Team Contest Reference

Crystal Math

Gabriëlle Zwaneveld, Sven Holtrop and Lucas Crijns

TU Delft

1 Modules

1.1 Python

- Priority queue → `heapq`
- Python base switching → `int(x, base=b)`
- Deque → `collections.deque`

1.2 C++

- `std::pair<T,G>, std::make_pair` → `#include <utility>`
- `std::vector<T>` → `#include <vector>`
- `std::queue<T>` (Priority queue) → `#include <queue>`

2 Mathematics

Formulas for Geometric Shapes

$$\begin{aligned}
\text{oppervlakte cirkel} &: \pi r^2 \\
\text{omtrek cirkel} &: \pi d \\
\text{oppervlakte ellips} &: \pi ab \\
\text{oppervlakte kegel} &: \pi r^2 + \pi r\sqrt{r^2 h^2} \\
\text{inhoud kegel} &: \frac{1}{3}\pi r^2 h \\
\text{oppervlakte bol} &: 4\pi r^2 \\
\text{inhoud bol} &: \frac{4}{3}\pi r^3 \\
\text{oppervlakte cillinder} &: 2\pi rh + 2\pi r^2 \\
\text{inhoud cilinder} &: \pi r^2 h
\end{aligned}$$

2.1 Trapezion

oppvervlakte trapezium: $\frac{a+b}{2}h$
Als de evenwijdige zijden a, b verschillende lengtes hebben, dan is h:

$$h = \frac{\sqrt{(-a+b+c+d)(a-b+c+d)(a-b+c-d)(a-b-c+d)}}{2|b-a|}$$

De lengtes van de diagonalen als $b > a$ de evenwijdige zijden zijn

$$p = \sqrt{\frac{ab^2 - a^2b - ac^2 + bd^2}{b-a}}$$
$$q = \sqrt{\frac{ab^2 - a^2b - ad^2 + bc^2}{b-a}}$$

2.2 Oppervlakte formules

Formule van Heron:

$$s = \frac{a+b+c}{2}$$
$$\sqrt{s(s-a)(s-b)(s-c)}$$

Bretschneider's formule:

$$s = \frac{a+b+c+d}{2}$$
$$\sqrt{(s-a)(s-b)(s-c)(s-d) - abcdcos^2\left(\frac{\alpha+\gamma}{2}\right)}$$

More Formulas

$$\text{least common multiple} : \text{lcm}(m,n) = \frac{|m \cdot n|}{\text{gcd(m, n)}}$$
$$\text{Catalan number} : C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^{n}\frac{n+k}{k}$$
$$\text{Catalan numbers} : C = \{1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796\}$$

Fibonacci Numbers

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584$$

- When we take a pairs of large consecutive Fibonacci numbers, we can approximate the golden ratio by dividing them.
- The sum of any ten consecutive Fibonacci numbers is divisible by 11.

- Two consecutive Fibonacci numbers are co-prime.
- The Fibonacci numbers in the composite-number (i.e. non-prime) positions are also composite numbers.

2.3 Vectoren

Cross product

$$a \times b = \begin{bmatrix} a_x \\ a_b \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$

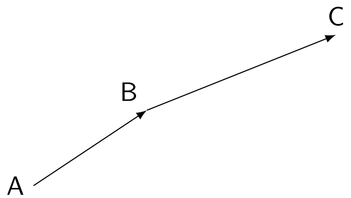
De projectie van een vector x op een andere vector y is:

$$\frac{\langle x, y \rangle}{\langle y, y \rangle} y$$

Het verschil van x en zijn projectie op y, staat loodrecht op y.
Twee vectoren x en y zijn loodrecht dan en slechts dan als $\langle x, y \rangle = 0$.

Projectie van een vector x op een vlak V. Bereken een normaal vector van V, bereken projectie p' van x op n. De projectie van x op V is dan x-p'.

