

Codesnatchers



Contents

1	Template	2
2	Data structures	2
2.1	STL Algorithms	2
2.2	Binary Search	3
2.3	Simplified DSU (Stolen from GGDem)	4
2.4	Disjoint Set Union	4
2.5	Segment Tree	4
2.6	Segment Tree Lazy	4
2.7	Trie	4
3	Graphs	4
3.1	Graph Transversal	4
3.1.1	BFS	4
3.1.2	DFS	5
3.1.3	BFS / DFS using queue and stack	6
3.2	Topological Sort	6
3.3	APSP: Floyd Warshall	6
3.4	SSSP	6
3.4.1	Lazy Dijkstra	6
3.4.2	Bellman-Ford	7
3.5	Strongly Connected Components: Kosaraju	7
3.6	Articulation Points and Bridges: ModTarjan	7
4	Math	7
4.1	Identities	7
4.2	Binary Exponentiation and Modular Arithmetic	7
4.2.1	Binary Exponentiation	7
4.2.2	Modular Arithmetic	7
4.3	Modular Inverse	8
4.4	Modular Binomial Coefficient and Permutations	9
4.5	Non-Mod Binomial Coefficient and Permutations	9
4.6	Modular Catalan Numbers	9
4.7	Fractional Ceiling	9
4.8	Fibonacci Numbers	9
4.9	Sieve Of Eratosthenes	9
4.10	Sieve-based Factorization	9
4.11	Cycle Finding	10
4.12	Berlekamp Massey	10
4.13	Modular Berlekamp Massey	10
4.14	Matrix exponentiation	10
4.15	Ecuaciones Diofantinas	10
4.16	Pollard-Rho, Stolen from GGDem	10
4.17	FFT, Stolen from GGDem	10
4.18	Euler Totient Function	10
5	Geometry	10
6	Strings	10
6.1	Explode by token	10
6.2	Multiple Hashings DS	10
6.3	Permute chars of string	10
6.4	Longest common subsequence	10
6.5	KMP	10
6.6	Suffix Array	10
6.7	STL Suffix Array	10
7	Classics	10
7.1	Job scheduling	10
7.1.1	One machine, linear penalty	10
7.1.2	One machine, deadlines	10
7.1.3	One machine, profit	10
7.1.4	Two machines, min time	10
8	Flow	10
8.1	Dinic	10

9 Miscellaneous	11
9.1 Policy Based Data Structures	11
9.2 Bit Manipulation	12
9.2.1 Bitmasking	12
10 Testing	13
10.1 Gen and AutoRun testcases	13
10.1.1 Gen.cpp	13
10.1.2 Stress testing	13
10.1.3 Autorun	13
10.2 Highly Composite Numbers	13

1 Template

```

1 #include <bits/stdc++.h>
2 #define endl '\n'
3 #define ll long long int
4 #define ull unsigned long long int
5 #define MOD7 1000000007
6 #define MOD9 1000000009
7 #define MAX 1000001
8
9 using namespace std;
10
11 /*-----SOLBEGIN-----*/
12
13 void solve() {
14     return;
15 }
16
17 int main() {
18     ios_base::sync_with_stdio(0);
19     cin.tie(0);
20
21     int t = 1; cin >> t;
22     while (t--) solve();
23
24     return 0;
25 }

```

2 Data structures

2.1 STL Algorithms

STL stands for Standard Template Library. It is a library that provides several generic classes and functions, allowing programmers to manipulate data structures in an easy and efficient way. The STL provides a range of algorithms which can be used to manipulate data stored in containers. The following list shows some of the algorithms provided by the STL and its functions:

Non-Manipulating Algorithms

- **sort(first_iterator, last_iterator)** - Sorts the elements in the range [first, last) in ascending order.
- **sort(frst_iterator, last_iterator, greater<int>())** - Sorts elements inside the vector, in descending order.

- **reverse(first_iterator, last_iterator)** - Reverses elements inside a vector.
- ***max_element(first_iterator, last_iterator)** - Finds the maximum element of a vector.
- ***min_element(first_iterator, last_iterator)** - Finds the minimum element of a vector.
- **accumulate(first_iterator, last_iterator, initial value of sum)** - Summates all the vector elements.
- **count(first_iterator, last_iterator, x)** - Counts all occurrences 'x' inside a vector. It has a linear complexity over the quantity of occurrences inside the container; if you fill a set with just one element, the count() function over that element will have an overall complexity of $O(|set|)$ where $|set|$ is the cardinality of the set.
- **find(first_iterator, last_iterator, x)** - Returns an iterator to the first occurrence of 'x' in vector and points to last address if the element is not present.
- **binary_search(first_iterator, last_iterator, x)** - Tests if 'x' exists in sorted vector or not.
- **lower_bound(first_iterator, last_iterator, x)** - Returns an element pointing to the first element in range [first, last), which has a value less than 'x'. Notice that it has a linear complexity over sets and multisets, to avoid this it is necessary to use the member method of those containers.
- **upper_bound(first_iterator, last_iterator, x)** - Returns an element pointing to the first element in range [first, last), which has a value greater than 'x'.

