

<b>1</b>	<b>Setup</b>	<b>2</b>							
1.0.1	Tips . . . . .	2	3.1.7	Suc. shortest path . . . . .	7	3.7.2	Matrix Solve . . . . .	16	
1.0.2	header.h . . . . .	2	3.1.8	Bipartite check . . . . .	7	3.7.3	Matrix Exp. . . . .	16	
1.0.3	Aux. helper C++ . . . . .	2	3.1.9	Find cycle directed . . . . .	7	3.7.4	Finite field . . . . .	16	
1.0.4	Aux. helper python . . . . .	2	3.1.10	Find cycle undirected . . . . .	8	3.7.5	Complex field . . . . .	16	
<b>2</b>	<b>Python</b>	<b>2</b>	3.1.11	Tarjan's SCC . . . . .	8	3.7.6	FFT . . . . .	17	
2.1	Graphs . . . . .	2	3.1.12	SCC edges . . . . .	9	3.7.7	Polyn. inv. div. . . . .	17	
2.1.1	BFS . . . . .	2	3.1.13	Topological sort . . . . .	9	3.7.8	Linear recurs. . . . .	17	
2.1.2	Dijkstra . . . . .	2	3.1.14	Bellmann-Ford . . . . .	9	3.7.9	Convolution . . . . .	18	
2.1.3	Topological Sort . . . . .	3	3.1.15	Ford-Fulkerson . . . . .	9	3.7.10	Partitions of $n$ . . . . .	18	
2.1.4	Kruskal (UnionFind) . . . . .	3	3.1.16	Dinic max flow . . . . .	10	3.7.11	Ternary search . . . . .	18	
2.1.5	Prim . . . . .	3	3.1.17	Edmonds-Karp . . . . .	10	3.8	Other Data Structures . . . . .	18	
2.2	Num. Th. / Comb. . . . .	3	3.2	Dynamic Programming . . . . .	10	3.8.1	Disjoint set . . . . .	18	
2.2.1	nCk % prime . . . . .	3	3.2.1	Longest Incr. Subseq. . . . .	10	3.8.2	Fenwick tree . . . . .	19	
2.2.2	Sieve of E. . . . .	3	3.2.2	0-1 Knapsack . . . . .	11	3.8.3	Trie . . . . .	19	
2.2.3	Modular Inverse . . . . .	4	3.2.3	Coin change . . . . .	11	3.8.4	Treap . . . . .	19	
2.2.4	Chinese rem. . . . .	4	3.3	Numerical . . . . .	11	3.8.5	Segment tree . . . . .	19	
2.2.5	Bezout . . . . .	4	3.3.1	Template (for this section) . . . . .	11	3.8.6	Lazy segment tree . . . . .	20	
2.2.6	Gen. chinese rem. . . . .	4	3.3.2	Polynomial . . . . .	11	3.8.7	Dynamic segment tree . . . . .	20	
2.3	Strings . . . . .	4	3.3.3	Poly Roots . . . . .	11	3.8.8	Suffix tree . . . . .	20	
2.3.1	Longest common substr. . . . .	4	3.3.4	Golden Section Search . . . . .	11	3.8.9	UnionFind . . . . .	21	
2.3.2	Longest common subseq. . . . .	4	3.3.5	Hill Climbing . . . . .	11	3.8.10	Indexed set . . . . .	21	
2.3.3	KMP . . . . .	4	3.3.6	Integration . . . . .	12	<b>4</b>	<b>Other Mathematics</b>	<b>21</b>	
2.3.4	Suffix Array . . . . .	4	3.3.7	Integration Adaptive . . . . .	12	4.1	Helpful functions . . . . .	21	
2.3.5	Longest common pref. . . . .	5	3.4	Num. Th. / Comb. . . . .	12	4.1.1	Euler's Totient Fuction . . . . .	21	
2.3.6	Edit distance . . . . .	5	3.4.1	Basic stuff . . . . .	12	4.1.2	Totient (again but .py) . . . . .	21	
2.3.7	Bitstring . . . . .	5	3.4.2	Mod. exponentiation . . . . .	12	4.1.3	Pascal's trinagle . . . . .	21	
2.4	Geometry . . . . .	5	3.4.3	GCD . . . . .	13	4.2	Theorems and definitions . . . . .	21	
2.4.1	Convex Hull . . . . .	5	3.4.4	Sieve of Eratosthenes . . . . .	13	4.3	Geometry Formulas . . . . .	22	
2.4.2	Geometry . . . . .	5	3.4.5	Fibonacci % prime . . . . .	13	4.4	Recurrences . . . . .	22	
2.5	Other Algorithms . . . . .	5	3.4.6	nCk % prime . . . . .	13	4.5	Sums . . . . .	22	
2.5.1	Rotate matrix . . . . .	5	3.5	Strings . . . . .	13	4.6	Series . . . . .	22	
2.6	Other Data Structures . . . . .	5	3.5.1	Z alg. . . . .	13	4.7	Quadrilaterals . . . . .	23	
2.6.1	Trie . . . . .	5	3.5.2	KMP . . . . .	13	4.8	Triangles . . . . .	23	
<b>3</b>	<b>C++</b>	<b>6</b>	3.5.3	Aho-Corasick . . . . .	13	4.9	Trigonometry . . . . .	23	
3.1	Graphs . . . . .	6	3.5.4	Long. palin. subs . . . . .	14	4.10	Combinatorics . . . . .	23	
3.1.1	BFS . . . . .	6	3.5.5	Bitstring . . . . .	14	4.11	Cycles . . . . .	23	
3.1.2	DFS . . . . .	6	3.6	Geometry . . . . .	14	4.12	Labeled unrooted trees . . . . .	23	
3.1.3	Dijkstra . . . . .	6	3.6.1	essentials.cpp . . . . .	14	4.13	Partition function . . . . .	23	
3.1.4	Floyd-Warshall . . . . .	6	3.6.2	Two segs. itersec. . . . .	15	4.14	Numbers . . . . .	23	
3.1.5	Kruskal . . . . .	6	3.6.3	Convex Hull . . . . .	15	4.15	Probability . . . . .	24	
3.1.6	Hungarian algorithm . . . . .	7	3.7	Other Algorithms . . . . .	15	4.16	Number Theory . . . . .	24	
			3.7.1	2-sat . . . . .	15	4.17	Discrete distributions . . . . .	25	
						4.18	Continuous distributions . . . . .	25	

## 1 Setup

**1.0.1 Tips Test session:** Check `__int128` and GNU builtins.

**C++ var. limits:** `int`  $-2^{31}$ ,  $2^{31} - 1$

`ll`  $-2^{63}$ ,  $2^{63} - 1$

`ull`  $0$ ,  $2^{64} - 1$

`__int128`  $-2^{127}$ ,  $2^{127} - 1$

`ld`  $-1.7e308$ ,  $1.7e308$ , 18 digits precision

### 1.0.2 header.h

```
1 #pragma once
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 #define ll long long
6 #define ull unsigned ll
7 #define ld long double
8 #define pl pair<ll, ll>
9 #define pi pair<int, int>
10 #define vl vector<ll>
11 #define vi vector<int>
12 #define vb vector<bool>
13 #define vvi vector<vi>
14 #define vvl vector<vl>
15 #define vpl vector<pl>
16 #define vpi vector<pi>
17 #define vld vector<ld>
18 #define vvp vector<vp>
19 #define in(el, cont) (cont.find(el) != cont.end())
20 // sets/maps
21 #define all(x) x.begin(), x.end()
22
23 constexpr int INF = 2000000000;
24 constexpr ll LLINF = 9000000000000000000LL;
25
26 // int main() {
27 //   ios::sync_with_stdio(false); // do not use
28 //   cout + printf
29 //   cin.tie(NULL);
30 //   cout << fixed << setprecision(12);
31 //   return 0;
32 // }
```

### 1.0.3 Aux. helper C++

```
1 #include "header.h"
2 int main() {
3     // Read in a line including white space
4     string line;
```

```
5     getline(cin, line);
6     // When doing the above read numbers as
7     // follows:
8     int n;
9     getline(cin, line);
10    stringstream ss(line);
11    ss >> n;
12
13    // Count the number of 1s in binary
14    // representation of a number
15    ull number;
16    __builtin_popcountll(number);
17 }
18
19 // __int128
20 using lll = __int128;
21 ostream& operator<<(ostream& o, __int128 n) {
22     auto t = n<0 ? -n : n; char b[128], *d = end(b)
23     ;
24     do *--d = '0'+t%10, t /= 10; while (t);
25     if(n<0) *--d = '-';
26     o.rdbuf()->sputn(d, end(b)-d);
27     return o;
28 }
```

### 1.0.4 Aux. helper python

```
1 from functools import lru_cache
2
3 # Read until EOF
4 while True:
5     try:
6         pattern = input()
7     except EOFError:
8         break
9
10 @lru_cache(maxsize=None)
11 def smth_memoi(i, j, s):
12     # Example in-built cache
13     return "sol"
14
15 # Fast I
16 import io, os
17 def fast_io():
18     finput = io.BytesIO(os.read(0,
19                             os.fstat(0).st_size)).readline
20     s = finput().decode()
21     return s
22
23 # Fast O
24 import sys
25 def fast_out():
26     n = 5
27     sys.stdout.write(str(n)+"\n")
```

## 2 Python

### 2.1 Graphs

#### 2.1.1 BFS

```
1 from collections import deque
2 def bfs(g, roots, n):
3     q = deque(roots)
4     explored = set()
5     distances = [0 if v in roots else float('inf')
6                  for v in range(n)]
7     while len(q) != 0:
8         node = q.popleft()
9         if node in explored: continue
10        explored.add(node)
11        for neigh in g[node]:
12            if neigh not in explored:
13                q.append(neigh)
14                distances[neigh] = distances[node] + 1
15    return distances
```

#### 2.1.2 Dijkstra

```
1 from heapq import *
2 def dijkstra(n, root, g): # g = {node: (cost,
3     # neigh)}
4     dist = [float("inf")]*n
5     dist[root] = 0
6     prev = [-1]*n
7
8     pq = [(0, root)]
9     heapify(pq)
10    visited = set([])
11
12    while len(pq) != 0:
13        _, node = heappop(pq)
14
15        if node in visited: continue
16        visited.add(node)
17
18        # In case of disconnected graphs
19        if node not in g:
20            continue
21
22        for cost, neigh in g[node]:
23            alt = dist[node] + cost
24            if alt < dist[neigh]:
25                dist[neigh] = alt
26                prev[neigh] = node
27                heappush(pq, (alt, neigh))
28    return dist
```

**2.1.3 Topological Sort** topological sorting of a DAG

---

```

1 from collections import defaultdict
2 class Graph:
3     def __init__(self, vertices):
4         self.graph = defaultdict(list) #adjacency
5         List
6         self.V = vertices #No. V
7
8     def addEdge(self, u, v):
9         self.graph[u].append(v)
10
11     def topologicalSortUtil(self, v, visited, stack):
12         :
13         visited[v] = True
14         # Recur for all the vertices adjacent to
15         this vertex
16         for i in self.graph[v]:
17             if visited[i] == False:
18                 self.topologicalSortUtil(i,
19                     visited, stack)
20         stack.insert(0, v)
21
22     def topologicalSort(self):
23         visited = [False]*self.V
24         stack = []
25         for i in range(self.V):
26             if visited[i] == False:
27                 self.topologicalSortUtil(i,
28                     visited, stack)
29         return stack
30
31     def isCyclicUtil(self, v, visited, recStack):
32         visited[v] = True
33         recStack[v] = True
34         for neighbour in self.graph[v]:
35             if visited[neighbour] == False:
36                 if self.isCyclicUtil(neighbour,
37                     visited, recStack) == True:
38                     return True
39             elif recStack[neighbour] == True:
40                 return True
41         recStack[v] = False
42         return False
43
44     def isCyclic(self):
45         visited = [False] * (self.V + 1)
46         recStack = [False] * (self.V + 1)
47         for node in range(self.V):
48             if visited[node] == False:
49                 if self.isCyclicUtil(node,
50                     visited, recStack) == True:
51                     return True
52         return False

```

---

**2.1.4 Kruskal (UnionFind)** Min. span. tree

---

```

1 class UnionFind:
2     def __init__(self, n):
3         self.parent = [-1]*n
4
5     def find(self, x):
6         if self.parent[x] < 0:
7             return x
8         self.parent[x] = self.find(self.parent[x])
9         return self.parent[x]
10
11     def connect(self, a, b):
12         ra = self.find(a)
13         rb = self.find(b)
14         if ra == rb:
15             return False
16         if self.parent[ra] > self.parent[rb]:
17             self.parent[rb] += self.parent[ra]
18             self.parent[ra] = rb
19         else:
20             self.parent[ra] += self.parent[rb]
21             self.parent[rb] = ra
22         return True
23
24 # Full MST is len(spanning==n-1)
25 def kruskal(n, edges):
26     uf = UnionFind(n)
27     spanning = []
28     edges.sort(key = lambda d: -d[2])
29     while edges and len(spanning) < n-1:
30         u, v, w = edges.pop()
31         if not uf.connect(u, v):
32             continue
33         spanning.append((u, v, w))
34     return spanning

```

---

**2.1.5 Prim** Min. span. tree - good for dense graphs

---

```

1 from heapq import heappush, heappop, heapify
2 def prim(G, n):
3     s = next(iter(G.keys()))
4     V = set([s])
5     M = []
6     c = 0
7
8     E = [(w,s,v) for v,w in G[s].items()]
9     heapify(E)
10
11     while E and len(M) < n-1:
12         w,u,v = heappop(E)
13         if v in V: continue
14         V.add(v)
15         M.append((u,v))

```

---



---

```

16     c += w
17     u = v
18     [heappush(E,(w,u,v)) for v,w in G[u].items()
19         if v not in V]
20
21     if len(M) == n-1:
22         return M, c
23     else:
24         return None, None

```

---

**2.2 Num. Th. / Comb.****2.2.1 nCk % prime** p must be prime and k < p

---

```

1 def fermtat_binom(n, k, p):
2     if k > n:
3         return 0
4     num = 1
5     for i in range(n-k+1, n+1):
6         num *= i % p
7     num %= p
8     denom = 1
9     for i in range(1, k+1):
10         denom *= i % p
11     denom %= p
12     # numerator * denominator^(p-2) (mod p)
13     return (num * pow(denom, p-2, p)) % p

```

---

**2.2.2 Sieve of E.**  $O(n)$  so actually faster than C++ version, but more memory

---

```

1 MAX_SIZE = 10**8+1
2 isprime = [True] * MAX_SIZE
3 prime = []
4 SPF = [None] * (MAX_SIZE)
5 def manipulated_seive(N): # Up to N (not
6     included)
7     isprime[0] = isprime[1] = False
8     for i in range(2, N):
9         if isprime[i] == True:
10             prime.append(i)
11             SPF[i] = i
12             j = 0
13             while (j < len(prime) and
14                 i * prime[j] < N and
15                 prime[j] <= SPF[i]):
16                 isprime[i * prime[j]] = False
17                 SPF[i * prime[j]] = prime[j]
18                 j += 1

```

---

### 2.2.3 Modular Inverse of a mod b

---

```

1 def modinv(a, b):
2     if b == 1: return 1
3     b0, x0, x1 = b, 0, 1
4     while a > 1:
5         q, a, b = a//b, b, a%b
6         x0, x1 = x1 - q * x0, x0
7     if x1 < 0: x1 += b0
8     return x1

```

---

**2.2.4 Chinese rem.** an  $x$  such that  $\forall y, m: yx = 1 \bmod m$  requires all  $m, m'$  to be  $\perp=1$  and coprime

---

```

1 def chinese_remainder(ys, ms):
2     N, x = 1, 0
3     for m in ms: N*=m
4     for y,m in zip(ys,ms):
5         n = N // m
6         x += n * y * modinv(n, m)
7     return x % N

```

---

### 2.2.5 Bezout

---

```

1 def bezout_id(a, b):
2     r,x,s,y,t,z = b,a,0,1,1,0
3     while r:
4         q = x // r
5         x, r = r, x % r
6         y, s = s, y - q * s
7         z, t = t, z - q * t
8     return y % (b // x), z % (-a // x)