Links of rechts ombuigen



$$\overrightarrow{AB} = \begin{bmatrix} p \\ q \end{bmatrix} \qquad \overrightarrow{n} = \begin{bmatrix} q \\ -p \end{bmatrix}$$

$\overrightarrow{n} \cdot \overrightarrow{BC} < 0 \Rightarrow$ linksaf

$\overrightarrow{n} \cdot \overrightarrow{BC} > 0 \Rightarrow$ rechtsaf

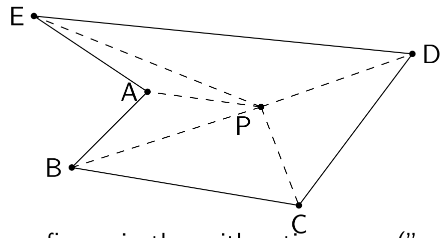
Punt in concaaf/convex polygon test

Tel het aantal doorsnijdingen van polygon met lijn P naar oneindig. Als het aantal doorsnijdingen oneven is, dan $P \in ABCDE$.

$\alpha = \angle APB + ... + \angle DPE + \angle EPA$

$\alpha = 0 \Rightarrow P \notin ABCDE$

$\alpha = 2\pi \Rightarrow P \in ABCDE$



Centroid of polygon

The centroid or geometric center of a plane figure is the arithmetic mean ("average") position of all the points in the shape. Informally, it is the point at which an infinitesimally thin cutout of the shape could be perfectly balanced on the tip of a pin.

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$A = \frac{1}{2} \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

Point to line distance

$$d(ax + by + c = 0, (x_0, y_0)) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

Number theory

$d(n)$ is het aantal positieve delers van een positief geheel getal n, met 1 en n inbegrepen.

$$d(1) + d(2) + \ldots + d(n) = \lfloor \frac{n}{1} \rfloor + \lfloor \frac{n}{2} \rfloor + \ldots + \lfloor \frac{n}{n} \rfloor$$

Aantal factoren van een priemgetal a in $n!$ is:

$$\lfloor \frac{n}{a} \rfloor + \lfloor \frac{n}{a^2} \rfloor + \ldots$$

Kleine stelling van Fermat, p priem en $a > 0$:

$$a^p \equiv a \mod p$$

Als a en p copriem zijn:

$$a^{p-1} \equiv 1 \mod p$$

Series and sums

Geometric series with r:

$$a + ar^2 + \ldots + ar^{n-1} = \sum_{k=0}^{n-1} ar^k = a\left(\frac{1-r^n}{1-r}\right)$$
$$\lim_{n\to\infty} S = \frac{a}{1-r} \qquad \forall |r| < 1$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

2.4 Lagrange polynomials

Given a set of (k+1) data points $(x_0, y_0), (x_1, y_1), \ldots (x_k, y_k)$ where no two x_j are the same. Find a polynomial of degree k that has all $(k+1)$ points

$$L(x) = \sum_{j=0}^{k} y_j l_j(x)$$
$$l_j(x) = \prod_{0\le m\le k, m\ne j} \frac{x - x_m}{x_j - x_m}$$

Graphs

For a connected planar graph, the number of vertices V, edges E and planar faces F obeys: $V - E + f = 2$ (this includes the outer face).

3 Algorithms

Extended Euclidean algorithm

```
def xgcd(a, b):
    """return (g, x, y) such that a*x + b*y = g = gcd(a, b)"""
    x0, x1, y0, y1 = 0, 1, 1, 0
    while a != 0:
        q, b, a = b // a, a, b % a
        y0, y1 = y1, y0 - q * y1
        x0, x1 = x1, x0 - q * x1
    return b, x0, y0
```

