]
 #Uncomment Below lines if Graph is Undirected
 '''try:
  graph[n2].append(n1)
 except:
  graph[n2]=[n1]'''

 return(graph)

def bfs(graph,start):
 n=len(graph)
 visited=[False]*n
 queue=[start]
 ans=[]

 while(queue!=[]):
  node=queue.pop(0)
  ans.append(node)
  visited[node]=True
  for i in graph[node]:
   if(visited[i]==False):
    queue.append(i)

 return(ans)

def dfs(graph,start):
 n=len(graph)
 visited=[False]*n
 stack=[start]
 ans=[]

 while(stack!=[]):
  node=stack.pop()
  ans.append(node)
  visited[node]=True
  for i in graph[node]:
   if(visited[i]==False):
    stack.append(i)

 return(ans)
```

2. **Prim's Algorithm (Adjacency Matrix)** -

```
import sys

class Graph():

 def __init__(self, vertices):
  self.V = vertices
  self.graph = [[0 for column in range(vertices)]
       for row in range(vertices)]

 def printMST(self, parent):
  print "Edge \tWeight"
  for i in range(1, self.V):
   print parent[i], "-", i, "\t", self.graph[i][
parent[i] ]

 # A utility function to find the vertex with
 # minimum distance value, from the set of
vertices
 # not yet included in shortest path tree
 def minKey(self, key, mstSet):

  # Initilaize min value
  min = sys.maxint

  for v in range(self.V):
   if key[v] < min and mstSet[v] == False:
    min = key[v]
    min_index = v

  return min_index
```

```
  def primMST(self):
    key = [sys.maxint] * self.V
    parent = [None] * self.V
    key[0] = 0
    mstSet = [False] * self.V

    parent[0] = -1

    for cout in range(self.V):

      # Pick the minimum distance vertex from
      # the set of vertices not yet processed.
      u = self.minKey(key, mstSet)

      # Put the minimum distance vertex in
      # the shortest path tree
      mstSet[u] = True

      # Update dist value of the adjacent vertices
      # of the picked vertex only if the current
      # distance is greater than new distance and
      # the vertex in not in the shotest path tree
      for v in range(self.V):
        if self.graph[u][v] > 0 and mstSet[v] == False
and key[v] > self.graph[u][v]:
            key[v] = self.graph[u][v]
            parent[v] = u

    self.printMST(parent)
```

3. **Kruskal's Algorithm (Dictionary/List)** –

```
import sys

class Graph():
  def __init__(self, vertices):
    self.V = vertices
    self.graph = [[0 for column in range(vertices)]
          for row in range(vertices)]

  def printMST(self, parent):
    print "Edge \tWeight"
    for i in range(1, self.V):
      print parent[i], "-", i, "\t", self.graph[i][
parent[i] ]

  # A utility function to find the vertex with
  # minimum distance value, from the set of
vertices
  # not yet included in shortest path tree
  def minKey(self, key, mstSet):

    # Initilaize min value
    min = sys.maxint

    for v in range(self.V):
      if key[v] < min and mstSet[v] == False:
        min = key[v]
        min_index = v

    return min_index

  # Function to construct and print MST for a
graph
  # represented using adjacency matrix
representation
  def primMST(self):
    key = [sys.maxint] * self.V
    parent = [None] * self.V
    key[0] = 0
    mstSet = [False] * self.V
    parent[0] = -1
    for cout in range(self.V):
      # Pick the minimum distance vertex from
      # the set of vertices not yet processed.
      u = self.minKey(key, mstSet)

      # Put the minimum distance vertex in
      # the shortest path tree
      mstSet[u] = True

      # Update dist value of the adjacent vertices
      # of the picked vertex only if the current
      # distance is greater than new distance and
      # the vertex in not in the shotest path tree
      for v in range(self.V):
        if self.graph[u][v] > 0 and mstSet[v] == False
and key[v] > self.graph[u][v]:
            key[v] = self.graph[u][v]
            parent[v] = u

    self.printMST(parent)
```

4. **Dijkstra's Algorithm (Adjacency Matrix)** –

```
import sys

class Graph():

  def __init__(self, vertices):
    self.V = vertices
    self.graph = [[0 for column in range(vertices)]
          for row in range(vertices)]
```

```
  def printSolution(self, dist):
    print "Vertex \tDistance from Source"
    for node in range(self.V):
      print node, "\t", dist[node]