```

---

### 2.2.6 Gen. chinese rem.

---

```

1 def general_chinese_remainder(a,b,m,n):
2     g = gcd(m,n)
3
4     if a == b and m == n:
5         return a, m
6     if (a % g) != (b % g):
7         return None, None
8
9     u,v = bezout_id(m,n)
10    x = (a*v*n + b*u*m) // g
11    return int(x) % lcm(m,n), int(lcm(m,n))

```

---

## 2.3 Strings

### 2.3.1 Longest common substr. (Consecutive)

$O(mn)$  time,  $O(m)$  space

---

```

1 from functools import lru_cache
2 @lru_cache
3 def lcs(s1, s2):
4     if len(s1) == 0 or len(s2) == 0:
5         return 0
6     return max(
7         lcs(s1[:-1], s2), lcs(s1, s2[:-1]),
8         (s1[-1] == s2[-1]) + lcs(s1[:-1], s2[:-1])
9     )

```

---

### 2.3.2 Longest common subseq. (Non-consecutive)

---

```

1 def longestCommonSubsequence(text1, text2):
2     n = len(text1)
3     m = len(text2)
4     prev = [0] * (m + 1)
5     cur = [0] * (m + 1)
6     for idx1 in range(1, n + 1):
7         for idx2 in range(1, m + 1):
8             # matching
9             if text1[idx1 - 1] == text2[idx2 - 1]:
10                cur[idx2] = 1 + prev[idx2 - 1]
11            else:
12                # not matching
13                cur[idx2] = max(cur[idx2 - 1],
14                               prev[idx2])
15            prev = cur.copy()
16    return cur[m]

```

---

### 2.3.3 KMP Return all matching pos. of P in T

---

```

1 class KMP:
2     def partial(self, pattern):
3         """ Calc. partial match table: String -> [Int]"""
4         ret = [0]
5         for i in range(1, len(pattern)):
6             j = ret[i - 1]
7             while j > 0 and pattern[j] != pattern[i]:
8                 j = ret[j - 1]
9             ret.append(j + 1 if pattern[j] == pattern[i] else j)
10        return ret
11
12    def search(self, T, P):
13        """KMPString -> String -> [Int]"""
14        partial, ret, j = self.partial(P), [], 0

```

---

```

14        for i in range(len(T)):
15            while j > 0 and T[i] != P[j]: j =
16                partial[j - 1]
17            if T[i] == P[j]: j += 1
18            if j == len(P):
19                ret.append(i - (j - 1))
20                j = partial[j - 1]
21        return ret

```

---

### 2.3.4 Suffix Array

---

```

1 class Entry:
2     def __init__(self, pos, nr):
3         self.p = pos
4         self.nr = nr
5
6     def __lt__(self, other):
7         return self.nr < other.nr
8 class SA:
9     def __init__(self, s):
10        self.P = []
11        self.n = len(s)
12        self.build(s)
13
14    def build(self, s): # n log log n
15        n = self.n
16        L = [Entry(0, 0) for _ in range(n)]
17        self.P = []
18        self.P.append([ord(c) for c in s])
19        step = 1
20        count = 1
21
22        # self.P[step][i] stores the position
23        # of the i-th longest suffix
24        # if suffixes are sorted according to
25        # their first 2^step characters.
26        while count < 2 * n:
27            self.P.append([0] * n)
28            for i in range(n):
29                nr = (self.P[step - 1][i],
30                     self.P[step - 1][i +
31                          count])
32                if i + count < n else -1)
33            L[i].p = i
34            L[i].nr = nr
35            L.sort()
36            for i in range(n):
37                if i > 0 and L[i].nr == L[i -
38                     1].nr:
39                    self.P[step][L[i].p] = \
40                        self.P[step][L[i - 1].p]
41                else:
42                    self.P[step][L[i].p] = i
43            step += 1

```

---

```

42     count *= 2
43
44     self.sa = [0] * n
45     for i in range(n):
46         self.sa[self.P[-1][i]] = i

```

---

**2.3.5 Longest common pref.** with the suffix array built we can do, e.g., longest common prefix of  $x$ ,  $y$  with suffixarray where  $x, y$  are suffixes of the string used  $O(\log n)$

---

```

1 def lcp(x, y, P):
2     res = 0
3     if x == y:
4         return n - x
5     for k in range(len(P) - 1, -1, -1):
6         if x >= n or y >= n:
7             break
8         if P[k][x] == P[k][y]:
9             x += 1 << k
10            y += 1 << k
11            res += 1 << k
12    return res

```

---

### 2.3.6 Edit distance

---

```

1 def editDistance(str1, str2):
2     m = len(str1)
3     n = len(str2)
4     curr = [0] * (n + 1)
5     for j in range(n + 1):
6         curr[j] = j
7     previous = 0
8     # dp rows
9     for i in range(1, m + 1):
10        previous = curr[0]
11        curr[0] = i
12
13    # dp cols
14    for j in range(1, n + 1):
15        temp = curr[j]
16        if str1[i - 1] == str2[j - 1]:
17            curr[j] = previous
18        else:
19            curr[j] = 1 + min(previous, curr[j - 1],
20                               curr[j])
21        previous = temp
22    return curr[n]

```

---

**2.3.7 Bitstring** Slower than a set for many elements, but hashable

---

```

1 def add_element(bit_string, index):
2     return bit_string | (1 << index)
3 def remove_element(bit_string, index):
4     return bit_string & ~(1 << index)
5 def contains_element(bit_string, index):
6     return (bit_string & (1 << index)) != 0

```

---

## 2.4 Geometry

### 2.4.1 Convex Hull

---

```

1 def vec(a,b):
2     return (b[0]-a[0], b[1]-a[1])
3 def det(a,b):
4     return a[0]*b[1] - b[0]*a[1]
5 def convexhull(P):
6     if (len(P) == 1):
7         return [(p[0][0], p[0][1])]
8
9     h = sorted(P)
10    lower = []
11    i = 0
12    while i < len(h):
13        if len(lower) > 1:
14            a = vec(lower[-2], lower[-1])
15            b = vec(lower[-1], h[i])
16            if det(a,b) <= 0 and len(lower) > 1:
17                lower.pop()
18                continue
19            lower.append(h[i])
20            i += 1
21
22    upper = []
23    i = 0
24    while i < len(h):
25        if len(upper) > 1:
26            a = vec(upper[-2], upper[-1])
27            b = vec(upper[-1], h[i])
28            if det(a,b) >= 0:
29                upper.pop()
30                continue
31            upper.append(h[i])
32            i += 1
33
34    reversedupper = list(reversed(upper[1:-1]))
35    reversedupper.extend(lower)
36    return reversedupper

```

---

### 2.4.2 Geometry

---

```

1
2 def vec(a,b):
3     return (b[0]-a[0], b[1]-a[1])
4
5 def det(a,b):
6     return a[0]*b[1] - b[0]*a[1]
7
8     lower = []
9     i = 0
10    while i < len(h):
11        if len(lower) > 1:
12            a = vec(lower[-2], lower[-1])
13            b = vec(lower[-1], h[i])
14            if det(a,b) <= 0 and len(lower) > 1:
15                lower.pop()
16                continue
17            lower.append(h[i])
18            i += 1
19
20    # find upper hull
21    # det <= 0 -> replace
22    upper = []
23    i = 0
24    while i < len(h):
25        if len(upper) > 1:
26            a = vec(upper[-2], upper[-1])
27            b = vec(upper[-1], h[i])
28            if det(a,b) >= 0:
29                upper.pop()
30                continue
31            upper.append(h[i])
32            i += 1

```

---

## 2.5 Other Algorithms

### 2.5.1 Rotate matrix

---

```

1 def rotate_matrix(m):
2     return [[m[j][i] for j in range(len(m))] for
3             i in range(len(m[0])-1,-1,-1)]

```

---

## 2.6 Other Data Structures

### 2.6.1 Trie

---

```

1 class TrieNode:
2     def __init__(self):
3         self.children = [None]*26
4         self.isEndOfWord = False
5
6 class Trie:

```

---

```

7  def __init__(self):
8      self.root = self.getNode()
9  def getNode(self):
10     return TrieNode()
11  def _charToIndex(self, ch):
12     return ord(ch)-ord('a')
13  def insert(self, key):
14     pCrawl = self.root
15     length = len(key)
16     for level in range(length):
17         index = self._charToIndex(key[level])
18         if not pCrawl.children[index]:
19             pCrawl.children[index] = self.
20                 getNode()
21             pCrawl.children[index]
22             pCrawl.isEndOfWord = True
23  def search(self, key):
24     pCrawl = self.root
25     length = len(key)
26     for level in range(length):
27         index = self._charToIndex(key[level])
28         if not pCrawl.children[index]:
29             return False
30         pCrawl = pCrawl.children[index]
31     return pCrawl.isEndOfWord

```

## 3 C++

### 3.1 Graphs

#### 3.1.1 BFS

```

1  #include "header.h"
2  #define graph unordered_map<ll, unordered_set<ll>>
3  vi bfs(int n, graph& g, vi& roots) {
4      vi parents(n+1, -1); // nodes are 1..n
5      unordered_set<int> visited;
6      queue<int> q;
7      for (auto x: roots) {
8          q.emplace(x);
9          visited.insert(x);
10     }
11     while (not q.empty()) {
12         int node = q.front();
13         q.pop();
14
15         for (auto neigh: g[node]) {
16             if (not in(neigh, visited)) {
17                 parents[neigh] = node;
18                 q.emplace(neigh);
19                 visited.insert(neigh);
20             }

```

```

21     }
22 }
23 return parents;
24 }
25 vi reconstruct_path(vi parents, int start, int
26 goal) {
27     vi path;
28     int curr = goal;
29     while (curr != start) {
30         path.push_back(curr);
31         if (parents[curr] == -1) return vi(); //
32             No path, empty vi
33         curr = parents[curr];
34     }
35     path.push_back(start);
36     reverse(path.begin(), path.end());
37     return path;

```

#### 3.1.2 DFS Cycle detection / removal

```

1  #include "header.h"
2  void removeCyc(ll node, unordered_map<ll, vector<
3      pair<ll, ll>>>& neighs, vector<bool>& visited
4      ,
5      vector<bool>& recStack, vector<ll>& ans) {
6      if (!visited[node]) {
7          visited[node] = true;
8          recStack[node] = true;
9          auto it = neighs.find(node);
10         if (it != neighs.end()) {
11             for (auto util: it->second) {
12                 ll nnode = util.first;
13                 if (recStack[nnode]) {
14                     ans.push_back(util.second);
15                 } else if (!visited[nnode]) {
16                     removeCyc(nnode, neighs,
17                         visited, recStack, ans);
18                 }
19             }
20         }
21         recStack[node] = false;

```

#### 3.1.3 Dijkstra

```

1  #include "header.h"
2  vector<int> dijkstra(int n, int root, map<int,
3      vector<pair<int, int>>>& g) {
4      unordered_set<int> visited;
5      vector<int> dist(n, INF);
6      priority_queue<pair<int, int>> pq;

```

```

6      dist[root] = 0;
7      pq.push({0, root});
8      while (!pq.empty()) {
9          int node = pq.top().second;
10         int d = -pq.top().first;
11         pq.pop();
12
13         if (in(node, visited)) continue;
14         visited.insert(node);
15
16         for (auto e : g[node]) {
17             int neigh = e.first;
18             int cost = e.second;
19             if (dist[neigh] > dist[node] + cost) {
20                 dist[neigh] = dist[node] + cost;
21                 pq.push({-dist[neigh], neigh});
22             }
23         }
24     }
25     return dist;
26 }

```

#### 3.1.4 Floyd-Warshall

```

1  #include "header.h"
2  // g[i][j] = infity if not path from i to j
3  // if g[i][i] < 0, i is contained in a negative
4  cycle
5  void warshall(vvl& g) {
6      for (int k=0; k<g.size(); ++k) {
7          for (int i=0; i<g.size(); ++i) {
8              for (int j=0; j<g.size(); ++j) {
9                  if (g[i][k] < LLONG_MAX and g[k][
10                     j] < LLONG_MAX and g[i][j] >
11                     g[i][k] + g[k][j]) {
12                     g[i][j] = g[i][k] + g[k][j];
13                 }
14             }
15         }
16     }

```

#### 3.1.5 Kruskal Minimum spanning tree of undirected weighted graph. $O(E \log E)$

```

1  #include "header.h"
2  #include "disjoint_set.h"
3  pair<set<pair<ll, ll>>, ll> kruskal(vector<tuple
4      <ll, ll, ll>>& edges, ll n) {
5      set<pair<ll, ll>> ans;
6      ll cost = 0;
7
8      sort(edges.begin(), edges.end());
9      DisjointSet<ll> fs(n);
10     ll dist, i, j;

```

```

11 for (auto edge: edges) {
12     dist = get<0>(edge);
13     i = get<1>(edge);
14     j = get<2>(edge);
15
16     if (fs.find_set(i) != fs.find_set(j)) {
17         fs.union_sets(i, j);
18         ans.insert({i, j});
19         cost += dist;
20     }
21 }
22 return pair<set<pair<ll, ll>>, ll> {ans, cost};
23 }

```

**3.1.6 Hungarian algorithm** Given  $J$  jobs and  $W$  workers ( $J \leq W$ ), computes the minimum cost to assign each prefix of jobs to distinct workers.

```

1 #include "header.h"
2 template <class T> bool ckmin(T &a, const T &b) {
3     return b < a ? a = b, 1 : 0; }
4
5 /**
6  * @tparam T: type large enough to represent
7  *           integers of  $0(J * \max(|C|))$ 
8  * @param C:  $J \times W$  matrix such that  $C[j][w] = \text{cost}$ 
9  *           to assign  $j$ -th
10  * job to  $w$ -th worker (possibly negative)
11  * @return a vector (length  $J$ ), with the  $j$ -th
12  *         entry = min. cost
13  * to assign the first  $(j+1)$  jobs to distinct
14  * workers
15 */
16 template <class T> vector<T> hungarian(const
17     vector<vector<T>> &C) {
18     const int J = (int)size(C), W = (int)size(C[0]);
19     assert(J <= W);
20     // a W-th worker added for convenience
21     vector<int> job(W + 1, -1);
22     vector<T> ys(J), yt(W + 1); // potentials
23     vector<T> answers;
24     const T inf = numeric_limits<T>::max();
25     for (int j_cur = 0; j_cur < J; ++j_cur) {
26         int w_cur = W;
27         job[w_cur] = j_cur;
28         vector<T> min_to(W + 1, inf);
29         vector<int> prv(W + 1, -1);
30         vector<bool> in_Z(W + 1);
31         while (job[w_cur] != -1) { // runs at
32             most j_cur + 1 times
33             in_Z[w_cur] = true;
34             const int j = job[w_cur];
35             T delta = inf;

```

```

28 int w_next;
29 for (int w = 0; w < W; ++w) {
30     if (!in_Z[w]) {
31         if (ckmin(min_to[w], C[j][w]
32             - ys[j] - yt[w]))
33             prv[w] = w_cur;
34         if (ckmin(delta, min_to[w]))
35             w_next = w;
36     }
37 }
38 for (int w = 0; w <= W; ++w) {
39     if (in_Z[w]) ys[job[w]] += delta,
40         yt[w] -= delta;
41     else min_to[w] -= delta;
42 }
43 w_cur = w_next;
44 }
45 for (int w; w_cur != W; w_cur = w) job[
46     w_cur] = job[w = prv[w_cur]];
47 answers.push_back(-yt[W]);
48 }
49 return answers;
50 }

```

**3.1.7 Suc. shortest path** Calculates max flow, min cost

```

1 #include "header.h"
2 // map<node, map<node, pair<cost, capacity>>>
3 #define graph unordered_map<int, unordered_map<
4     int, pair<ld, int>>>
5 graph g;
6 const ld inf = 1e60l; // Change if necessary
7 ld fill(int n, vld& potential) { // Finds max
8     flow, min cost
9     priority_queue<pair<ld, int>> pq;
10     vector<bool> visited(n+2, false);
11     vi parent(n+2, 0);
12     vld dist(n+2, inf);
13     dist[0] = 0.1;
14     pq.emplace(make_pair(0.1, 0));
15     while (not pq.empty()) {
16         int node = pq.top().second;
17         pq.pop();
18         if (visited[node]) continue;
19         visited[node] = true;
20         for (auto& x : g[node]) {
21             int neigh = x.first;
22             int capacity = x.second.second;
23             ld cost = x.second.first;
24             if (capacity and not visited[neigh]) {