Hopcroft-Karp bipartite max-cardinality matching and max independent set

```
def bipartiteMatch(graph):
    '''Find maximum cardinality matching of a bipartite graph (U,V,E).
    The input format is a dictionary mapping members of U to a list
    of their neighbors in V.  The output is a triple (M,A,B) where M is a
    dictionary mapping members of V to their matches in U, A is the part
    of the maximum independent set in U, and B is the part of the MIS in V
    .
    The same object may occur in both U and V, and is treated as two
    distinct vertices if this happens.'''

    # initialize greedy matching (redundant, but faster than full search)
    matching = {}
    for u in graph:
        for v in graph[u]:
            if v not in matching:
                matching[v] = u
                break

    while 1:
        # structure residual graph into layers
        # pred[u] gives the neighbor in the previous layer for u in U
        # preds[v] gives a list of neighbors in the previous layer for v
    in V
        # unmatched gives a list of unmatched vertices in final layer of V
    ,
        # and is also used as a flag value for pred[u] when u is in the
    first layer
        preds = {}
        unmatched = []
        pred = dict([(u, unmatched) for u in graph])
        for v in matching:
            del pred[matching[v]]
        layer = list(pred)

        # repeatedly extend layering structure by another pair of layers
        while layer and not unmatched:
            newLayer = {}
            for u in layer:
                for v in graph[u]:
```

```python
                if v not in preds:
                    newLayer.setdefault(v, []).append(u)
        layer = []
        for v in newLayer:
            preds[v] = newLayer[v]
            if v in matching:
                layer.append(matching[v])
                pred[matching[v]] = v
            else:
                unmatched.append(v)

    # did we finish layering without finding any alternating paths?
    if not unmatched:
        unlayered = {}
        for u in graph:
            for v in graph[u]:
                if v not in preds:
                    unlayered[v] = None
        return (matching, list(pred), list(unlayered))

    # recursively search backward through layers to find alternating
paths
    # recursion returns true if found path, false otherwise
    def recurse(v):
        if v in preds:
            L = preds[v]
            del preds[v]
            for u in L:
                if u in pred:
                    pu = pred[u]
                    del pred[u]
                    if pu is unmatched or recurse(pu):
                        matching[v] = u
                        return 1
        return 0

    for v in unmatched: recurse(v)

print(bipartiteMatch({ 0:[4], 1:[2,3], 2:[0,1], 3:[0], 4:[1] }))
```

Newton-Raphson

Note: Algorithm to find zeros of an arbitrary function

```python
def derivative(f, x, h):
      return (f(x+h) - f(x-h)) / (2.0*h)  # might want to return a small
    non-zero if ==0

def quadratic(x):
    return 2*x*x-5*x+1     # just a function to show it works

def solve(f, x0, h):
    lastX = x0
    nextX = lastX + 10* h  # "different than lastX so loop starts OK
    while (abs(lastX - nextX) > h):  # this is how you terminate the loop
    - note use of abs()
        newY = f(nextX)                     # just for debug... see what
    happens
        print "f(", nextX, ") = ", newY     # print out progress... again
    just debug
        lastX = nextX
        nextX = lastX - newY / derivative(f, lastX, h)  # update estimate
    using N-R
    return nextX

xFound = solve(quadratic, 5, 0.01)    # call the solver
```

Kosaraju's algorithm

Note: Algorithm that finds all strongly connected components

```python
from collections import defaultdict
# This class represents a directed graph using adjacency list
    representation
class Graph:
    def __init__(self, vertices):
        self.V = vertices  # No. of vertices
        self.graph = defaultdict(list)  # default dictionary to store
    graph

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):
        # Mark the current node as visited and print it
        visited[v] = True
        print(v, end = " ")
        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)

    def fillOrder(self, v, visited, stack):
        # Mark the current node as visited
```

```
        visited[v] = True
        # Recur for all the vertices adjacent to this vertex
        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)

        # Recur for all the vertices adjacent to this vertex
        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 (For first DFS)
        visited = [False] * (self.V)
        # Fill vertices in stack according to their finishing
        # times
        for i in range(self.V):
            if visited[i] == False:
                self.fillOrder(i, visited, stack)

        # Create a reversed graph
        gr = self.getTranspose()

        # Mark all the vertices as not visited (For second DFS)
        visited = [False] * (self.V)

        # Now process all vertices in order defined by Stack
        counter = 0
        while stack:
            i = stack.pop()
            if visited[i] == False:
                gr.DFSUtil(i, visited)  # TO PRINT
                print("")               # TO PRINT
                counter += 1
        return counter # number of SCCs
```

```
# Create a graph given in the above diagram
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 1)
g.addEdge(0, 3)
g.addEdge(3, 4)

print("Following are strongly connected components " +
      "in given graph")
c = g.printSCCs()
print(c)
# prints out:
# 0 1 2
# 3
# 4
# 3 # is number of SCCs
```