  # A utility function to find the vertex with
  # minimum distance value, from the set of
vertices
  # not yet included in shortest path tree
  def minDistance(self, dist, sptSet):

    # Initialize minimum distance for next node
    min = sys.maxint

    # Search not nearest vertex not in the
    # shortest path tree
    for v in range(self.V):
      if dist[v] < min and sptSet[v] == False:
        min = dist[v]
        min_index = v

    return min_index

  # Function that implements Dijkstra's single
source
  # shortest path algorithm for a graph represented
  # using adjacency matrix representation
  def dijkstra(self, src):

    dist = [sys.maxint] * self.V
    dist[src] = 0
    sptSet = [False] * self.V

    for cout in range(self.V):

      # Pick the minimum distance vertex from
      # the set of vertices not yet processed.
      # u is always equal to src in first iteration
      u = self.minDistance(dist, sptSet)

      # Put the minimum distance vertex in the
      # shotest path tree
      sptSet[u] = True

      # Update dist value of the adjacent vertices
      # of the picked vertex only if the current
      # distance is greater than new distance and
      # the vertex in not in the shotest path tree
      for v in range(self.V):
        if (self.graph[u][v] > 0 and sptSet[v] ==
False and dist[v] > dist[u] + self.graph[u][v]):
          dist[v] = dist[u] + self.graph[u][v]

    self.printSolution(dist)
```

5. **Strongly Connected components (Adjacency List) –**

```
from collections import defaultdict

class Graph:
  def __init__(self,vertices):
    self.V= vertices
    self.graph = defaultdict(list)

  def addEdge(self,u,v):
    self.graph[u].append(v)

  # A function used by DFS
  def DFSUtil(self,v,visited):
    for i in self.graph[v]:
      if visited[i]==False:
        self.DFSUtil(i,visited)

  def fillOrder(self,v,visited, stack):
    for i in self.graph[v]:
      if visited[i]==False:
        self.fillOrder(i, visited, stack)
    stack = stack.append(v)

  # Function that returns reverse (or transpose) of
this graph
  def getTranspose(self):
    g = Graph(self.V)
    for i in self.graph:
      for j in self.graph[i]:
        g.addEdge(j,i)
    return g

  # The main function that finds and prints all
strongly connected components
  def printSCCs(self):
    stack = []
    # Mark all the vertices as not visited
    visited =[False]*(self.V)
    for i in range(self.V):
      if visited[i]==False:
        self.fillOrder(i, visited, stack)

    gr = self.getTranspose()
    visited =[False]*(self.V)

    while stack:
      i = stack.pop()
```

```
    if visited[i]==False:
      gr.DFSUtil(i, visited)
      print ("")
```

6. **Connected components of a graph (Adjacency List) –**

```
class Graph:
  def __init__(self,V):
    self.V = V
    self.adj = [[] for i in range(V)]

  def DFSUtil(self, temp, v, visited):
    visited[v] = True
    temp.append(v)
    for i in self.adj[v]:
      if visited[i] == False:
        temp = self.DFSUtil(temp, i, visited)
    return temp

  # method to add an undirected edge
  def addEdge(self, v, w):
    self.adj[v].append(w)
    self.adj[w].append(v)

  def connectedComponents(self):
    visited = []
    cc = []
    for i in range(self.V):
      visited.append(False)
    for v in range(self.V):
      if visited[v] == False:
        temp = []
        cc.append(self.DFSUtil(temp, v, visited))
    return cc
```

7. **Finding indegree and outdegree of all vertices in a graph (Adjacency List) –**

```
def findInOutDegree(adjList, n):
  _in = [0] * n
  out = [0] * n
  for i in range(0, len(adjList)):

    List = adjList[i]
    # Out degree for ith vertex will be the count
    # of direct paths from i to other vertices
    out[i] = len(List)
    for j in range(0, len(List)):

      # Every vertex that has
      # an incoming edge from i
      _in[List[j]] += 1

  print("Vertex\tIn\tOut")
  for k in range(0, n):
    print(str(k) + "\t" + str(_in[k]) +
          "\t" + str(out[k]))
```

8. **Topological Sort (Adjacency List) –**