```

```

25         dist[neigh] = d;
26         pq.emplace(make_pair(-d, neigh));
27         parent[neigh] = node;
28     }
29 }
30 for (int i = 0; i < n+2; i++) {
31     potential[i] = min(infty, potential[i] + dist[i]);
32 }
33 if (not parent[n+1]) return infty;
34 ld ans = 0.1;
35 for (int x = n+1; x; x=parent[x]) {
36     ans += g[parent[x]][x].first;
37     g[parent[x]][x].second--;
38     g[x][parent[x]].second++;
39 }
40 return ans;
41 }

```

**3.1.8 Bipartite check**

```

1 #include "header.h"
2 int main() {
3     int n;
4     vvi adj(n);
5
6     vi side(n, -1); // will have 0's for one
7     // side 1's for other side
8     bool is_bipartite = true; // becomes false
9     // if not bipartite
10     queue<int> q;
11     for (int st = 0; st < n; ++st) {
12         if (side[st] == -1) {
13             q.push(st);
14             side[st] = 0;
15             while (!q.empty()) {
16                 int v = q.front();
17                 q.pop();
18                 for (int u : adj[v]) {
19                     if (side[u] == -1) {
20                         side[u] = side[v] ^ 1;
21                         q.push(u);
22                     } else {
23                         is_bipartite &= side[u]
24                             != side[v];
25                     }
26                 }
27             }
28         }
29     }
30 }

```

**3.1.9 Find cycle directed**

```

1 #include "header.h"
2 int n;
3 const int mxN = 2e5+5;

```



```

4 vvi adj(mxN);
5 vector<char> color;
6 vi parent;
7 int cycle_start, cycle_end;
8 bool dfs(int v) {
9     color[v] = 1;
10    for (int u : adj[v]) {
11        if (color[u] == 0) {
12            parent[u] = v;
13            if (dfs(u)) return true;
14        } else if (color[u] == 1) {
15            cycle_end = v;
16            cycle_start = u;
17            return true;
18        }
19    }
20    color[v] = 2;
21    return false;
22 }
23 void find_cycle() {
24     color.assign(n, 0);
25     parent.assign(n, -1);
26     cycle_start = -1;
27     for (int v = 0; v < n; v++) {
28         if (color[v] == 0 && dfs(v)) break;
29     }
30     if (cycle_start == -1) {
31         cout << "Acyclic" << endl;
32     } else {
33         vector<int> cycle;
34         cycle.push_back(cycle_start);
35         for (int v = cycle_end; v != cycle_start;
36             v = parent[v])
37             cycle.push_back(v);
38         cycle.push_back(cycle_start);
39         reverse(cycle.begin(), cycle.end());
40
41         cout << "Cycle Found: ";
42         for (int v : cycle) cout << v << " ";
43         cout << endl;
44     }
45 }

```

### 3.1.10 Find cycle undirected

```

1 #include "header.h"
2 int n;
3 const int mxN = 2e5 + 5;
4 vvi adj(mxN);
5 vector<bool> visited;
6 vi parent;
7 int cycle_start, cycle_end;
8 bool dfs(int v, int par) { // passing vertex and
9     // its parent vertex

```

```

9     visited[v] = true;
10    for (int u : adj[v]) {
11        if (u == par) continue; // skipping edge
12        // to parent vertex
13        if (visited[u]) {
14            cycle_end = v;
15            cycle_start = u;
16            return true;
17        }
18        parent[u] = v;
19        if (dfs(u, parent[u]))
20            return true;
21    }
22    return false;
23 }
24 void find_cycle() {
25     visited.assign(n, false);
26     parent.assign(n, -1);
27     cycle_start = -1;
28     for (int v = 0; v < n; v++) {
29         if (!visited[v] && dfs(v, parent[v]))
30             break;
31     }
32     if (cycle_start == -1) {
33         cout << "Acyclic" << endl;
34     } else {
35         vector<int> cycle;
36         cycle.push_back(cycle_start);
37         for (int v = cycle_end; v != cycle_start;
38             v = parent[v])
39             cycle.push_back(v);
40         cycle.push_back(cycle_start);
41         cout << "Cycle Found: ";
42         for (int v : cycle) cout << v << " ";
43         cout << endl;
44     }
45 }

```

### 3.1.11 Tarjan's SCC

```

1 #include "header.h"
2 struct Tarjan {
3     vvi &edges;
4     int V, counter = 0, C = 0;
5     vi n, l;
6     vector<bool> vs;
7     stack<int> st;
8     Tarjan(vvi &e) : edges(e), V(e.size()), n(V,
9         -1), l(V, -1), vs(V, false) {}
10    void visit(int u, vi &com) {
11        l[u] = n[u] = counter++;
12        st.push(u);
13        vs[u] = true;
14        for (auto &&v : edges[u]) {

```

```

14         if (n[v] == -1) visit(v, com);
15         if (vs[v]) l[u] = min(l[u], l[v]);
16     }
17     if (l[u] == n[u]) {
18         while (true) {
19             int v = st.top();
20             st.pop();
21             vs[v] = false;
22             com[v] = C; // <== ACT HERE
23             if (u == v) break;
24         }
25         C++;
26     }
27 }
28 int find_sccs(vi &com) { // component indices
29     // will be stored in 'com'
30     com.assign(V, -1);
31     C = 0;
32     for (int u = 0; u < V; ++u)
33         if (n[u] == -1) visit(u, com);
34     return C;
35 }
36 // scc is a map of the original vertices of the
37 // graph to the vertices of the SCC graph,
38 // scc_graph is its adjacency list. SCC
39 // indices and edges are stored in 'scc' and '
40 // scc_graph'.
41 void scc_collapse(vi &scc, vvi &scc_graph) {
42     find_sccs(scc);
43     scc_graph.assign(C, vi());
44     set<pi> rec; // recorded edges
45     for (int u = 0; u < V; ++u) {
46         assert(scc[u] != -1);
47         for (int v : edges[u]) {
48             if (scc[v] == scc[u] ||
49                 rec.find({scc[u], scc[v]}) != rec.end())
50                 continue;
51             scc_graph[scc[u]].push_back(scc[v]);
52             rec.insert({scc[u], scc[v]});
53         }
54     }
55 }
56 // The number of edges needed to be added is
57 // max(sources.size(), sinks.())
58 void findSourcesAndSinks(const vvi &scc_graph,
59     vi &sources, vi &sinks) {
60     vi in_degree(C, 0), out_degree(C, 0);
61     for (int u = 0; u < C; ++u) {
62         for (auto v : scc_graph[u]) {
63             in_degree[v]++;
64             out_degree[u]++;
65         }
66     }
67     for (int i = 0; i < C; ++i) {
68         if (in_degree[i] == 0) sources.push_back(i)

```



```

    ;
    if (out_degree[i] == 0) sinks.push_back(i);
}
};

```

**3.1.12 SCC edges** Prints out the missing edges to make the input digraph strongly connected

```

1 #include "header.h"
2 const int N=1e5+10;
3 int n,a[N],cnt[N],vis[N];
4 vector<int> hd,tl;
5 int dfs(int x){
6     vis[x]=1;
7     if(!vis[a[x]])return vis[x]=dfs(a[x]);
8     return vis[x]=x;
9 }
10 int main(){
11     scanf("%d",&n);
12     for(int i=1;i<=n;i++){
13         scanf("%d",&a[i]);
14         cnt[a[i]]++;
15     }
16     int k=0;
17     for(int i=1;i<=n;i++){
18         if(!cnt[i]){
19             k++;
20             hd.push_back(i);
21             tl.push_back(dfs(i));
22         }
23     }
24     int tk=k;
25     for(int i=1;i<=n;i++){
26         if(!vis[i]){
27             k++;
28             hd.push_back(i);
29             tl.push_back(dfs(i));
30         }
31     }
32     if(k==1&&!tk)k=0;
33     printf("%d\n",k);
34     for(int i=0;i<k;i++)printf("%d_ %d\n",tl[i],hd
35         [(i+1)%k]);
36     return 0;
37 }

```

### 3.1.13 Topological sort

```

1 #include "header.h"
2 int n; // number of vertices
3 vvi adj; // adjacency list of graph
4 vector<bool> visited;

```

```

5 vi ans;
6 void dfs(int v) {
7     visited[v] = true;
8     for (int u : adj[v]) {
9         if (!visited[u]) dfs(u);
10    }
11    ans.push_back(v);
12 }
13 void topological_sort() {
14     visited.assign(n, false);
15     ans.clear();
16     for (int i = 0; i < n; ++i) {
17         if (!visited[i]) dfs(i);
18     }
19     reverse(ans.begin(), ans.end());
20 }

```

**3.1.14 Bellmann-Ford** Same as Dijkstra but allows neg. edges

```

1 #include "header.h"
2 // Switch vi and vvpi to vl and vvpl if necessary
3 void bellmann_ford_extended(vvpi &e, int source,
4     vi &dist, vb &cyc) {
5     dist.assign(e.size(), INF);
6     cyc.assign(e.size(), false); // true when u is
7     in a <0 cycle
8     dist[source] = 0;
9     for (int iter = 0; iter < e.size() - 1; ++iter)
10    {
11        bool relax = false;
12        for (int u = 0; u < e.size(); ++u)
13            if (dist[u] == INF) continue;
14            else for (auto &e : e[u])
15                if(dist[u]+e.second < dist[e.first])
16                    dist[e.first] = dist[u]+e.second, relax
17                    = true;
18            if(!relax) break;
19    }
20    bool ch = true;
21    while (ch) { // keep going untill no more
22        changes
23        ch = false; // set dist to -INF when in cycle
24        for (int u = 0; u < e.size(); ++u)
25            if (dist[u] == INF) continue;
26            else for (auto &e : e[u])
27                if (dist[e.first] > dist[u] + e.second
28                    && !cyc[e.first]) {
29                    dist[e.first] = -INF;
30                    ch = true; //return true for cycles
31                    cyc[e.first] = true;
32                }
33    }
34 }

```

### 3.1.15 Ford-Fulkerson Basic Max. flow

```

1 #include "header.h"
2 #define V 6 // Num. of vertices in given graph
3 /* Returns true if there is a path from source 's'
4    't' to sink
5    't' in residual graph. Also fills parent[] to
6    store the
7    path */
8 bool bfs(int rGraph[V][V], int s, int t, int
9     parent[]) {
10    bool visited[V];
11    memset(visited, 0, sizeof(visited));
12    queue<int> q;
13    q.push(s);
14    visited[s] = true;
15    parent[s] = -1;
16    while (!q.empty()) {
17        int u = q.front();
18        q.pop();
19        for (int v = 0; v < V; v++) {
20            if (visited[v] == false && rGraph[u][v] >
21                0) {
22                if (v == t) {
23                    parent[v] = u;
24                    return true;
25                }
26                q.push(v);
27                parent[v] = u;
28                visited[v] = true;
29            }
30        }
31    }
32    return false;
33 }
34 // Returns the maximum flow from s to t
35 int fordFulkerson(int graph[V][V], int s, int t)
36 {
37     int u, v;
38     int rGraph[V][V];
39     for (u = 0; u < V; u++)
40         for (v = 0; v < V; v++)
41             rGraph[u][v] = graph[u][v];
42     int parent[V]; // BFS-filled (to store path)
43     int max_flow = 0; // no flow initially
44     while (bfs(rGraph, s, t, parent)) {
45         int path_flow = INT_MAX;
46         for (v = t; v != s; v = parent[v]) {
47             u = parent[v];
48             path_flow = min(path_flow, rGraph[u][v]);
49         }
50         for (v = t; v != s; v = parent[v]) {
51             u = parent[v];
52             rGraph[u][v] -= path_flow;
53             rGraph[v][u] += path_flow;
54         }
55         max_flow += path_flow;
56     }
57     return max_flow;
58 }

```

```

50     rGraph[u][v] -= path_flow;
51     rGraph[v][u] += path_flow;
52 }
53     max_flow += path_flow;
54 }
55     return max_flow;
56 }

```

### 3.1.16 Dinic max flow $O(V^2E)$ , $O(Ef)$

```

1 #include "header.h"
2 using F = ll; using W = ll; // types for flow and
   weight/cost
3 struct S{
4     const int v;      // neighbour
5     const int r;      // index of the reverse edge
6     F f;              // current flow
7     const F cap;      // capacity
8     const W cost;     // unit cost
9     S(int v, int ri, F c, W cost = 0) :
10         v(v), r(ri), f(0), cap(c), cost(cost) {}
11     inline F res() const { return cap - f; }
12 };
13 struct FlowGraph : vector<vector<S>> {
14     FlowGraph(size_t n) : vector<vector<S>>(n) {}
15     void add_edge(int u, int v, F c, W cost = 0){
16         auto &t = *this;
17         t[u].emplace_back(v, t[v].size(), c, cost);
18         t[v].emplace_back(u, t[u].size()-1, c, -cost);
19     }
20     void add_arc(int u, int v, F c, W cost = 0){
21         auto &t = *this;
22         t[u].emplace_back(v, t[v].size(), c, cost);
23         t[v].emplace_back(u, t[u].size()-1, 0, -cost);
24     }
25     void clear() { for (auto &E : *this) for (
26         auto &e : E) e.f = 0LL; }
27 };
28 struct Dinic{
29     FlowGraph &edges; int V,s,t;
30     vi l; vector<vector<S>::iterator> its; //
31     levels and iterators
32     Dinic(FlowGraph &edges, int s, int t) :
33         edges(edges), V(edges.size()), s(s), t(t),
34         l(V,-1), its(V) {}
35     ll augment(int u, F c) { // we reuse the same
36         iterators
37         if (u == t) return c; ll r = 0LL;
38         for(auto &i = its[u]; i != edges[u].end()
39             ; i++){

```

```

33     auto &e = *i;
34     if (e.res() && l[u] < l[e.v]) {
35         auto d = augment(e.v, min(c, e.
36             res()));
37         if (d > 0) { e.f += d; edges[e.v
38             ][e.r].f -= d; c -= d;
39             r += d; if (!c) break; }
40     }
41     return r;
42 }
43 ll run() {
44     ll flow = 0, f;
45     while(true) {
46         fill(l.begin(), l.end(), -1); l[s]=0;
47         queue<int> q; q.push(s);
48         while(!q.empty()){
49             auto u = q.front(); q.pop(); its[
50                 u] = edges[u].begin();
51             for(auto &e : edges[u]) if(e.res
52                 () && l[e.v]<0)
53                 l[e.v] = l[u]+1, q.push(e.v);
54         }
55         if (l[t] < 0) return flow;
56         while ((f = augment(s, INF)) > 0)
57             flow += f;
58     }
59 }

```

**3.1.17 Edmonds-Karp** (Max) flow algorithm with time  $O(VE^2)$ . To get edge flow values, compare capacities before and after, and take the positive values only.

```

1 #include "header.h"
2 template<class T> T edmondsKarp(vector<
   unordered_map<int, T>>&
3     graph, int source, int sink) {
4     assert(source != sink);
5     T flow = 0;
6     vi par(sz(graph)), q = par;
7
8     for (;;) {
9         fill(all(par), -1);
10        par[source] = 0;
11        int ptr = 1;
12        q[0] = source;
13
14        rep(i,0,ptr) {
15            int x = q[i];
16            for (auto e : graph[x]) {
17                if (par[e.first] == -1 && e.second > 0) {
18                    par[e.first] = x;
19                    q[ptr++] = e.first;
20                    if (e.first == sink) goto out;
21                }

```

```

22        }
23    }
24    return flow;
25 out:
26    T inc = numeric_limits<T>::max();
27    for (int y = sink; y != source; y = par[y])
28        inc = min(inc, graph[par[y]][y]);
29
30    flow += inc;
31    for (int y = sink; y != source; y = par[y]) {
32        int p = par[y];
33        if ((graph[p][y] -= inc) <= 0) graph[p].
34            erase(y);
35        graph[y][p] += inc;
36    }
37 }