Segment Tree

```
class SegmentTree:
    def __init__(s, l):
        s.n = len(l)
        s.tree = [0] * s.n + list(l)
        for i in range(s.n - 1, 0, -1):
            s.tree[i] = s.tree[2*i] + s.tree[2*i + 1]

    def update(s, p, val):
        p += s.n
        s.tree[p] = val
        while p > 1:
            p >>= 1
            s.tree[p] = s.tree[2*p] + s.tree[2*p + 1] #sum segment tree
            # s.tree[p] = max(s.tree[2*p], s.tree[2*p + 1]) # max segment
    tree


    def query(s, l, r): # interval [l, r)
        l += s.n; r += s.n
        t = 0
        while l < r:
            if l&1:
                t += s.tree[l] # r = max(res, tree[l])
                l += 1
            if r&1:
                r -= 1
                t += s.tree[r] # r = max(res, tree[r])
            l //= 2; r //= 2
```

```
        return t
```

Maximum subset sum (contiguous)

```
#Option 1:
def subsetsum(array,num):
    if num == 0 or num < 1:
        return None
    elif len(array) == 0:
        return None
    else:
        if array[0] == num:
            return [array[0]]
        else:
            with_v = subsetsum(array[1:],(num - array[0]))
            if with_v:
                return [array[0]] + with_v
            else:
                return subsetsum(array[1:],num)

#Option 2:
# use a binary number (represented as string) as a mask
def mask(lst, m):
    # pad number to create a valid selection mask
    # according to definition in the solution laid out
    m = m.zfill(len(lst))
    return map(lambda x: x[0], filter(lambda x: x[1] != '0', zip(lst, m)))

def subset_sum(lst, target):
    # there are 2^n binary numbers with length of the original list
    for i in range(2**len(lst)):
        # create the pick corresponsing to current number
        pick = mask(lst, bin(i)[2:])
        if sum(pick) == target:
            yield pick

# use 'list' to unpack the generator
from operator import itemgetter
#data = ((1,), (3,))
#map(itemgetter(0), data)
lst = subsetsum([1,2,3,4,5], 7)
print(lst)
```

Bellman Ford (Dijkstra with negative numbers)

Note: Pseudo code implementation

```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]
   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       distance[v] := inf              // At the beginning , all vertices
    have a weight of infinity
       predecessor[v] := null          // And a null predecessor

  distance[source] := 0               // The weight is zero at the source

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"
   return distance[], predecessor[]
```

Binary search

Note: Expects a sorted array

```
def bin_search(X, arr):
  low = 0
  high = len(arr)
  while (low < high):
    mid = (low + high)/2
    if arr[mid] == x:
      return mid
    elif X > arr[mid]:
      low = mid + 1
    else:
      high = mid
  return high
```

Chinese Remainder Theorem

Note: only works for coprime numbers

```python
from functools import reduce
def chinese_remainder(n, a):
    sum=0
    prod=reduce(lambda a, b: a*b, n)
    for n_i, a_i in zip(n,a):
        p=prod/n_i
        sum += a_i* mul_inv(p, n_i)*p
    return sum % prod


def mul_inv(a, b):
    b0= b
    x0, x1= 0,1
    if b== 1: return 1
    while a>1 :
        q=a// b
        a, b= b, a%b
        x0, x1=x1 -q *x0, x0
    if x1<0 : x1+= b0
    return x1
```

Coin sum

Note: Expects a list with the coin values, and repetitions if there are more coins with this value.

```python
def recMC(coinValueList,change):
   minCoins = change
   if change in coinValueList:
     return 1
   else:
      for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recMC(coinValueList,change-i)
         if numCoins < minCoins:
            minCoins = numCoins
   return minCoins
print(recMC([1,5,10,25],63))

def getWays(n, c):
    c.sort()
    arr = [0 for k in range(n + 1)]
    arr[0] = 1
    for coin in c:
        for i in range(coin, n + 1):
            arr[i] += arr[i - coin]
    return arr[n]
print(10, [1, 2, 5])
```