```
from collections import defaultdict
class Graph:
  def __init__(self,vertices):
    self.graph = defaultdict(list)
    self.V = vertices

  def addEdge(self,u,v):
    self.graph[u].append(v)

  def topologicalSortUtil(self,v,visited,stack):
    visited[v] = True
    for i in self.graph[v]:
      if visited[i] == False:
        self.topologicalSortUtil(i,visited,stack)
    stack.insert(0,v)

  def topologicalSort(self):
    visited = [False]*self.V
    stack =[]
    for i in range(self.V):
      if visited[i] == False:
        self.topologicalSortUtil(i,visited,stack)
    print (stack)
```

9. **Floyd-Warshall Algorithm (Adjacency Matrix) –**

```
import sys
V = int(input())
# Define infinity as the large enough value. This
value will be
# used for vertices not connected to each other.
INF  = sys.maxsize
# Solves all pair shortest path via Floyd Warshall
Algorithm
def floydWarshall(graph):
  dist = map(lambda i : map(lambda j : j , i) ,
graph)
  for k in range(V):
    for i in range(V):
      for j in range(V):
        dist[i][j] = min(dist[i][j] ,
            dist[i][k]+ dist[k][j])
```

```
  printSolution(dist)

def printSolution(dist):
  for i in range(V):
    for j in range(V):
      if(dist[i][j] == INF):
        print ("%7s" %("INF"),)
      else:
        print ("%7d\t" %(dist[i][j]),)
      if j == V-1:
        print ("")
```

Dynamic Programming:

1. **Longest increasing subsequence –**

```
def CeilIndex(A, l, r, key):

  while (r - l > 1):
    m = l + (r - l)//2
    if (A[m] >= key):
      r = m
    else:
      l = m
  return r

def LongestIncreasingSubsequenceLength(A, size):

  tailTable = [0 for i in range(size + 1)]
  len = 0

  tailTable[0] = A[0]
  len = 1

  for i in range(1, size):

    if (A[i] < tailTable[0]):
      tailTable[0] = A[i]
    elif (A[i] > tailTable[len-1]):
      tailTable[len] = A[i]
      len+= 1
    else:
      tailTable[CeilIndex(tailTable, -1, len-1, A[i])] = A[i]

  return len
```

2. **Longest increasing subarray –**

```
def printLogestIncSubArr( arr, n) :
  m = 1
  l = 1
  maxIndex = 0
  for i in range(1, n) :
    if (arr[i] > arr[i-1]) :
      l =l + 1
    else :
      if (m < l)  :
        m = l
        maxIndex = i - m
      l = 1
  if (m < l) :
    m = l
    maxIndex = n - m
  for i in range(maxIndex, (m+maxIndex)) :
    print(arr[i] , end=" ")
```

3. **Knapsack problem –**

```
def knapSack(W, wt, val, n):
  K = [[0 for x in range(W+1)] for x in range(n+1)]

  # Build table K[][] in bottom up manner
  for i in range(n+1):
    for w in range(W+1):
      if i==0 or w==0:
        K[i][w] = 0
      elif wt[i-1] <= w:
        K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
      else:
        K[i][w] = K[i-1][w]

  return K[n][W]
```

4. **Maximum contiguous subarray:**

```
from sys import maxsize

def maxSubArraySum(a,size):

  max_so_far = -maxsize - 1
  max_ending_here = 0
  start = 0
  end = 0
  s = 0

  for i in range(0,size):

    max_ending_here += a[i]

    if max_so_far < max_ending_here:
```

```
    max_so_far = max_ending_here
    start = s
    end = i

  if max_ending_here < 0:
    max_ending_here = 0
    s = i+1

print ("Maximum contiguous sum, max_so_far)
print ("Starting Index", start)
print ("Ending Index", end)
```

5. Longest common subsequence –

```
def lcs(X, Y, m, n):
  L = [[0 for x in xrange(n+1)] for x in
xrange(m+1)]

  for i in xrange(m+1):
    for j in xrange(n+1):
      if i == 0 or j == 0:
        L[i][j] = 0
      elif X[i-1] == Y[j-1]:
        L[i][j] = L[i-1][j-1] + 1
      else:
        L[i][j] = max(L[i-1][j], L[i][j-1])

  index = L[m][n]

  lcs = [""] * (index+1)
  lcs[index] = ""

  i = m
  j = n
  while i > 0 and j > 0:

    if X[i-1] == Y[j-1]:
      lcs[index-1] = X[i-1]
      i-=1
      j-=1
      index-=1

    elif L[i-1][j] > L[i][j-1]:
      i-=1
    else:
      j-=1
  return ("".join(lcs))
```

6. Suffix array –