```

## 3.2 Dynamic Programming

### 3.2.1 Longest Incr. Subseq.

```

1 #include "header.h"
2 template<class T>
3 vector<T> index_path_lis(vector<T>& nums) {
4     int n = nums.size();
5     vector<T> sub;
6     vector<int> subIndex;
7     vector<T> path(n, -1);
8     for (int i = 0; i < n; ++i) {
9         if (sub.empty() || sub[sub.size() - 1] <
10             nums[i]) {
11             path[i] = sub.empty() ? -1 : subIndex[sub.
12                 size() - 1];
13             sub.push_back(nums[i]);
14             subIndex.push_back(i);
15             } else {
16                 int idx = lower_bound(sub.begin(), sub.end(),
17                     nums[i]) - sub.begin();
18                 path[i] = idx == 0 ? -1 : subIndex[idx - 1];
19                 sub[idx] = nums[i];
20                 subIndex[idx] = i;
21             }
22     }
23     vector<T> ans;
24     int t = subIndex[subIndex.size() - 1];
25     while (t != -1) {
26         ans.push_back(t);
27         t = path[t];
28     }
29     reverse(ans.begin(), ans.end());
30     return ans;
31 }
32 // Length only

```

---

```

30 template<class T>
31 int length_lis(vector<T> &a) {
32     set<T> st;
33     typename set<T>::iterator it;
34     for (int i = 0; i < a.size(); ++i) {
35         it = st.lower_bound(a[i]);
36         if (it != st.end()) st.erase(it);
37         st.insert(a[i]);
38     }
39     return st.size();
40 }

```

---

**3.2.2 0-1 Knapsack** Given a number of coins, calculate all possible distinct sums

---

```

1 #include "header.h"
2 int main() {
3     int n;
4     vi coins(n); // possible coins to use
5     int sum = 0; // their sum of the coins
6     vi dp(sum + 1, 0); // dp[x] = 1 if sum x can be
7         made
8     dp[0] = 1;
9     for (int c = 0; c < n; ++c)
10         for (int x = sum; x >= 0; --x)
11             if (dp[x]) dp[x + coins[c]] = 1;

```

---

**3.2.3 Coin change** Total distinct ways to make sum using  $n$  coins of different vals

---

```

1 #include "header.h"
2 int count(vi& coins, int n, int sum) {
3     vvi dp(n + 1, vi(sum + 1, 0));
4     dp[0][0] = 1;
5     for (int i = 1; i <= n; i++) {
6         for (int j = 0; j <= sum; j++) {
7             // without using the current coin,
8             dp[i][j] += dp[i - 1][j];
9             // using the current coin
10            if ((j - coins[i - 1]) >= 0)
11                dp[i][j] += dp[i][j - coins[i - 1]];
12        }
13    }
14    return dp[n][sum];
15 }

```

---

## 3.3 Numerical

### 3.3.1 Template (for this section)

---

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 #define rep(i, a, b) for(int i = a; i < (b); ++i)
4 #define all(x) begin(x), end(x)
5 #define sz(x) (int)(x).size()
6 typedef long long ll;
7 typedef pair<int, int> pii;
8 typedef vector<int> vi;

```

---

### 3.3.2 Polynomial

---

```

1 #include "template.cpp"
2 struct Poly {
3     vector<double> a;
4     double operator()(double x) const {
5         double val = 0;
6         for (int i = sz(a); i--;) (val *= x) += a[i];
7         return val;
8     }
9     void diff() {
10         rep(i, 1, sz(a)) a[i-1] = i*a[i];
11         a.pop_back();
12     }
13     void divroot(double x0) {
14         double b = a.back(), c; a.back() = 0;
15         for(int i=sz(a)-1; i--;) c = a[i], a[i] = a[i+1]*x0+b, b=c;
16         a.pop_back();
17     }
18 };

```

---

**3.3.3 Poly Roots** Finds the real roots to a polynomial.  $O(n^2 \log(1/\epsilon))$

---

```

1 // Usage: polyRoots({{2,-3,1}},-1e9,1e9) = solve
2 // x^2-3x+2 = 0
3 #include "Polynomial.h"
4 #include "template.cpp"
5 vector<double> polyRoots(Poly p, double xmin,
6     double xmax) {
7     if (sz(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
8     vector<double> ret;
9     Poly der = p;
10    der.diff();
11    auto dr = polyRoots(der, xmin, xmax);
12    dr.push_back(xmin-1);
13    dr.push_back(xmax+1);
14    sort(all(dr));
15    rep(i, 0, sz(dr)-1) {
16        double l = dr[i], h = dr[i+1];
17        bool sign = p(l) > 0;
18        if (sign ^ (p(h) > 0)) {

```

---

```

17         rep(it, 0, 60) { // while (h - l > 1e-8)
18             double m = (l + h) / 2, f = p(m);
19             if ((f <= 0) ^ sign) l = m;
20             else h = m;
21         }
22         ret.push_back((l + h) / 2);
23     }
24 }
25 return ret;
26 }

```

---

**3.3.4 Golden Section Search** Finds the argument minimizing the function  $f$  in the interval  $[a, b]$  assuming  $f$  is unimodal on the interval, i.e. has only one local minimum and no local maximum. The maximum error in the result is  $\epsilon$ . Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.  $O(\log((b-a)/\epsilon))$

---

```

1 /** Usage:
2     double func(double x) { return 4+x+.3*x*x; }
3     double xmin = gss(-1000,1000,func); */
4 #include "template.cpp"
5 // It is important for r to be precise, otherwise
6 // we don't necessarily maintain the inequality
7 // a < x1 < x2 < b.
8 double gss(double a, double b, double (*f)(double)) {
9     double r = (sqrt(5)-1)/2, eps = 1e-7;
10    double x1 = b - r*(b-a), x2 = a + r*(b-a);
11    double f1 = f(x1), f2 = f(x2);
12    while (b-a > eps)
13        if (f1 < f2) { //change to > to find maximum
14            b = x2; x2 = x1; f2 = f1;
15            x1 = b - r*(b-a); f1 = f(x1);
16        } else {
17            a = x1; x1 = x2; f1 = f2;
18            x2 = a + r*(b-a); f2 = f(x2);
19        }
20    return a;
21 }

```

---

**3.3.5 Hill Climbing** Poor man's optimization for unimodal functions.

---

```

1 #include "template.cpp"
2 typedef array<double, 2> P;
3 template<class F> pair<double, P> hillClimb(P
4     start, F f) {
5     pair<double, P> cur(f(start), start);
6     for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {

```

---

```

6 rep(j,0,100) rep(dx,-1,2) rep(dy,-1,2) {
7     P p = cur.second;
8     p[0] += dx*jmp;
9     p[1] += dy*jmp;
10    cur = min(cur, make_pair(f(p), p));
11 }
12 }
13 return cur;
14 }

```

**3.3.6 Integration** Simple integration of a function over an interval using Simpson's rule. The error should be proportional to  $h^4$ , although in practice you will want to verify that the result is stable to desired precision when epsilon changes.

```

1 #include "template.cpp"
2 template<class F>
3 double quad(double a, double b, F f, const int n
4     = 1000) {
5     double h = (b - a) / 2 / n, v = f(a) + f(b);
6     rep(i,1,n*2)
7         v += f(a + i*h) * (i&1 ? 4 : 2);
8     return v * h / 3;
9 }

```

**3.3.7 Integration Adaptive** Fast integration using an adaptive Simpson's rule.

```

1 /** Usage:
2 double sphereVolume = quad(-1, 1, [](double x) {
3 return quad(-1, 1, [&](double y) {
4 return quad(-1, 1, [&](double z) {
5 return x*x + y*y + z*z < 1; });});}); */
6 #include "template.cpp"
7 typedef double d;
8 #define S(a,b) (f(a) + 4*f((a+b) / 2) + f(b)) * (
9     b-a) / 6
10 template <class F>
11 d rec(F& f, d a, d b, d eps, d S) {
12     d c = (a + b) / 2;
13     d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
14     if (abs(T - S) <= 15 * eps || b - a < 1e-10)
15         return T + (T - S) / 15;
16     return rec(f, a, c, eps / 2, S1) + rec(f, c, b,
17         eps / 2, S2);
18 }
19 template<class F>
20 d quad(d a, d b, F f, d eps = 1e-8) {
21     return rec(f, a, b, eps, S(a, b));
22 }

```

## 3.4 Num. Th. / Comb.

### 3.4.1 Basic stuff

```

1 #include "header.h"
2 ll gcd(ll a, ll b) { while (b) { a %= b; swap(a,
3     b); } return a; }
4 ll lcm(ll a, ll b) { return (a / gcd(a, b)) * b;
5     }
6 ll mod(ll a, ll b) { return ((a % b) + b) % b; }
7 // Finds x, y s.t. ax + by = d = gcd(a, b).
8 void extended_euclid(ll a, ll b, ll &x, ll &y, ll
9     &d) {
10     ll xx = y = 0;
11     ll yy = x = 1;
12     while (b) {
13         ll q = a / b;
14         ll t = b; b = a % b; a = t;
15         t = xx; xx = x - q * xx; x = t;
16         t = yy; yy = y - q * yy; y = t;
17     }
18     d = a;
19 }
20 // solves ab = 1 (mod n), -1 on failure
21 ll mod_inverse(ll a, ll n) {
22     ll x, y, d;
23     extended_euclid(a, n, x, y, d);
24     return (d > 1 ? -1 : mod(x, n));
25 }
26 // All modular inverses of [1..n] mod P in O(n)
27 // time.
28 vi inverses(ll n, ll P) {
29     vi I(n+1, 1LL);
30     for (ll i = 2; i <= n; ++i)
31         I[i] = mod(-(P/i) * I[P%i], P);
32     return I;
33 }
34 // (a*b)%m
35 ll mulmod(ll a, ll b, ll m){
36     ll x = 0, y=a%m;
37     while(b>0){
38         if(b&1) x = (x+y)%m;
39         y = (2*y)%m, b /= 2;
40     }
41     return x % m;
42 }
43 // Finds b^e % m in O(lg n) time, ensure that b <
44 // m to avoid overflow!
45 ll powmod(ll b, ll e, ll m) {
46     ll p = e<2 ? 1 : powmod((b*b)%m,e/2,m);
47     return e&1 ? p*b%m : p;
48 }
49 // Solve ax + by = c, returns false on failure.
50 bool linear_diophantine(ll a, ll b, ll c, ll &x,
51     ll &y) {
52     ll d = gcd(a, b);

```

```

47 if (c % d) {
48     return false;
49 } else {
50     x = c / d * mod_inverse(a / d, b / d);
51     y = (c - a * x) / b;
52     return true;
53 }
54 }
55 // Description: Tonelli-Shanks algorithm for
56 // modular square roots. Finds x s.t. x^2 = a
57 // \pmod p$ (-x$ gives the other solution). 0
58 // (\log^2 p) worst case, 0(\log p) for most $p$
59 ll sqrtmod(ll a, ll p) {
60     a %= p; if (a < 0) a += p;
61     if (a == 0) return 0;
62     assert(powmod(a, (p-1)/2, p) == 1); // else no
63     // solution
64     if (p % 4 == 3) return powmod(a, (p+1)/4, p);
65     // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if
66     // p % 8 == 5
67     ll s = p - 1, n = 2;
68     int r = 0, m;
69     while (s % 2 == 0)
70         ++r, s /= 2;
71     /// find a non-square mod p
72     while (powmod(n, (p - 1) / 2, p) != p - 1) ++n;
73     ll x = powmod(a, (s + 1) / 2, p);
74     ll b = powmod(a, s, p), g = powmod(n, s, p);
75     for (; r = m) {
76         ll t = b;
77         for (m = 0; m < r && t != 1; ++m)
78             t = t * t % p;
79         if (m == 0) return x;
80         ll gs = powmod(g, 1LL << (r - m - 1), p);
81         g = gs * gs % p;
82         x = x * gs % p;
83         b = b * g % p;
84     }
85 }

```

### 3.4.2 Mod. exponentiation Or use pow() in python

```

1 #include "header.h"
2 ll mod_pow(ll base, ll exp, ll mod) {
3     if (mod == 1) return 0;
4     if (exp == 0) return 1;
5     if (exp == 1) return base;
6
7     ll res = 1;
8     base %= mod;
9     while (exp) {
10         if (exp % 2 == 1) res = (res * base) % mod;
11         exp >>= 1;
12         base = (base * base) % mod;

```

```

13 }
14
15 return res % mod;
16 }

```

### 3.4.3 GCD Or math.gcd in python, std::gcd in C++

```

1 #include "header.h"
2 ll gcd(ll a, ll b) {
3     if (a == 0) return b;
4     return gcd(b % a, a);
5 }

```

### 3.4.4 Sieve of Eratosthenes

```

1 #include "header.h"
2 vl primes;
3 void getprimes(ll n) { // Up to n (not included)
4     vector<bool> p(n, true);
5     p[0] = false;
6     p[1] = false;
7     for(ll i = 0; i < n; i++) {
8         if(p[i]) {
9             primes.push_back(i);
10            for(ll j = i*2; j < n; j+=i) p[j] =
11                false;
12        }
13    }
14 }

```

### 3.4.5 Fibonacci % prime

```

1 #include "header.h"
2 const ll MOD = 1000000007;
3 unordered_map<ll, ll> Fib;
4 ll fib(ll n) {
5     if (n < 2) return 1;
6     if (Fib.find(n) != Fib.end()) return Fib[n];
7     Fib[n] = (fib((n + 1) / 2) * fib(n / 2) + fib
8         ((n - 1) / 2) * fib((n - 2) / 2)) % MOD;
9     return Fib[n];
10 }

```

### 3.4.6 nCk % prime

```

1 #include "header.h"
2 ll binom(ll n, ll k) {
3     ll ans = 1;
4     for(ll i = 1; i <= min(k, n-k); ++i) ans = ans
5         *(n+1-i)/i;
6     return ans;
7 }

```

```

7 ll mod_nCk(ll n, ll k, ll p) {
8     ll ans = 1;
9     while(n){
10         ll np = n%p, kp = k%p;
11         if(kp > np) return 0;
12         ans *= binom(np, kp);
13         n /= p; k /= p;
14     }
15     return ans;
16 }

```

## 3.5 Strings

### 3.5.1 Z alg. KMP alternative (same complexities)

```

1 #include "../header.h"
2 void Z_algorithm(const string &s, vi &Z) {
3     Z.assign(s.length(), -1);
4     int L = 0, R = 0, n = s.length();
5     for (int i = 1; i < n; ++i) {
6         if (i > R) {
7             L = R = i;
8             while (R < n && s[R - L] == s[R]) R++;
9             Z[i] = R - L; R--;
10        } else if (Z[i - L] >= R - i + 1) {
11            L = i;
12            while (R < n && s[R - L] == s[R]) R++;
13            Z[i] = R - L; R--;
14        } else Z[i] = Z[i - L];
15    }
16 }

```

### 3.5.2 KMP

```

1 #include "header.h"
2 void compute_prefix_function(string &w, vi &
3     prefix) {
4     prefix.assign(w.length(), 0);
5     int k = prefix[0] = -1;
6
7     for(int i = 1; i < w.length(); ++i) {
8         while(k >= 0 && w[k + 1] != w[i]) k = prefix[
9             k];
10        if(w[k + 1] == w[i]) k++;
11        prefix[i] = k;
12    }
13 }
14 void knuth_morris_pratt(string &s, string &w) {
15     int q = -1;
16     vi prefix;
17     compute_prefix_function(w, prefix);
18     for(int i = 0; i < s.length(); ++i) {

```

```

17     while(q >= 0 && w[q + 1] != s[i]) q = prefix[
18         q];
19     if(w[q + 1] == s[i]) q++;
20     if(q + 1 == w.length()) {
21         // Match at position (i - w.length() + 1)
22         q = prefix[q];
23     }
24 }

```

### 3.5.3 Aho-Corasick Also can be used as Knuth-Morris-Pratt algorithm