Dijkstra's algorithm

Note: nodes is a list with node i at index i and then this node contains a list of tuples that indicate the length of the edge and whereto.

```python
def dijkstra(a, b, nodes):
    import heapq
    visited = {}
    q = [(0, a)]
    distances = [float('inf') for _ in range(len(nodes))]
    distances[a] = 0
    while q:
        _, current = heapq.heappop(q)
        if current not in visited:
            visited.add(current)
            if current == b: return distances[b]
            for w, n in nodes[current]:
                if n in visited: continue
                if distances[n] > w + distances[current]:
                    distances[n] = w + distances[current]
                    heapq.heappush((distances[n], n))
    return float('inf')
```

Fenwick Tree

Note: Efficiently calculate sums and update elements. The sum is taken from 0 up to index r. Updates happen with deltas and not with setting.

```python
class FenwickSum:
    def __init__(self, items):
        self.items = [0 for _ in range(len(items))]
        self.size = len(items)
        for index, element in enumerate(items):
            self.update(index, element)
    def sum(self, r):
        if r < 0: return 0
        if r >= self.size: r = size - 1
        result = 0
        while r >= 0:
            result += self.items[r]
            r = (r & (r+1)) - 1
        return result
    def update(self, i, delta):
        while i < self.size:
            self.items[i] += delta
            i = i | (i+1)
```

Floyd-Warshall's algorithm

Note: The object dist is a weighted adjacency matrix

```
def floyd_warshall(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix)):
            for k in range(len(matrix)):
                matrix[j][k] = min(matrix[j][k], matrix[j][i] + matrix[i][
    k])
```

Primality check and factorisation

Note: Changing `return False` to `print(i)` shows all prime factors of n

```
def is_prime(n):
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False
    for i in range(3, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

Knapsack problem

```
def totalvalue(comb):
    ' Totalise a particular combination of items'
    totwt = totval = 0
    for item, wt, val in comb:
        totwt  += wt
        totval += val
    return (totval, -totwt) if totwt <= 400 else (0, 0)

items = (
    ("map", 9, 150), ("compass", 13, 35), ("water", 153, 200), ("sandwich"
    , 50, 160),
    ("glucose", 15, 60), ("tin", 68, 45), ("banana", 27, 60), ("apple",
    39, 40),
    ("cheese", 23, 30), ("beer", 52, 10), ("suntan cream", 11, 70), ("
    camera", 32, 30),
    ("t-shirt", 24, 15), ("trousers", 48, 10), ("umbrella", 73, 40),
    ("waterproof trousers", 42, 70), ("waterproof overclothes", 43, 75),
    ("note-case", 22, 80), ("sunglasses", 7, 20), ("towel", 18, 12),
    ("socks", 4, 50), ("book", 30, 10),
    )

def knapsack01_dp(items, limit):
    table = [[0 for w in range(limit + 1)] for j in xrange(len(items) + 1)
    ]

    for j in xrange(1, len(items) + 1):
        item, wt, val = items[j-1]
        for w in xrange(1, limit + 1):
            if wt > w:
                table[j][w] = table[j-1][w]
            else:
                table[j][w] = max(table[j-1][w],
                                  table[j-1][w-wt] + val)

    result = []
    w = limit
    for j in range(len(items), 0, -1):
        was_added = table[j][w] != table[j-1][w]

        if was_added:
            item, wt, val = items[j-1]
            result.append(items[j-1])
            w -= wt

    return result


bagged = knapsack01_dp(items, 400)
print("Bagged the following items\n  " +
      '\n  '.join(sorted(item for item,_,_ in bagged)))
val, wt = totalvalue(bagged)
print("for a total value of %i and a total weight of %i" % (val, -wt))
```

Longest increasing subsequence

```
def longest_sub(arr): #length only
  res = []
  for i in arr:
    p = bisect_left(res, i)
    if p < len(res):
      res[p] = i
    else:
      res += [i]
  return len(res)
#other
from collections import namedtuple
from functools import total_ordering
from bisect import bisect_left