```
def suffix_array_best(s):
  n = len(s)
  k = 1
  line = to_int_keys_best(s)
  while max(line) < n - 1:

    line = to_int_keys_best(
      [a * (n + 1) + b + 1
      for (a, b) in
      zip_longest(line, islice(line, k, None),
          fillvalue=-1)])
    k <<= 1
  return line
```

Greedy Algorithms:

1. Activity selection:

```
def printMaxActivities(s , f ):
  n = len(f)
  i = 0
  print i,

  for j in xrange(n):
    if s[j] >= f[i]:
      print j,
      i = j
```

Searching Algorithms:

1. Binary Search:

```
def binarySearch (arr, l, r, x):
  if r >= l:
    mid = l + (r - l)/2
    if arr[mid] == x:
      return mid
    elif arr[mid] > x:
      return binarySearch(arr, l, mid-1, x)
    else:
      return binarySearch(arr, mid + 1, r, x)
  else:
    # Element is not present in the array
    return -1
```

2. Ternary Search:

```
def ternarySearch(l, r, key, ar):

  if (r >= l):
    mid1 = l + (r - l) //3
    mid2 = r - (r - l) //3
```

```
    if (ar[mid1] == key):
      return mid1
    if (ar[mid2] == key):
      return mid2
    if (key < ar[mid1]):
      return ternarySearch(l, mid1 - 1, key, ar)
    elif (key > ar[mid2]):
      return ternarySearch(mid2 + 1, r, key, ar)
    else:
      return ternarySearch(mid1 + 1,
            mid2 - 1, key, ar)

  # Key not found
  return -1
```

Math:

1. **Factors of a No:**

```
from functools import reduce

def factors(n):
    return set(reduce(list.__add__,
            ([i, n//i] for i in range(1, int(n**0.5) + 1)
if not n % i)))
```

2. **GCD:**

```
def GCD(arr):
    return(math.gcd(arr[0],arr[1]))
def gcdlist(arr):
    if len(arr) > 2:
        return reduce(lambda x,y: GCD([x,y]), arr)
   else:
        return(GCD(arr))
```

3. **Sieve of Erastothenes:**

```
MAX_SIZE = 1000001
isprime = [True] * MAX_SIZE
prime = []
SPF = [None] * (MAX_SIZE)
def manipulated_sieve(N):
  isprime[0] = isprime[1] = False

  for i in range(2, N):
    if isprime[i]==True:
      prime.append(i)
      SPF[i] = i
    j = 0
    while (j < len(prime) and i * prime[j] < N and
prime[j] <= SPF[i]):
      isprime[i * prime[j]] = False
      SPF[i * prime[j]] = prime[j]
      j += 1
```

4. **Fast nCr Function:**

```
import operator as op
from functools import reduce

def ncr(n, r):
    r = min(r, n-r)
    numer = reduce(op.mul, range(n, n-r, -1), 1)
    denom = reduce(op.mul, range(1, r+1), 1)
    return numer / denom
```

5. **Pascal Triangle:**

```
def pascal(n):
    line = [1]
    for k in range(n//2):
        line.append((line[k] * (n-k) // (k+1)))
    if(n%2==1):
        line=line+line[::-1]
    else:
        nline=line[:-1]
        line=line+nline[::-1]
    return line
```

6. **Wilson theorem:**

A natural number p > 1 is a prime number if and only if

$(p - 1)! \equiv -1 \mod p$ (OR)
$(p - 1)! \equiv (p-1) \mod p$

7. **Fermat's Little theorem:**

$a^{p-1} \equiv 1 \pmod p$
OR
$a^{p-1} \% p = 1$
Here a is not divisible by p.

8. **Chinese Remainder theorem:**

It states that there always exists a x that satisfies given congruence modulo. Used to find minimum x, when a list of numbers and their remainders when mod with x is given.