```

1 #include "header.h"
2 map<char, int> cti;
3 int cti_size;
4 template<int ALPHABET_SIZE, int (*mp)(char)>
5 struct AC_FSM {
6     struct Node {
7         int child[ALPHABET_SIZE], failure = 0,
8             match_par = -1;
9         vi match;
10        Node() { for (int i = 0; i < ALPHABET_SIZE;
11            ++i) child[i] = -1; }
12    };
13    vector<Node> a;
14    vector<string> &words;
15    AC_FSM(vector<string> &words) : words(words) {
16        a.push_back(Node());
17        construct_automaton();
18    }
19    void construct_automaton() {
20        for (int w = 0, n = 0; w < words.size(); ++w,
21            n = 0) {
22            for (int i = 0; i < words[w].size(); ++i) {
23                if (a[n].child[mp(words[w][i])] == -1) {
24                    a[n].child[mp(words[w][i])] = a.size();
25                    a.push_back(Node());
26                }
27                n = a[n].child[mp(words[w][i])];
28            }
29            a[n].match.push_back(w);
30        }
31        queue<int> q;
32        for (int k = 0; k < ALPHABET_SIZE; ++k) {
33            if (a[0].child[k] == -1) a[0].child[k] = 0;
34            else if (a[0].child[k] > 0) {
35                a[a[0].child[k]].failure = 0;
36                q.push(a[0].child[k]);
37            }
38        }
39        while (!q.empty()) {
40            int r = q.front(); q.pop();

```

```

38     for (int k = 0, arck; k < ALPHABET_SIZE; ++
39           k) {
40         if ((arck = a[r].child[k]) != -1) {
41             q.push(arck);
42             int v = a[r].failure;
43             while (a[v].child[k] == -1) v = a[v].
44                 failure;
45             a[arck].failure = a[v].child[k];
46             a[arck].match_par = a[v].child[k];
47             while (a[arck].match_par != -1
48                   && a[a[arck].match_par].match.empty
49                     ())
50                 a[arck].match_par = a[a[arck].
51                     match_par].match_par;
52         }
53     }
54 }
55 void aho_corasick(string &sentence, vvi &
56     matches){
57     matches.assign(words.size(), vi());
58     int state = 0, ss = 0;
59     for (int i = 0; i < sentence.length(); ++i,
60           ss = state) {
61         while (a[ss].child[mp(sentence[i])] == -1)
62             ss = a[ss].failure;
63         state = a[ss].child[mp(sentence[i])];
64         = a[ss].child[mp(sentence[i])];
65         for (ss = state; ss != -1; ss = a[ss].
66             match_par)
67             for (int w : a[ss].match)
68                 matches[w].push_back(i + 1 - words[w].
69                     length());
70     }
71 }
72 int char_to_int(char c) {
73     return cti[c];
74 }
75 int main() {
76     ll n;
77     string line;
78     while(getline(cin, line)) {
79         stringstream ss(line);
80         ss >> n;
81
82         vector<string> patterns(n);
83         for (auto& p: patterns) getline(cin, p);
84
85         string text;
86         getline(cin, text);
87
88         cti = {}, cti_size = 0;
89         for (auto c: text) {
90             if (not in(c, cti)) {

```

```

85         cti[c] = cti_size++;
86     }
87 }
88 for (auto& p: patterns) {
89     for (auto c: p) {
90         if (not in(c, cti)) {
91             cti[c] = cti_size++;
92         }
93     }
94 }
95
96 vvi matches;
97 AC_FSM <128+1, char_to_int> ac_fms(patterns);
98 ac_fms.aho_corasick(text, matches);
99 for (auto& x: matches) cout << x << endl;
100 }
101
102 }

```

### 3.5.4 Long. palin. subs Manacher - $O(n)$

```

1 #include "header.h"
2 void manacher(string &s, vi &pal) {
3     int n = s.length(), i = 1, l, r;
4     pal.assign(2 * n + 1, 0);
5     while (i < 2 * n + 1) {
6         if ((i&1) && pal[i] == 0) pal[i] = 1;
7         l = i / 2 - pal[i] / 2; r = (i-1) / 2 + pal[i]
8             / 2;
9
10        while (l - 1 >= 0 && r + 1 < n && s[l - 1] ==
11            s[r + 1])
12            --l, ++r, pal[i] += 2;
13
14        for (l = i - 1, r = i + 1; l >= 0 && r < 2 *
15            n + 1; --l, ++r) {
16            if (l <= i - pal[i]) break;
17            if (l / 2 - pal[l] / 2 > i / 2 - pal[i] /
18                2)
19                pal[r] = pal[l];
20            else { if (l >= 0)
21                pal[r] = min(pal[l], i + pal[i] - r);
22                break;
23            }
24        }
25        i = r;
26    }
27 }

```

### 3.5.5 Bitstring Slower than an unordered set (for many elements), but hashable

```

1 #include "../header.h"
2 template<size_t len>

```

```

3 struct pair_hash { // To make it hashable (pair<
4     int, bitset<len>>)
5     std::size_t operator()(const std::pair<int,
6         std::bitset<len>>& p) const {
7         std::size_t h1 = std::hash<int>{}(p.first
8             );
9         std::size_t h2 = std::hash<std::bitset<
10             len>>{}(p.second);
11         return h1 ^ (h2 << 1);
12     }
13 };
14 #define MAXN 1000
15 std::bitset<MAXN> bs;
16 // bs.set(idx) <- set idx-th bit (1)
17 // bs.reset(idx) <- reset idx-th bit (0)
18 // bs.flip(idx) <- flip idx-th bit
19 // bs.test(idx) <- idx-th bit == 1
20 // bs.count() <- number of 1s
21 // bs.any() <- any bit == 1

```

## 3.6 Geometry

### 3.6.1 essentials.cpp

```

1 #include "../header.h"
2 using C = ld; // could be ll or ld
3 constexpr C EPS = 1e-10; // change to 0 for C=ll
4 struct P { // may also be used as a 2D vector
5     C x, y;
6     P(C x = 0, C y = 0) : x(x), y(y) {}
7     P operator+ (const P &p) const { return {x + p.
8         x, y + p.y}; }
9     P operator- (const P &p) const { return {x - p.
10         x, y - p.y}; }
11     P operator* (C c) const { return {x * c, y * c
12         }; }
13     P operator/ (C c) const { return {x / c, y / c
14         }; }
15     C operator* (const P &p) const { return x*p.x +
16         y*p.y; }
17     C operator^ (const P &p) const { return x*p.y -
18         p.x*y; }
19     P perp() const { return P{y, -x}; }
20     C lensq() const { return x*x + y*y; }
21     ld len() const { return sqrt((ld)lensq()); }
22     static ld dist(const P &p1, const P &p2) {
23         return (p1-p2).len(); }
24     bool operator==(const P &r) const {
25         return ((*this)-r).lensq() <= EPS*EPS; }
26 };
27 C det(P p1, P p2) { return p1^p2; }
28 C det(P p1, P p2, P o) { return det(p1-o, p2-o);
29     }
30 C det(const vector<P> &ps) {

```



```

24 C sum = 0; P prev = ps.back();
25 for(auto &p : ps) sum += det(p, prev), prev = p
    ;
26 return sum;
27 }
28 // Careful with division by two and C=11
29 C area(P p1, P p2, P p3) { return abs(det(p1, p2,
    p3))/C(2); }
30 C area(const vector<P> &poly) { return abs(det(
    poly))/C(2); }
31 int sign(C c){ return (c > C(0)) - (c < C(0)); }
32 int ccw(P p1, P p2, P o) { return sign(det(p1, p2
    , o)); }
33
34 // Only well defined for C = 1d.
35 P unit(const P &p) { return p / p.len(); }
36 P rotate(P p, ld a) { return P{p.x*cos(a)-p.y*sin
    (a), p.x*sin(a)+p.y*cos(a)}; }

```

### 3.6.2 Two segs. itersec.

```

1 #include "header.h"
2 #include "essentials.cpp"
3 bool intersect(P a1, P a2, P b1, P b2) {
4     if (max(a1.x, a2.x) < min(b1.x, b2.x)) return
        false;
5     if (max(b1.x, b2.x) < min(a1.x, a2.x)) return
        false;
6     if (max(a1.y, a2.y) < min(b1.y, b2.y)) return
        false;
7     if (max(b1.y, b2.y) < min(a1.y, a2.y)) return
        false;
8     bool l1 = ccw(a2, b1, a1) * ccw(a2, b2, a1) <=
        0;
9     bool l2 = ccw(b2, a1, b1) * ccw(b2, a2, b1) <=
        0;
10    return l1 && l2;
11 }

```

### 3.6.3 Convex Hull

```

1 #include "header.h"
2 #include "essentials.cpp"
3 struct ConvexHull { // O(n lg n) monotone chain.
4     size_t n;
5     vector<size_t> h, c; // Indices of the hull
        are in 'h', ccw.
6     const vector<P> &p;
7     ConvexHull(const vector<P> &p) : n(p.size()),
        c(n), p(p) {
8         std::iota(c.begin(), c.end(), 0);
9         std::sort(c.begin(), c.end(), [this](size_t l
        , size_t r) -> bool { return p[l].x != p[

```

```

    r].x ? p[l].x < p[r].x : p[l].y < p[r].y;
    });
10    c.erase(std::unique(c.begin(), c.end(), [this
    ](size_t l, size_t r) { return p[l] == p[
    r]; }), c.end());
11    for (size_t s = 1, r = 0; r < 2; ++r, s = h.
    size()) {
12        for (size_t i : c) {
13            while (h.size() > s && ccw(p[h.end()
    [-2]], p[h.end()[-1]], p[i]) <= 0)
14                h.pop_back();
15            h.push_back(i);
16        }
17        reverse(c.begin(), c.end());
18    }
19    if (h.size() > 1) h.pop_back();
20 }
21 size_t size() const { return h.size(); }
22 template <class T, void U(const P &, const P &,
    const P &, T &)>
23 void rotating_calipers(T &ans) {
24     if (size() <= 2)
25         U(p[h[0]], p[h.back()], p[h.back()], ans);
26     else
27         for (size_t i = 0, j = 1, s = size(); i < 2
            * s; ++i) {
28             while (det(p[h[(i + 1) % s]] - p[h[i % s
    ]], p[h[(j + 1) % s]] - p[h[j % s]]) >=
                0)
29                 j = (j + 1) % s;
30             U(p[h[i % s]], p[h[(i + 1) % s]], p[h[j
    % s]], ans);
31         }
32 }
33 };
34 // Example: furthest pair of points. Now set ans
    = 0LL and call
35 // ConvexHull(pts).rotating_calipers<ll, update>(
    ans);
36 void update(const P &p1, const P &p2, const P &o,
    ll &ans) {
37     ans = max(ans, (ll)max((p1 - o).lenq(), (p2 -
    o).lenq()));
38 }
39 int main() {
40     ios::sync_with_stdio(false); // do not use
        cout + printf
41     cin.tie(NULL);
42
43     int n;
44     cin >> n;
45     while (n) {
46         vector<P> ps;
47         int x, y;
48         for (int i = 0; i < n; i++) {

```

```

49         cin >> x >> y;
50         ps.push_back({x, y});
51     }
52
53     ConvexHull ch(ps);
54     cout << ch.h.size() << endl;
55     for(auto& p: ch.h) {
56         cout << ps[p].x << " " << ps[p].y <<
            endl;
57     }
58     cin >> n;
59 }
60
61 return 0;
62 }

```

## 3.7 Other Algorithms

### 3.7.1 2-sat

```

1 #include "../header.h"
2 #include "../Graphs/tarjan.cpp"
3 struct TwoSAT {
4     int n;
5     vvi imp; // implication graph
6     Tarjan tj;
7
8     TwoSAT(int _n) : n(_n), imp(2 * _n, vi()), tj(
        imp) { }
9
10    // Only copy the needed functions:
11    void add_implies(int c1, bool v1, int c2, bool
        v2) {
12        int u = 2 * c1 + (v1 ? 1 : 0),
13        v = 2 * c2 + (v2 ? 1 : 0);
14        imp[u].push_back(v); // u => v
15        imp[v^1].push_back(u^1); // -v => -u
16    }
17    void add_equivalence(int c1, bool v1, int c2,
        bool v2) {
18        add_implies(c1, v1, c2, v2);
19        add_implies(c2, v2, c1, v1);
20    }
21    void add_or(int c1, bool v1, int c2, bool v2) {
22        add_implies(c1, !v1, c2, v2);
23    }
24    void add_and(int c1, bool v1, int c2, bool v2)
        {
25        add_true(c1, v1); add_true(c2, v2);
26    }
27    void add_xor(int c1, bool v1, int c2, bool v2)
        {
28        add_or(c1, v1, c2, v2);
29        add_or(c1, !v1, c2, !v2);

```



```

30 }
31 void add_true(int c1, bool v1) {
32     add_implies(c1, !v1, c1, v1);
33 }
34
35 // on true: a contains an assignment.
36 // on false: no assignment exists.
37 bool solve(vb &a) {
38     vi com;
39     tj.find_sccs(com);
40     for (int i = 0; i < n; ++i)
41         if (com[2 * i] == com[2 * i + 1])
42             return false;
43
44     vvi bycom(com.size());
45     for (int i = 0; i < 2 * n; ++i)
46         bycom[com[i]].push_back(i);
47
48     a.assign(n, false);
49     vb vis(n, false);
50     for(auto &&component : bycom){
51         for (int u : component) {
52             if (vis[u / 2]) continue;
53             vis[u / 2] = true;
54             a[u / 2] = (u % 2 == 1);
55         }
56     }
57     return true;
58 }
59 };

```

### 3.7.2 Matrix Solve

```

1 #include "header.h"
2 #define REP(i, n) for(auto i = decltype(n)(0); i
   < (n); i++)
3 using T = double;
4 constexpr T EPS = 1e-8;
5 template<int R, int C>
6 using M = array<array<T,C>,R>; // matrix
7 template<int R, int C>
8 T ReducedRowEchelonForm(M<R,C> &m, int rows) {
9     // return the determinant
10    int r = 0; T det = 1; // MODIFIES
11    the input
12    for(int c = 0; c < rows && r < rows; c++) {
13        int p = r;
14        for(int i=r+1; i<rows; i++) if(abs(m[i][c]) >
15            abs(m[p][c])) p=i;
16        if(abs(m[p][c]) < EPS){ det = 0; continue; }
17        swap(m[p], m[r]); det = -det;
18        T s = 1.0 / m[r][c]; t; det *= m[r][c];
19        REP(j,C) m[r][j] *= s; // make leading
20        term in row 1

```

```

17     REP(i,rows) if (i!=r){ t = m[i][c]; REP(j,C)
18         m[i][j] -= t*m[r][j]; }
19     ++r;
20     return det;
21 }
22 bool error, inconst; // error => multiple or
23   inconsistent
24 template<int R,int C> // Mx = a; M:R*R, v:R*C =>
25   x:R*C
26 M<R,C> solve(const M<R,R> &m, const M<R,C> &a,
27     int rows){
28     M<R,R+C> q;
29     REP(r,rows){
30         REP(c,rows) q[r][c] = m[r][c];
31         REP(c,C) q[r][R+c] = a[r][c];
32     }
33     ReducedRowEchelonForm<R,R+C>(q,rows);
34     M<R,C> sol; error = false, inconst = false;
35     REP(c,C) for(auto j = rows-1; j >= 0; --j){
36         T t=0; bool allzero=true;
37         for(auto k = j+1; k < rows; ++k)
38             t += q[j][k]*sol[k][c], allzero &= abs(q[j]
39             ][k]) < EPS;
40         if(abs(q[j][j]) < EPS)
41             error = true, inconst |= allzero && abs(q[j]
42             ][R+c]) > EPS;
43         else sol[j][c] = (q[j][R+c] - t) / q[j][j];
44         // usually q[j][j]=1
45     }
46     return sol;
47 }

```

### 3.7.3 Matrix Exp.