@total_ordering
class Node(namedtuple('Node_', 'val back')):
    def __iter__(self):
```

```
        while self is not None:
            yield self.val
            self = self.back
    def __lt__(self, other):
        return self.val < other.val
    def __eq__(self, other):
        return self.val == other.val

def lis(d):
    """Return one of the L.I.S. of list d using patience sorting."""
    if not d:
        return []
    pileTops = []
    for di in d:
        j = bisect_left(pileTops, Node(di, None))
        #bisect right for non-strictly
        new_node = Node(di, pileTops[j-1] if j > 0 else None)
        if j == len(pileTops):
            pileTops.append(new_node)
        else:
            pileTops[j] = new_node

    return list(pileTops[-1])[::-1]
```

Max Flow (Ford-Fulkerson)

Note: Using a matrix g with source s and sink t

```
from queue import Queue
import math
def max_flow(s, t, g):
    m = 0
    path = bfs(s, t, g)
    while path[t] != -1:
        flow = math.inf
        current = t
        while current != s:
            flow = min(flow, g[path[current]][current])
            current = path[current]
        current = t
        while current != s:
            g[path[current]][current] -= flow
            g[current][path[current]] += flow
            current = path[current]
        m += flow
        path = bfs(s, t, g)
    return m
```

```
def bfs(s, t, g):
    visited = [False for i in range(len(g))]
    q = Queue()
    q.put(s)
    visited[s] = True
    path = [-1 for i in range(len(g))]
    while not q.empty():
        n = q.get()
        if n == t: break
        for i in range(len(g)):
            if not visited[i] and g[n][i] > 0:
                q.put(i)
                path[i] = n
                visited[i] = True
    return path
```

Binomial

Note: Avoid calculating too much factorials

```
def fact(n):
    x = 1
    for i in range(1, n+1): x *= 1
    return x
def nChoosek(n, k):
    if n-k < k:
        k = n-k
    f = n
    res = 1
    for i in range(k):
        res *= f
        f -= 1
    return res//fact(k)
```

Prime Sieve

```
def sieve_for_primes_to(n):
    size = n//2
    sieve = [1]*size
    limit = int(n**0.5)
    for i in range(1,limit):
        if sieve[i]:
            val = 2*i+1
            tmp = ((size-1) - i)//val
            sieve[i+val::val] = [0]*tmp
    return [2] + [i*2+1 for i, v in enumerate(sieve) if v and i>0]
```

Topological sort

Note: graph is an adjacency matrix with directed edges

```
def topological_sort(graph):
    from collections import deque
    indeg = [0] * len(graph)
    result = []
    q = deque()
    for ns in graph:
        for n in ns:
            indeg[n] += 1

    for i in range(len(graph)):
        if indeg[i] == 0:
            q.appendleft(i)

    while q:
        n = q.pop()
        result.append(n)
        for i in graph[n]:
            indeg[i] -= 1
            if indeg[i] == 0:
                q.appendleft(i)
    return result
```

Euler's totient function

```
def totient(n):
    ans = n
    i = 2
    while i**2 <= n:
        if n % i == 0: ans -= ans / i
        while n % i == 0: n /= i
        i += 1
    if n > 1: ans -= ans / n
    return ans
```

Union Find

Note: Union find as used in Kruskal's minimum spanning tree.

```
class UnionFind:

    def __init__(self, nodes):
        self.nodes = list(range(nodes))
        self.rank = [0]*nodes
    def union(self, x, y):
        rootx = self.nodes[x]
        rooty = self.nodes[y]
        if self.rank[rootx] > self.rank[rooty]:
            self.nodes[rooty] = rootx
        elif self.rank[rootx] < self.rank[rooty]:
            self.nodes[rootx] = rooty
        elif rootx != rooty:
            self.nodes[rooty] = rootx
            self.rank[rootx] += 1

    def find(self, x):
        if self.nodes[x] == x:
            return x
        self.nodes[x] = self.find(self.nodes[x])
        return self.nodes[x]
```