```
def inv(a, m) :

  m0 = m
  x0 = 0
  x1 = 1

  if (m == 1) :
    return 0

  while (a > 1) :
    q = a // m
    t = m
    m = a % m
    a = t
    t = x0
    x0 = x1 - q * x0
    x1 = t
  # Make x1 positive
  if (x1 < 0) :
    x1 = x1 + m0
  return x1

def findMinX(num, rem, k) :
  prod = 1
  for i in range(0, k) :
    prod = prod * num[i]
  result = 0
  for i in range(0,k):
    pp = prod // num[i]
    result = result + rem[i] * inv(pp, num[i]) * pp
  return result % prod
```

9. Euler Totient function (?(n)):

Finds the count of numbers in {1,2,3…n} that are relatively prime to n (i.e) the numbers whose GCD with n is 1.

Euler Theorem: $a^{?(n)} \equiv 1 \pmod{n}$

```
def phi(n):
  result = n
  p = 2

  while(p * p <= n):
    if (n % p == 0):
      while (n % p == 0):
        n = int(n / p)
      result -= int(result / p)
    p += 1
  if (n > 1):
    result -= int(result / n)
  return result
```

10. Convex Hull (Graham's scan):

```
from functools import reduce
def convex_hull_graham(points):
  TURN_LEFT, TURN_RIGHT, TURN_NONE = (1, -1, 0)

  def cmp(a, b):
    return (a > b) - (a < b)

  def turn(p, q, r):
    return cmp((q[0] - p[0])*(r[1] - p[1]) - (r[0] - p[0])*(q[1] - p[1]), 0)

  def _keep_left(hull, r):
    while (len(hull) > 1 and turn(hull[-2], hull[-1], r) != TURN_LEFT):
      hull.pop()
    if not len(hull) or hull[-1] != r:
      hull.append(r)
    return hull

  points = sorted(points)
  l = reduce(_keep_left, points, [])
  u = reduce(_keep_left, reversed(points), [])
  return l.extend(u[i] for i in range(1, len(u) - 1)) or l
```

11. Modular Exponentiation (Find x^y % p):

```
def power(x, y, p) :

  res = 1
  # Update x if it is more
  # than or equal to p
  x = x % p

  while (y > 0) :
    # If y is odd, multiply
    # x with result
    if ((y & 1) == 1) :
      res = (res * x) % p

    # y must be even now
    y = y >> 1      # y = y/2
    x = (x * x) % p

  return res
```

12. Fast Fibonacci:

```
def _fib(n):
    if n == 0:
        return (0, 1)
    else:
        a, b = _fib(n // 2)
        c = a * (b * 2 - a)
        d = a * a + b * b
        if n % 2 == 0:
            return (c, d)
        else:
            return (d, c + d)
```

13. Bit Manipulation techniques:

1. To multiply by 2^x : S = S<<x
2. To divide by 2^x : S = S>>x
3. To set jth bit : S|=(1<<j)
4. To check jth bit : T = S &(1<<j) (If T=0 not set else set)
5. To turn off jth bit : S&=~(1<<j)
6. To flip jth bit : S^=(1<<j)
7. To get value of LSB: T = (S &(-S)) (Gives 2^position)
8. To turn on all bits S = (1<<n) - 1
in a set of size n:

<u>Important packages to use –</u>

1. Heapq
2. Itertools
3. Bisect
4. Counter (in collections)
5. Defaultdict
6. Functools (reduce and @lru_cache)
7. Time (perf_counter)

```
t1_start=perf_counter()
t1_stop=perf_counter()
#do something
elapsed_time=t1_start-t1_stop
```

8. Regex (Cheat Sheet)

- ^ Matches the beginning of a line
- $ Matches the end of the line
- . Matches any character
- \s Matches whitespace
- \S Matches any non-whitespace character
- Repeats a character zero or more times
- *? Repeats a character zero or more times (non-greedy)
- + Repeats a character one or more times
- +? Repeats a character one or more times(non-greedy)
- [aeiou] Matches a single character in the listed set
- [^XYZ] Matches a single character not in the listed set
- [a-z0-9] The set of characters can include a range
- (Indicates where string extraction is to start
-) Indicates where string extraction is to end

<u>Game Theory:</u>

1. If nim-sum is non-zero, player starting first wins.
2. Mex: smallest non-negative number not present in a set.
3. Grundy=0 means game lost.
4. Grundy=mex (Minimum excludant) of all possible next states.
5. Sprague-Grundy theorem:
If a game consists of sub games (nim with multiple piles)
Calculate grundy number of each sub game (each pile)
Take xor of all grundy numbers:
If non-zero, player starting first wins.