```

1 #include "header.h"
2 #define ITERATE_MATRIX(w) for (int r = 0; r < (w)
3   ; ++r) \
4   for (int c = 0; c < (w); ++c)
5 template <class T, int N>
6 struct M {
7     array<array<T,N>,N> m;
8     M() { ITERATE_MATRIX(N) m[r][c] = 0; }
9     static M id() {
10         M I; for (int i = 0; i < N; ++i) I.m[i][i] =
11         1; return I;
12     }
13     M operator*(const M &rhs) const {
14         M out;
15         ITERATE_MATRIX(N) for (int i = 0; i < N; ++i)
16             out.m[r][c] += m[r][i] * rhs.m[i][c];
17         return out;
18     }
19     M raise(ll n) const {

```

```

18     if(n == 0) return id();
19     if(n == 1) return *this;
20     auto r = (*this**this).raise(n / 2);
21     return (n%2 ? *this*r : r);
22 }
23 };

```

### 3.7.4 Finite field For FFT

```

1 #include "header.h"
2 #include "../NumberTheory/elementary.cpp"
3 template<ll p,ll w> // prime, primitive root
4 struct Field { using T = Field; ll x; Field(ll x
5   =0) : x{x} {}
6   T operator+(T r) const { return {(x+r.x)%p}; }
7   T operator-(T r) const { return {(x-r.x+p)%p}; }
8   T operator*(T r) const { return {(x*r.x)%p}; }
9   T operator/(T r) const { return (*this)*r.inv()
10   ; }
11   T inv() const { return {mod_inverse(x,p)}; }
12   static T root(ll k) { assert( (p-1)%k==0 );
13   // (p-1)%k == 0?
14   auto r = powmod(w, (p-1)/abs(k), p); // k-
15   th root of unity
16   return k>=0 ? T{r} : T{r}.inv();
17 }
18 bool zero() const { return x == 0LL; }
19 };
20 using F1 = Field<1004535809,3 >;
21 using F2 = Field<1107296257,10>; // 1<<30 + 1<<25
22   + 1
23 using F3 = Field<2281701377,3 >; // 1<<31 + 1<<27
24   + 1

```

### 3.7.5 Complex field For FFR

```

1 #include "header.h"
2 const double m_pi = M_PI/64x;
3 struct Complex { using T = Complex; double u,v;
4   Complex(double u=0, double v=0) : u{u}, v{v} {}
5   T operator+(T r) const { return {u+r.u, v+r.v}; }
6   T operator-(T r) const { return {u-r.u, v-r.v}; }
7   T operator*(T r) const { return {u*r.u - v*r.v,
8   u*r.v + v*r.u}; }
9   T operator/(T r) const {
10     auto norm = r.u*r.u+r.v*r.v;
11     return {(u*r.u + v*r.v)/norm, (v*r.u - u*r.v)
12     /norm};
13 }

```

```

12 T operator*(double r) const { return T{u*r, v*r}; }
13 T operator/(double r) const { return T{u/r, v/r}; }
14 T inv() const { return T{1,0}/ *this; }
15 T conj() const { return T{u, -v}; }
16 static T root(11 k){ return {cos(2*m_pi/k), sin(2*m_pi/k)}; }
17 bool zero() const { return max(abs(u), abs(v)) < 1e-6; }
18 };

```

### 3.7.6 FFT

```

1 #include "header.h"
2 #include "complex_field.cpp"
3 #include "fin_field.cpp"
4 void brinc(int &x, int k) {
5     int i = k - 1, s = 1 << i;
6     x ^= s;
7     if ((x & s) != s) {
8         --i; s >>= 1;
9         while (i >= 0 && ((x & s) == s))
10             x = x &~ s, --i, s >>= 1;
11         if (i >= 0) x |= s;
12     }
13 }
14 using T = Complex; // using T=F1,F2,F3
15 vector<T> roots;
16 void root_cache(int N) {
17     if (N == (int)roots.size()) return;
18     roots.assign(N, T{0});
19     for (int i = 0; i < N; ++i)
20         roots[i] = ((i&-i) == i)
21             ? T{cos(2.0*m_pi*i/N), sin(2.0*m_pi*i/N)}
22             : roots[i&-i] * roots[i-(i&-i)];
23 }
24 void fft(vector<T> &A, int p, bool inv = false) {
25     int N = 1<<p;
26     for(int i = 0, r = 0; i < N; ++i, brinc(r, p))
27         if (i < r) swap(A[i], A[r]);
28     // Uncomment to precompute roots (for T=Complex)
29     // . Slower but more precise.
30     // root_cache(N);
31     // , sh=p-1 , --sh
32     for (int m = 2; m <= N; m <= 1) {
33         T w, w_m = T::root(inv ? -m : m);
34         for (int k = 0; k < N; k += m) {
35             w = T{1};
36             for (int j = 0; j < m/2; ++j) {
37                 T w = (!inv ? roots[j<<sh] : roots[j<<sh].conj());
38                 T t = w * A[k + j + m/2];
39                 A[k + j + m/2] = A[k + j] - t;

```

```

39         A[k + j] = A[k + j] + t;
40         w = w * w_m;
41     }
42 }
43 }
44 if(inv){ T inverse = T(N).inv(); for(auto &x : A) x = x*inverse; }
45 }
46 // convolution leaves A and B in frequency domain state
47 // C may be equal to A or B for in-place convolution
48 void convolution(vector<T> &A, vector<T> &B, vector<T> &C){
49     int s = A.size() + B.size() - 1;
50     int q = 32 - __builtin_clz(s-1), N=1<<q; // fails if s=1
51     A.resize(N,{}); B.resize(N,{}); C.resize(N,{});
52     fft(A, q, false); fft(B, q, false);
53     for (int i = 0; i < N; ++i) C[i] = A[i] * B[i];
54     fft(C, q, true); C.resize(s);
55 }
56 void square_inplace(vector<T> &A) {
57     int s = 2*A.size()-1, q = 32 - __builtin_clz(s-1), N=1<<q;
58     A.resize(N,{}); fft(A, q, false);
59     for(auto &x : A) x = x*x;
60     fft(A, q, true); A.resize(s);
61 }

```

### 3.7.7 Polyn. inv. div.

```

1 #include "header.h"
2 #include "fft.cpp"
3 vector<T> &rev(vector<T> &A) { reverse(A.begin(), A.end()); return A; }
4 void copy_into(const vector<T> &A, vector<T> &B, size_t n) {
5     std::copy(A.begin(), A.begin()+min({n, A.size(), B.size()}), B.begin());
6 }
7 // Multiplicative inverse of A modulo x^n. Requires A[0] != 0!!
8 vector<T> inverse(const vector<T> &A, int n) {
9     vector<T> Ai{A[0].inv()};
10    for (int k = 0; (1<<k) < n; ++k) {
11        vector<T> As(4<<k, T(0)), Ais(4<<k, T(0));
12        copy_into(A, As, 2<<k); copy_into(Ai, Ais, Ai.size());
13        fft(As, k+2, false); fft(Ais, k+2, false);
14        for (int i = 0; i < (4<<k); ++i) As[i] = As[i] * Ais[i] * Ais[i];
15        fft(As, k+2, true); Ai.resize(2<<k, {});
16        for (int i = 0; i < (2<<k); ++i) Ai[i] = T(2) * Ai[i] - As[i];

```

```

17 }
18 Ai.resize(n);
19 return Ai;
20 }
21 // Polynomial division. Returns {Q, R} such that A = QB+R, deg R < deg B.
22 // Requires that the leading term of B is nonzero.
23 pair<vector<T>, vector<T>> divmod(const vector<T> &A, const vector<T> &B) {
24     size_t n = A.size()-1, m = B.size()-1;
25     if (n < m) return {vector<T>(1, T(0)), A};
26
27     vector<T> X(A), Y(B), Q, R;
28     convolution(rev(X), Y = inverse(rev(Y), n-m+1), Q);
29     Q.resize(n-m+1); rev(Q);
30
31     X.resize(Q.size()), copy_into(Q, X, Q.size());
32     Y.resize(B.size()), copy_into(B, Y, B.size());
33     convolution(X, Y, X);
34
35     R.resize(m), copy_into(A, R, m);
36     for (size_t i = 0; i < m; ++i) R[i] = R[i] - X[i];
37     while (R.size() > 1 && R.back().zero()) R.pop_back();
38     return {Q, R};
39 }
40 vector<T> mod(const vector<T> &A, const vector<T> &B) {
41     return divmod(A, B).second;
42 }

```

**3.7.8 Linear recurs.** Given a linear recurrence of the form

$$a_n = \sum_{i=0}^{k-1} c_i a_{n-i-1}$$

this code computes  $a_n$  in  $O(k \log k \log n)$  time.

```

1 #include "header.h"
2 #include "poly.cpp"
3 // x^k mod f
4 vector<T> xmod(const vector<T> f, 11 k) {
5     vector<T> r{T(1)};
6     for (int b = 62; b >= 0; --b) {
7         if (r.size() > 1)
8             square_inplace(r), r = mod(r, f);
9         if ((k>>b)&1) {
10             r.insert(r.begin(), T(0));
11             if (r.size() == f.size()) {
12                 T c = r.back() / f.back();

```

```

13     for (size_t i = 0; i < f.size(); ++i)
14         r[i] = r[i] - c * f[i];
15     r.pop_back();
16 }
17 }
18 }
19 return r;
20 }
21 // Given A[0,k) and C[0, k), computes the n-th
    term of:
22 // A[n] = \sum_i C[i] * A[n-i-1]
23 T nth_term(const vector<T> &A, const vector<T> &C
    , ll n) {
24     int k = (int)A.size();
25     if (n < k) return A[n];
26
27     vector<T> f(k+1, T{1});
28     for (int i = 0; i < k; ++i)
29         f[i] = T{-1} * C[k-i-1];
30     f = xmod(f, n);
31
32     T r = T{0};
33     for (int i = 0; i < k; ++i)
34         r = r + f[i] * A[i];
35     return r;
36 }

```

### 3.7.9 Convolution Precise up to 9e15

```

1 #include "header.h"
2 #include "fft.cpp"
3 void convolution_mod(const vi &A, const vi &B, ll
    MOD, vi &C) {
4     int s = A.size() + B.size() - 1; ll m15 = (1LL
        <<15)-1LL;
5     int q = 32 - __builtin_clz(s-1), N=1<<q; //
        fails if s=1
6     vector<T> Ac(N), Bc(N), R1(N), R2(N);
7     for (size_t i = 0; i < A.size(); ++i) Ac[i] = T
        {A[i]&m15, A[i]>>15};
8     for (size_t i = 0; i < B.size(); ++i) Bc[i] = T
        {B[i]&m15, B[i]>>15};
9     fft(Ac, q, false); fft(Bc, q, false);
10    for (int i = 0, j = 0; i < N; ++i, j = (N-1)&(N
        -i)) {
11        T as = (Ac[i] + Ac[j].conj()) / 2;
12        T al = (Ac[i] - Ac[j].conj()) / T{0, 2};
13        T bs = (Bc[i] + Bc[j].conj()) / 2;
14        T bl = (Bc[i] - Bc[j].conj()) / T{0, 2};
15        R1[i] = as*bs + al*bl*T{0,1}, R2[i] = as*bl +
            al*bs;
16    }
17    fft(R1, q, true); fft(R2, q, true);
18    ll p15 = (1LL<<15)%MOD, p30 = (1LL<<30)%MOD; C.
        resize(s);

```

```

19     for (int i = 0; i < s; ++i) {
20         ll l = llround(R1[i].u), m = llround(R2[i].u)
            , h = llround(R1[i].v);
21         C[i] = (l + m*p15 + h*p30) % MOD;
22     }
23 }

```

### 3.7.10 Partitions of $n$ Finds all possible partitions of a number

```

1 #include "header.h"
2 void printArray(int p[], int n) {
3     for (int i = 0; i < n; i++)
4         cout << p[i] << " ";
5     cout << endl;
6 }
7 void printAllUniqueParts(int n) {
8     int p[n]; // array to store a partition
9     int k = 0; // idx of last element in a
        partition
10    p[k] = n;
11
12    // The loop stops when the current partition
        has all 1s
13    while (true) {
14        printArray(p, k + 1);
15        int rem_val = 0;
16        while (k >= 0 && p[k] == 1) {
17            rem_val += p[k];
18            k--;
19        }
20        // no more partitions
21        if (k < 0) return;
22
23        p[k]--;
24        rem_val++;
25
26        // sorted order is violated (fix)
27        while (rem_val > p[k]) {
28            p[k + 1] = p[k];
29            rem_val = rem_val - p[k];
30            k++;
31        }
32
33        p[k + 1] = rem_val;
34        k++;
35    }
36 }

```

**3.7.11 Ternary search** Find the smallest  $i$  in  $[a, b]$  that maximizes  $f(i)$ , assuming that  $f(a) < \dots < f(i) \geq \dots \geq f(b)$ . To reverse which of the sides allows non-strict

inequalities, change the  $<$  marked with (A) to  $\leq$ , and reverse the loop at (B). To minimize  $f$ , change it to  $>$ , also at (B).  $O(\log(b-a))$

```

1 // Usage: int ind = ternSearch(0,n-1,[\&](int i){
    return a[i];});
2 #include "../Numerical/template.cpp"
3 template<class F>
4 int ternSearch(int a, int b, F f) {
5     assert(a <= b);
6     while (b - a >= 5) {
7         int mid = (a + b) / 2;
8         if (f(mid) < f(mid+1)) a = mid; // (A)
9         else b = mid+1;
10    }
11    rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
12    return a;
13 }

```

## 3.8 Other Data Structures

### 3.8.1 Disjoint set (i.e. union-find)

```

1 template <typename T>
2 class DisjointSet {
3     typedef T * iterator;
4     T *parent, n, *rank;
5     public:
6         // O(n), assumes nodes are [0, n)
7         DisjointSet(T n) {
8             this->parent = new T[n];
9             this->n = n;
10            this->rank = new T[n];
11            for (T i = 0; i < n; i++) {
12                parent[i] = i;
13                rank[i] = 0;
14            }
15        }
16
17        // O(log n)
18        T find_set(T x) {
19            if (x == parent[x]) return x;
20            return parent[x] = find_set(parent[x]
                );
21        }
22
23        // O(log n)
24        void union_sets(T x, T y) {
25            x = this->find_set(x);
26            y = this->find_set(y);
27
28            if (x == y) return;
29            if (rank[x] < rank[y]) {
30                T z = x;

```

```

31     x = y;
32     y = z;
33 }
34 parent[y] = x;
35 if (rank[x] == rank[y]) rank[x]++;
36 }
37 };

```

**3.8.2 Fenwick tree** (i.e. BIT) eff. update + prefix sum calc. Can be generalized to arbitrary dimensions by duplicating loops.

```

1 // #include "header.h"
2 template < class T >
3 struct FenwickTree { // use 1 based indices !!!
4     int n ; vector <T > tree ;
5     FenwickTree ( int n ) : n ( n ) { tree .
6         assign ( n + 1 , 0 ) ; }
7     T query ( int l , int r ) { return query ( r
8         ) - query ( l - 1 ) ; }
9     T query ( int r ) {
10         T s = 0;
11         for ( ; r > 0; r -= ( r & ( - r ) ) ) s +=
12             tree [ r ];
13         return s ;
14     }
15 void update ( int i , T v ) {
16     for ( ; i <= n ; i += ( i & ( - i ) ) )
17         tree [ i ] += v ;
18 }
19 };

```

### 3.8.3 Trie

```

1 #include "header.h"
2 const int ALPHABET_SIZE = 26;
3 inline int mp(char c) { return c - 'a'; }
4 struct Node {
5     Node* ch[ALPHABET_SIZE];
6     bool isleaf = false;
7     Node() {
8         for(int i = 0; i < ALPHABET_SIZE; ++i) ch[i]
9             = nullptr;
10     }
11 void insert(string &s, int i = 0) {
12     if (i == s.length()) isleaf = true;
13     else {
14         int v = mp(s[i]);
15         if (ch[v] == nullptr)
16             ch[v] = new Node();
17         ch[v]->insert(s, i + 1);

```

```

18     }
19 }
20
21 bool contains(string &s, int i = 0) {
22     if (i == s.length()) return isleaf;
23     else {
24         int v = mp(s[i]);
25         if (ch[v] == nullptr) return false;
26         else return ch[v]->contains(s, i + 1);
27     }
28 }
29
30 void cleanup() {
31     for (int i = 0; i < ALPHABET_SIZE; ++i)
32         if (ch[i] != nullptr) {
33             ch[i]->cleanup();
34             delete ch[i];
35         }
36 }
37 };

```

**3.8.4 Treap** A binary tree whose nodes contain two values, a key and a priority, such that the key keeps the BST property

```

1 #include "header.h"
2 struct Node {
3     ll v;
4     int sz, pr;
5     Node *l = nullptr, *r = nullptr;
6     Node(ll val) : v(val), sz(1) { pr = rand(); }
7 };
8 int size(Node *p) { return p ? p->sz : 0; }
9 void update(Node* p) {
10     if (!p) return;
11     p->sz = 1 + size(p->l) + size(p->r);
12     // Pull data from children here
13 }
14 void propagate(Node *p) {
15     if (!p) return;
16     // Push data to children here
17 }
18 void merge(Node *&t, Node *l, Node *r) {
19     propagate(l), propagate(r);
20     if (!l) t = r;
21     else if (!r) t = l;
22     else if (l->pr > r->pr)
23         merge(l->r, l->r, r), t = l;
24     else merge(r->l, l, r->l), t = r;
25     update(t);
26 }
27 void split(Node *t, Node *&l, Node *&r, int
28     index) {
29     propagate(t);

```

```

29     if (!t) { l = r = nullptr; return; }
30     int id = size(t->l);
31     if (index <= id) // id \in [index, \infty), so
32         move it right
33         split(t->l, l, t->l, index), r = t;
34     else
35         split(t->r, t->r, r, index - id), l = t;
36     update(t);
37 void splitv(Node *t, Node *&l, Node *&r, ll val)
38     {
39     propagate(t);
40     if (!t) { l = r = nullptr; return; }
41     if (val <= t->v) // t->v \in [val, \infty), so
42         move it right
43         splitv(t->l, l, t->l, val), r = t;
44     else
45         splitv(t->r, t->r, r, val), l = t;
46     update(t);
47 }
48 void clean(Node *p) {
49     if (p) { clean(p->l), clean(p->r); delete p; }
50 }

```

### 3.8.5 Segment tree

```

1 #include "../header.h"
2 // example: SegmentTree<int, min> st(n, INT_MAX);
3 const int& addOp(const int& a, const int& b) {
4     static int result;
5     result = a + b;
6     return result;
7 }
8 template <class T, const T&(*op)(const T&, const
9     T&)>
10 struct SegmentTree {
11     int n; vector<T> tree; T id;
12     SegmentTree(int _n, T _id) : n(_n), tree(2 * n,
13         _id), id(_id) { }
14 void update(int i, T val) {
15     for (tree[i+n] = val, i = (i+n)/2; i > 0; i
16         /= 2)
17         tree[i] = op(tree[2*i], tree[2*i+1]);
18 }
19 T query(int l, int r) {
20     T lhs = T(id), rhs = T(id);
21     for (l += n, r += n; l < r; l >>= 1, r >>= 1)
22     {
23         if (l&1) lhs = op(lhs, tree[l++]);
24         if (!(r&1)) rhs = op(tree[r--], rhs);
25     }
26     return op(l == r ? op(lhs, tree[l]) : lhs,
27         rhs);
28 }

```

---

24 };

---

### 3.8.6 Lazy segment tree Optimizes range updates

---

```

1 #include "../header.h"
2 using T=int; using U=int; using I=int;    //
   exclusive right bounds
3 T t_id; U u_id;
4 T op(T a, T b){ return a+b; }
5 void join(U &a, U b){ a+=b; }
6 void apply(T &t, U u, int x){ t+=x*u; }
7 T convert(const I &i){ return i; }
8 struct LazySegmentTree {
9     struct Node { int l, r, lc, rc; T t; U u;
10         Node(int l, int r, T t=t_id):l(l),r(r),lc(-1),rc(-1),t(t),u(u_id){}
11     };
12     int N; vector<Node> tree; vector<I> &init;
13     LazySegmentTree(vector<I> &init) : N(init.size()), init(init){
14         tree.reserve(2*N-1); tree.push_back({0,N});
15         build(0, 0, N);
16     }
17     void build(int i, int l, int r) { auto &n = tree[i];
18         if (r > l+1) { int m = (l+r)/2;
19             n.lc = tree.size(); n.rc = n.lc+1;
20             tree.push_back({l,m}); tree.push_back({m,r});
21             build(n.lc,l,m); build(n.rc,m,r);
22             n.t = op(tree[n.lc].t, tree[n.rc].t);
23         } else n.t = convert(init[l]);
24     }
25     void push(Node &n, U u){ apply(n.t, u, n.r-n.l);
26         join(n.u,u); }
27     void push(Node &n){push(tree[n.lc],n.u);push(tree[n.rc],n.u);n.u=u_id;}
28     T query(int l, int r, int i = 0) { auto &n = tree[i];
29         if(r <= n.l || n.r <= l) return t_id;
30         if(l <= n.l && n.r <= r) return n.t;
31         return push(n, op(query(l,r,n.lc),query(l,r,n.rc)));
32     }
33     void update(int l, int r, U u, int i = 0) {
34         auto &n = tree[i];
35         if(r <= n.l || n.r <= l) return;
36         if(l <= n.l && n.r <= r) return push(n,u);
37         push(n); update(l,r,u,n.lc); update(l,r,u,n.rc);
38         n.t = op(tree[n.lc].t, tree[n.rc].t);
39     }
40 };

```

---

### 3.8.7 Dynamic segment tree Sparse, i.e., larges values, i.e., not storred as an array

---

```

1 #include "../header.h"
2 using T=ll; using U=ll;    // exclusive
   right bounds
3 T t_id; U u_id;
4 T op(T a, T b){ return a+b; }
5 void join(U &a, U b){ a+=b; }
6 void apply(T &t, U u, int x){ t+=x*u; }
7 T part(T t, int r, int p){ return t/r*p; }
8 struct DynamicSegmentTree {
9     struct Node { int l, r, lc, rc; T t; U u;
10         Node(int l, int r):l(l),r(r),lc(-1),rc(-1),t(t_id),u(u_id){}
11     };
12     vector<Node> tree;
13     DynamicSegmentTree(int N) { tree.push_back({0,N}); }
14     void push(Node &n, U u){ apply(n.t, u, n.r-n.l);
15         join(n.u,u); }
16     void push(Node &n){push(tree[n.lc],n.u);push(tree[n.rc],n.u);n.u=u_id;}
17     T query(int l, int r, int i = 0) { auto &n = tree[i];
18         if(r <= n.l || n.r <= l) return t_id;
19         if(l <= n.l && n.r <= r) return n.t;
20         if(n.lc < 0) return part(n.t, n.r-n.l, min(n.r,r)-max(n.l,l));
21         return push(n, op(query(l,r,n.lc),query(l,r,n.rc)));
22     }
23     void update(int l, int r, U u, int i = 0) {
24         auto &n = tree[i];
25         if(r <= n.l || n.r <= l) return;
26         if(l <= n.l && n.r <= r) return push(n,u);
27         if(n.lc < 0) { int m = (n.l + n.r) / 2;
28             n.lc = tree.size(); n.rc = n.lc+1;
29             tree.push_back({tree[i].l, m}); tree.push_back({m, tree[i].r});
30         }
31         push(tree[i]); update(l,r,u,tree[i].lc);
32         update(l,r,u,tree[i].rc);
33         tree[i].t = op(tree[tree[i].lc].t, tree[tree[i].rc].t);
34     }
35 };

```

---

### 3.8.8 Suffix tree

---

```

1 #include "../header.h"
2 using T = char;
3 using M = map<T,int>; // or array<T,ALPHABET_SIZE>

```

---

```

4 using V = string; // could be vector<T> as well
5 using It = V::const_iterator;
6 struct Node{
7     It b, e; M edges; int link; // end is exclusive
8     Node(It b, It e) : b(b), e(e), link(-1) {}
9     int size() const { return e-b; }
10 };
11 struct SuffixTree{
12     const V &s; vector<Node> t;
13     int root,n,len,remainder,llink; It edge;
14     SuffixTree(const V &s) : s(s) { build(); }
15     int add_node(It b, It e){ return t.push_back({b,e}), t.size()-1; }
16     int add_node(It b){ return add_node(b,s.end()); }
17     void link(int node){ if(llink) t[llink].link = node; llink = node; }
18     void build(){
19         len = remainder = 0; edge = s.begin();
20         n = root = add_node(s.begin(), s.begin());
21         for(auto i = s.begin(); i != s.end(); ++i){
22             ++remainder; llink = 0;
23             while(remainder){
24                 if(len == 0) edge = i;
25                 if(t[n].edges[*edge] == 0){
26                     t[n].edges[*edge] = add_node(i); link(n);
27                 } else {
28                     auto x = t[n].edges[*edge];
29                     if(len >= t[x].size()){
30                         len -= t[x].size(); edge += t[x].size(); n = x;
31                         continue;
32                     }
33                     if(*(t[x].b + len) == *i){
34                         ++len; link(n); break;
35                     }
36                     auto split = add_node(t[x].b, t[x].b + len);
37                     t[n].edges[*edge] = split;
38                     t[x].b += len;
39                     t[split].edges[*i] = add_node(i);
40                     t[split].edges[*t[x].b] = x;
41                     link(split);
42                 }
43             }
44             --remainder;
45             if(n == root && len > 0)
46                 --len, edge = i - remainder + 1;
47             else n = t[n].link > 0? t[n].link: root;
48         }
49     }
50 };

```

---

### 3.8.9 UnionFind

```

1 #include "header.h"
2 struct UnionFind {
3     std::vector<int> par, rank, size;
4     int c;
5     UnionFind(int n) : par(n), rank(n, 0), size(n,
6         1), c(n) {
7         for(int i = 0; i < n; ++i) par[i] = i;
8     }
9     int find(int i) { return (par[i] == i ? i : (
10         par[i] = find(par[i]))); }
11     bool same(int i, int j) { return find(i) ==
12         find(j); }
13     int get_size(int i) { return size[find(i)]; }
14     int count() { return c; }
15     int merge(int i, int j) {
16         if((i = find(i)) == (j = find(j))) return -1;
17         --c;
18         if(rank[i] > rank[j]) swap(i, j);
19         par[i] = j;
20         size[j] += size[i];
21         if(rank[i] == rank[j]) rank[j]++;
22         return j;
23     }
24 };

```

**3.8.10 Indexed set** Similar to set, but allows accessing elements by index using `find_by_order()` in  $O(\log n)$

```

1 #include "../header.h"
2 #include <ext/pb_ds/assoc_container.hpp>
3 using namespace __gnu_pbds;
4 using namespace std;
5 typedef tree<int, null_type, less<int>, rb_tree_tag,
6     tree_order_statistics_node_update>
7     indexed_set;

```

## 4 Other Mathematics

### 4.1 Helpful functions

**4.1.1 Euler's Totient Function**  $n = p_1^{k_1-1} \cdot (p_1 - 1) \cdot \dots \cdot p_r^{k_r-1} \cdot (p_r - 1)$ , where  $p_1^{k_1} \cdot \dots \cdot p_r^{k_r}$  is the prime factorization of  $n$ .

```

1 # include "header.h"
2 ll phi(ll n) { // \Phi(n)
3     ll ans = 1;
4     for (ll i = 2; i*i <= n; i++) {
5         if (n % i == 0) {

```

```

6         ans *= i-1;
7         n /= i;
8         while (n % i == 0) {
9             ans *= i;
10            n /= i;
11        }
12    }
13 }
14 if (n > 1) ans *= n-1;
15 return ans;
16 }
17 vi phis(int n) { // All \Phi(i) up to n
18     vi phi(n+1, 0LL);
19     iota(phi.begin(), phi.end(), 0LL);
20     for (ll i = 2LL; i <= n; ++i)
21         if (phi[i] == i)
22             for (ll j = i; j <= n; j += i)
23                 phi[j] -= phi[j] / i;
24     return phi;
25 }

```

#### 4.1.2 Totient (again but .py)

```

1 def totatives(n):
2     if n == 1:
3         return 1
4     phi = int(n > 1 and n)
5     for p in range(2, int(n**.5) + 1):
6         if not n % p:
7             phi -= phi // p
8             while not n % p:
9                 n //= p
10    #if n is > 1 it means it is prime
11    if n > 1: phi -= phi // n
12    return phi

```

**Formulas**  $\Phi(n)$  counts all numbers in  $1, \dots, n-1$  coprime to  $n$ .  
 $a^{\varphi(n)} \equiv 1 \pmod n$ ,  $a$  and  $n$  are coprimes.  
 $\forall e > \log_2 m : n^e \pmod m = n^{\Phi(m)+e} \pmod m$   
 $\gcd(m, n) = 1 \Rightarrow \Phi(m \cdot n) = \Phi(m) \cdot \Phi(n)$ .

**4.1.3 Pascal's trinagle**  $\binom{n}{k}$  is  $k$ -th element in the  $n$ -th row, indexing both from 0

```

1 #include "header.h"
2 void printPascal(int n) {
3     for (int line = 1; line <= n; line++) {
4         int C = 1; // used to represent C(line, i)

```

```

5         for (int i = 1; i <= line; i++) {
6             cout << C << " ";
7             C = C * (line - i) / i;
8         }
9         cout << "\n";
10    }
11 }

```

## 4.2 Theorems and definitions

**Subfactorial (Derangements)** Permutations of a set such that none of the elements appear in their original position:

$$!n = n! \sum_{i=0}^n \frac{(-1)^i}{i!}$$

$$!(0) = 1, !n = n!(n-1) + (-1)^n$$

$$!n = (n-1)(!(n-1) + !(n-2)) = \left\lfloor \frac{n!}{e} \right\rfloor \quad (1)$$

$$!n = 1 - e^{-1}, n \rightarrow \infty \quad (2)$$

### Binomials and other partitionings

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} = \prod_{i=1}^k \frac{n-i+1}{i}$$

This last product may be computed incrementally since any product of  $k'$  consecutive values is divisible by  $k'!$ .

Basic identities: The hockeystick identity:

$$\sum_{k=r}^n \binom{k}{r} = \binom{n+1}{r+1}$$

or

$$\sum_{k \leq n} \binom{r+k}{k} = \binom{r+n+1}{n}$$

Also

$$\sum_{k=0}^n \binom{k}{m} = \binom{n+1}{m+1}$$

$$\sum_{i=0}^n \binom{n}{i} = 2^n$$



For  $n, m \geq 0$  and  $p$  prime: write  $n, m$  in base  $p$ , i.e.  $n = n_k p^k + \dots + n_1 p + n_0$  and  $m = m_k p^k + \dots + m_1 p + m_0$ . Then by Lucas theorem we have  $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$ , with the convention that  $n_i < m_i \implies \binom{n_i}{m_i} = 0$ .

**Fibonacci** (See also number theory section)

$$\sum_{0 \leq k \leq n} \binom{n-k}{k} = F_{n+1}$$

$$F_n = \frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1-\sqrt{5}}{2} \right)^n$$

$$\sum_{i=1}^n F_i = F_{n+2} - 1, \quad \sum_{i=1}^n F_i^2 = F_n F_{n+1}$$

$$\gcd(F_m, F_n) = F_{\gcd(m, n)}$$

$$\gcd(F_n, F_{n+1}) = \gcd(F_n, F_{n+2}) = 1$$

**Bit stuff**  $a + b = a \oplus b + 2(a \& b) = a|b + a \& b$ .

$k$ th bit is set in  $x$  iff  $x \bmod 2^{k-1} \geq 2^k$ , or iff  $x \bmod 2^{k-1} - x \bmod 2^k \neq 0$  (i.e.  $= 2^k$ ) It comes handy when you need to look at the bits of the numbers which are pair sums or subset sums etc.

$$n \bmod 2^i = n \& (2^i - 1).$$

$$\forall k: 1 \oplus 2 \oplus \dots \oplus (4k-1) = 0$$

### 4.3 Geometry Formulas

$$\text{Euler:} \quad 1 + CC = V - E + F$$

$$\text{Pick:} \quad \text{Area} = \text{itr pts} + \frac{\text{bdry pts}}{2} - 1$$

Given a non-self-intersecting closed polygon on  $n$  vertices, given as  $(x_i, y_i)$ , its centroid  $(C_x, C_y)$  is given as:

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

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

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) = \text{polygon area}$$

**Inclusion-Exclusion** For appropriate  $f$  compute  $\sum_{S \subseteq T} (-1)^{|T \setminus S|} f(S)$ , or if only the size of  $S$  matters,  $\sum_{s=0}^n (-1)^{n-s} \binom{n}{s} f(s)$ . In some contexts we might use Stirling numbers, not binomial coefficients!

Some useful applications:

**Graph coloring** Let  $I(S)$  count the number of independent sets contained in  $S \subseteq V$  ( $I(\emptyset) = 1$ ,  $I(S) = I(S \setminus v) + I(S \setminus N(v))$ ). Let  $c_k = \sum_{S \subseteq V} (-1)^{|V \setminus S|} I(S)$ . Then  $V$  is  $k$ -colorable iff  $v > 0$ . Thus we can compute the chromatic number of a graph in  $O^*(2^n)$  time.

**Burnside's lemma** Given a group  $G$  acting on a set  $X$ , the number of elements in  $X$  up to symmetry is

$$\frac{1}{|G|} \sum_{g \in G} |X^g|$$

with  $X^g$  the elements of  $X$  invariant under  $g$ . For example, if  $f(n)$  counts “configurations” of some sort of length  $n$ , and we want to count them up to rotational symmetry using  $G = \mathbb{Z}/n\mathbb{Z}$ , then

$$g(n) = \frac{1}{n} \sum_{k=0}^{n-1} f(\gcd(n, k)) = \frac{1}{n} \sum_{k|n} f(k) \phi(n/k)$$

I.e. for coloring with  $c$  colors we have  $f(k) = c^k$ .

Relatedly, in Pólya's enumeration theorem we imagine  $X$  as a set of  $n$  beads with  $G$  permuting the beads (e.g. a necklace, with  $G$  all rotations and reflections of the  $n$ -cycle, i.e. the dihedral group  $D_n$ ). Suppose further that we had  $Y$  colors, then the number of  $G$ -invariant colorings  $Y^X/G$  is counted by

$$\frac{1}{|G|} \sum_{g \in G} |Y|^{c(g)}$$

with  $c(g)$  counting the number of cycles of  $g$  when viewed as a permutation of  $X$ . We can generalize this to a weighted version: if the color  $i$  can occur exactly  $r_i$  times, then this is counted by the coefficient of  $t_1^{r_1} \dots t_n^{r_n}$  in the polynomial

$$Z(t_1, \dots, t_n) = \frac{1}{|G|} \sum_{g \in G} \prod_{m \geq 1} (t_1^m + \dots + t_n^m)^{c_m(g)}$$

where  $c_m(g)$  counts the number of length  $m$  cycles in  $g$  acting as a permutation on  $X$ . Note we get the original formula by setting all  $t_i = 1$ . Here  $Z$  is the cycle index. Note: you can cleverly deal with even/odd sizes by setting some  $t_i$  to  $-1$ .

**Lucas Theorem** If  $p$  is prime, then:

$$\frac{p^a}{k} \equiv 0 \pmod{p}$$

Thus for non-negative integers  $m = m_k p^k + \dots + m_1 p + m_0$  and  $n = n_k p^k + \dots + n_1 p + n_0$ :

$$\frac{m}{n} = \prod_{i=0}^k \frac{m_i}{n_i} \pmod{p}$$

Note: The fraction's mean integer division.

### 4.4 Recurrences

If  $a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$ , and  $r_1, \dots, r_k$  are distinct roots of  $x^k - c_1 x^{k-1} - \dots - c_k$ , there are  $d_1, \dots, d_k$  s.t.

$$a_n = d_1 r_1^n + \dots + d_k r_k^n.$$

Non-distinct roots  $r$  become polynomial factors, e.g.  $a_n = (d_1 n + d_2) r^n$ .

### 4.5 Sums

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$1^4 + 2^4 + 3^4 + \dots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

### 4.6 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, \quad (-\infty < x < \infty)$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, \quad (-1 < x \leq 1)$$

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots, \quad (-1 \leq x \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, \quad (-\infty < x < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, \quad (-\infty < x < \infty)$$



## 4.7 Quadrilaterals

With side lengths  $a, b, c, d$ , diagonals  $e, f$ , diagonals angle  $\theta$ , area  $A$  and magic flux  $F = b^2 + d^2 - a^2 - c^2$ :

$$4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is  $180^\circ$ ,  $ef = ac + bd$ , and  $A = \sqrt{(p-a)(p-b)(p-c)(p-d)}$ .

## 4.8 Triangles

Side lengths:  $a, b, c$

Semiperimeter:  $p = \frac{a+b+c}{2}$

Area:

$$[ABC] = rp = \frac{1}{2}ab \sin \gamma$$

$$= \frac{abc}{4R} = \sqrt{p(p-a)(p-b)(p-c)} = \frac{1}{2} |(B-A, C-A)^T|$$

Circumradius:  $R = \frac{abc}{4A}$ , Inradius:  $r = \frac{A}{p}$

Length of median (divides triangle into two equal-area triangles):  $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Length of bisector (divides angles in two):  $s_a =$

$$\sqrt{bc \left[ 1 - \left( \frac{a}{b+c} \right)^2 \right]}$$

Law of tangents:  $\frac{a+b}{a-b} = \frac{\tan \frac{\alpha+\beta}{2}}{\tan \frac{\alpha-\beta}{2}}$

## 4.9 Trigonometry

$$\tan(v+w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$

$$\sin v + \sin w = 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$\cos v + \cos w = 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$(V+W) \tan(v-w)/2 = (V-W) \tan(v+w)/2$$

where  $V, W$  are lengths of sides opposite angles  $v, w$ .

$$a \cos x + b \sin x = r \cos(x - \phi)$$

$$a \sin x + b \cos x = r \sin(x + \phi)$$

where  $r = \sqrt{a^2 + b^2}$ ,  $\phi = \text{atan2}(b, a)$ .

## 4.10 Combinatorics

Combinations and Permutations

$$P(n, r) = \frac{n!}{(n-r)!}$$

$$C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

$$C(n, r) = C(n, n-r)$$

## 4.11 Cycles

Let  $g_S(n)$  be the number of  $n$ -permutations whose cycle lengths all belong to the set  $S$ . Then

$$\sum_{n=0}^{\infty} g_S(n) \frac{x^n}{n!} = \exp \left( \sum_{n \in S} \frac{x^n}{n} \right)$$

## 4.12 Labeled unrooted trees

# on  $n$  vertices:  $n^{n-2}$

# on  $k$  existing trees of size  $n_i$ :  $n_1 n_2 \dots n_k n^{k-2}$

# with degrees  $d_i$ :  $(n-2)! / ((d_1-1)! \dots (d_n-1)!)$

## 4.13 Partition function

Number of ways of writing  $n$  as a sum of positive integers, disregarding the order of the summands.

$$p(0) = 1, \quad p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k-1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

$n$	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	$\sim 2e5$	$\sim 2e8$

## 4.14 Numbers

**Bernoulli numbers** EGF of Bernoulli numbers is  $B(t) = \frac{t}{e^t - 1}$  (FFT-able).  $B[0, \dots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \dots]$

Sums of powers:

$$\sum_{i=1}^n i^m = \frac{1}{m+1} \sum_{k=0}^m \binom{m+1}{k} B_k \cdot (n+1)^{m+1-k}$$

Euler-Maclaurin formula for infinite sums:

$$\sum_{i=m}^{\infty} f(i) = \int_m^{\infty} f(x) dx - \sum_{k=1}^{\infty} \frac{B_k}{k!} f^{(k-1)}(m) \\ \approx \int_m^{\infty} f(x) dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f'''(m)}{720} + O(f^{(5)}(m))$$

**Stirling's numbers First kind:**  $S_1(n, k)$  count permutations on  $n$  items with  $k$  cycles.  $S_1(n, k) = S_1(n-1, k-1) + (n-1)S_1(n-1, k)$  with  $S_1(0, 0) = 1$ . Note:

$$\sum_{k=0}^n S_1(n, k) x^k = x(x+1) \dots (x+n-1)$$

$$\sum_{k=0}^n S_1(n, k) = n!$$

$$S_1(8, k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$$

$$S_1(n, 2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots$$

**Second kind:**  $S_2(n, k)$  count partitions of  $n$  distinct elements into exactly  $k$  non-empty groups.

$$S_2(n, k) = S_2(n-1, k-1) + k S_2(n-1, k)$$

$$S_2(n, 1) = S_2(n, n) = 1$$

$$S_2(n, k) = \frac{1}{k!} \sum_{i=0}^k (-1)^{k-i} \binom{k}{i} i^n$$

**Catalan Numbers** - Number of correct bracket sequence consisting of  $n$  opening and  $n$  closing brackets.

The number of ways to completely parenthesize  $n+1$  factors.

The number of triangulations of a convex polygon with  $n+2$  sides (i.e. the number of partitions of polygon into disjoint triangles by using the diagonals).

The number of ways to connect the  $2n$  points on a circle to form  $n$  disjoint i.e. non-intersecting chords.

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

$$C_0 = 1, \quad C_1 = 1, \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

**Narayana numbers** The number of expressions containing  $n$  pairs of parentheses, which are correctly matched and which contain  $k$  distinct nestings.

$$N(n, k) = \frac{1}{n} \frac{n}{k} \frac{n}{k-1}$$

**Eulerian numbers** Number of permutations  $\pi \in S_n$  in which exactly  $k$  elements are greater than the previous element.  $k$   $j$ :s s.t.  $\pi(j) > \pi(j+1)$ ,  $k+1$   $j$ :s s.t.  $\pi(j) \geq j$ ,  $k$   $j$ :s s.t.  $\pi(j) > j$ .

$$E(n, k) = (n-k)E(n-1, k-1) + (k+1)E(n-1, k)$$

$$E(n, 0) = E(n, n-1) = 1$$

$$E(n, k) = \sum_{j=0}^k (-1)^j \binom{n+1}{j} (k+1-j)^n$$

**Bell numbers** Total number of partitions of  $n$  distinct elements.  $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$ . For  $p$  prime,

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod{p}$$

**Catalan numbers**

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

$$C_0 = 1, C_{n+1} = \frac{2(2n+1)}{n+2} C_n, C_{n+1} = \sum C_i C_{n-i}$$

$$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots$$

- sub-diagonal monotone paths in an  $n \times n$  grid.
- strings with  $n$  pairs of parenthesis, correctly nested.
- binary trees with  $n+1$  leaves (0 or 2 children).
- ordered trees with  $n+1$  vertices.
- ways a convex polygon with  $n+2$  sides can be cut into triangles by connecting vertices with straight lines.
- permutations of  $[n]$  with no 3-term increasing subseq.

## 4.15 Probability

Stochastic variables

$$P(X=r) = C(n, r) \cdot p^r \cdot (1-p)^{n-r}$$

**Bayes' Theorem**  $P(B|A) = \frac{P(A|B)P(B)}{P(A)}$

$$P(B|A) = \frac{P(A|B)P(B)}{P(A|B)P(B) + P(A|\bar{B})P(\bar{B})}$$

$$P(B_k|A) = \frac{P(A|B_k)P(B_k)}{P(A|B_1)P(B_1) + \dots + P(A|B_n)P(B_n)}$$

**Expectation** Let  $X$  be a discrete random variable with probability  $p_X(x)$  of assuming the value  $x$ . It will then have an expected value (mean)  $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$  and variance  $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$  where  $\sigma$  is the standard deviation. If  $X$  is instead continuous it will have a probability density function  $f_X(x)$  and the sums above will instead be integrals with  $p_X(x)$  replaced by  $f_X(x)$ .

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent  $X$  and  $Y$ ,

$$V(aX + bY) = a^2 V(X) + b^2 V(Y).$$

## 4.16 Number Theory

**Bezout's Theorem**

$$a, b \in \mathbb{Z}^+ \implies \exists s, t \in \mathbb{Z} : \gcd(a, b) = sa + tb$$

**Bézout's identity** For  $a \neq 0$ ,  $b \neq 0$ , then  $d = \gcd(a, b)$  is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If  $(x, y)$  is one solution, then all solutions are given by

$$\left( x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)} \right), \quad k \in \mathbb{Z}$$

**Partial Coprime Divisor Property**

$$(\gcd(a, b) = 1) \wedge (a \mid bc) \implies (a \mid c)$$

**Coprime Modulus Equivalence Property**

$$(\gcd(c, m) = 1) \wedge (ac \equiv bc \pmod{m}) \implies (a \equiv b \pmod{m})$$

**Fermat's Little Theorem**

$$(\text{prime}(p)) \wedge (p \nmid a) \implies (a^{p-1} \equiv 1 \pmod{p})$$

$$(\text{prime}(p)) \implies (a^p \equiv a \pmod{p})$$

**Pythagorean Triples** The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \quad b = k \cdot (2mn), \quad c = k \cdot (m^2 + n^2),$$

with  $m > n > 0$ ,  $k > 0$ ,  $m \perp n$ , and either  $m$  or  $n$  even.

**Primes**  $p = 962592769$  is such that  $2^{21} \mid p-1$ , which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

Primitive roots exist modulo any prime power  $p^a$ , except for  $p = 2, a > 2$ , and there are  $\phi(\phi(p^a))$  many. For  $p = 2, a > 2$ , the group  $\mathbb{Z}_{2^a}^\times$  is instead isomorphic to  $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$ .

**Estimates**  $\sum_{d|n} d = O(n \log \log n)$ .

The number of divisors of  $n$  is at most around 100 for  $n < 5e4$ , 500 for  $n < 1e7$ , 2000 for  $n < 1e10$ , 200 000 for  $n < 1e19$ .

**Mobius Function**

$$\mu(n) = \begin{cases} 0 & n \text{ is not square free} \\ 1 & n \text{ has even number of prime factors} \\ -1 & n \text{ has odd number of prime factors} \end{cases}$$

Mobius Inversion:

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d) g(n/d)$$

Other useful formulas/forms:

$$\sum_{d|n} \mu(d) = [n=1] \text{ (very useful)}$$

$$g(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} \mu(d/n) g(d)$$

$$g(n) = \sum_{1 \leq m \leq n} f\left(\left\lfloor \frac{n}{m} \right\rfloor\right) \Leftrightarrow f(n) = \sum_{1 \leq m \leq n} \mu(m) g\left(\left\lfloor \frac{n}{m} \right\rfloor\right)$$

**4.17 Discrete distributions**

**Binomial distribution** The number of successes in  $n$  independent yes/no experiments, each which yields success with probability  $p$  is  $\text{Bin}(n, p)$ ,  $n = 1, 2, \dots$ ,  $0 \leq p \leq 1$ .

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mu = np, \sigma^2 = np(1-p)$$

$\text{Bin}(n, p)$  is approximately  $\text{Po}(np)$  for small  $p$ .

**First success distribution** The number of trials needed to get the first success in independent yes/no experiments, each which yields success with probability  $p$  is  $\text{Fs}(p)$ ,  $0 \leq p \leq 1$ .

$$p(k) = p(1-p)^{k-1}, k = 1, 2, \dots$$

$$\mu = \frac{1}{p}, \sigma^2 = \frac{1-p}{p^2}$$

**Poisson distribution** The number of events occurring in a fixed period of time  $t$  if these events occur with a known average rate  $\kappa$  and independently of the time since the last event is  $\text{Po}(\lambda)$ ,  $\lambda = t\kappa$ .

$$p(k) = e^{-\lambda} \frac{\lambda^k}{k!}, k = 0, 1, 2, \dots$$

$$\mu = \lambda, \sigma^2 = \lambda$$

**4.18 Continuous distributions**

**Uniform distribution** If the probability density function is constant between  $a$  and  $b$  and 0 elsewhere it is  $\text{U}(a, b)$ ,  $a < b$ .

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2}, \sigma^2 = \frac{(b-a)^2}{12}$$

**Exponential distribution** The time between events in a Poisson process is  $\text{Exp}(\lambda)$ ,  $\lambda > 0$ .

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2}$$

**Normal distribution** Most real random values with mean  $\mu$  and variance  $\sigma^2$  are well described by  $\mathcal{N}(\mu, \sigma^2)$ ,  $\sigma > 0$ .

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If  $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$  and  $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$  then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$