Manipulating Algorithms

- **arr.erase(position to delete)** - Erases selected element in vector and shifts and resizes it accordingly.
- **arr.erase(unique(arr.begin(), arr.end()), arr.end())** - Erases the duplicate occurrences in sorted vector in a single line.
- **next_permutation(first_iterator, last_iterator)** - Modifies the vector to its next permutation.
- **prev_permutation(first_iterator, last_iterator)** - Modifies the vector to its previous permutation.
- **distance(first_iterator, desired_iterator)** - Returns the distance of the desired position from the first iterator to a desired one.

2.2 Binary Search

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  vector<int> vec;
5
6  int binary_search_first_occurrence(const vector<int>& vec, int value) {
7      // Binary search algorithm finds the first occurrence of a value in
8      // a sorted vector
9      // Declare left and right pointers
10     int left = 0;
11     int right = vec.size() - 1;
12     int result = -1;
13     // While left and right pointers do not cross, keep searching
14     while (left <= right) {
15         // Calculate the middle element of the vector
16         int mid = left + (right - left) / 2;
17         // If the middle element is the value we are looking for, return
18         // its index
19         if (vec[mid] == value) {
20             result = mid;
21             // left = mid + 1; // Continue searching in the right half
22             // (for last occurrence)
23             right = mid - 1; // Continue searching in the left half
24             // If the middle element is smaller than the value we are
25             // looking for, search in the right half
26         } else if (vec[mid] < value) {
27             left = mid + 1;
28             // If the middle element is greater than the value we are
29             // looking for, search in the left half
30         } else {
31             right = mid - 1;
32         }
33     }
34     return result; // Returns -1 if value is not found
35 }
36
37 int main() {
38     // Assign the variable value to the value you want to search
39     int elements, value = 0;
40     cin >> elements;
41     // Read the elements of the vector

```

```

37     for (int i = 0; i < elements; i++) {
38         int x;
39         cin >> x;
40         vec.push_back(x);
41     }
42     cout << binary_search_first_occurrence(vec, value);
43
44     return 0;
45 }

```

2.3 Simplified DSU (Stolen from GGDem)

2.4 Disjoint Set Union

2.5 Segment Tree

```

1 // MAXN = 2^k, n = size of the array
2 #define MAXN 262160
3
4 // Define the function fst to be used in the segment tree
5 int stsize;
6 long long int neut;
7 int n;
8 // Storing the segment tree
9 long long int* st = new long long int[2 * MAXN - 1]();
10 // Combine the results of two child nodes in the segment tree
11 long long int fst(long long int a, long long int b);
12
13 // Recursive function to build the segment tree, combining the results
   of the two child nodes
14 long long int build(int sti, int csize){
15     if(csize == 1) return st[sti];
16     // Combine the results of the two child nodes
17     return st[sti] = fst(build(sti * 2 + 1, csize / 2), build(sti * 2 +
18     2, csize / 2));
19 }
20
21 // Initialize the segment tree, setting all the nodes to a neutral value
22 void ininit(){
23     for(int i = 0; i < stsize; i++) st[i] = neut;
24     /*
25     int d = 0;
26     for(int i = stsize-n; i<stsize && d<n; i++){
27         st[i] = arr[d]; d++;

```

```

27     }
28     */
29     build(0,n);
30 }
31
32 // Update the value of a node in the segment tree and all its ancestor
   nodes
33 void upd(int ind, long long int val){
34     ind = stsize - n + ind;
35     st[ind] = val; ind--; ind /= 2;
36     while (true) {
37         st[ind] = fst(st[ind * 2 + 1], st[ind * 2 + 2]);
38         ind--;
39         if (ind < 0) break;
40         ind /= 2;
41     }
42 }
43
44 // Process a query in the segment tree, from index l to r
45 long long int rqu(int l, int r, int sti, int ls, int rs){
46     if (l <= ls && rs <= r) return st[sti];
47     if (r < ls || l > rs) return neut;
48     int m = (rs + ls) / 2;
49     return fst(rqu(l, r, sti * 2 + 1, ls, m), rqu(l, r, sti * 2 + 2, m +
50     1, rs));
51 }
52
53 long long int query(int l, int r){
54     return rqu(l, r, 0, 0, n - 1);
55 }

```

2.6 Segment Tree Lazy

2.7 Trie

3 Graphs

3.1 Graph Transversal

3.1.1 BFS

```

1 #include <bits/stdc++.h>
2 using namespace std;
3

```

```
4 vector<bool> visited;
5 vector<vector<int>> adj;
6
7 void breadth_first_search(int node) {
8     // BFS requires queue data structure, starting from a given initial
9     // node
10    queue<int> q;
11    q.push(node);
12    visited[node] = true;
13    // While queue is not empty, pop the first element and push its
14    // children
15    while (!q.empty()) {
16        int v = q.front();
17        cout << v << " ";
18        q.pop();
19        // Push all children of v
20        for (int u : adj[v]) {
21            // If not visited, push and mark as visited
22            if (!visited[u]) {
23                q.push(u);
24                visited[u] = true;
25            }
26        }
27    }
28
29 int main() {
30     int nodes, edges;
31     cin >> nodes >> edges;
32     // Initialize visited and adjacency list
33     visited.assign(nodes, false);
34     adj.assign(nodes, vector<int>());
35     int u, v;
36     // Values of nodes, given as pairs
37     for (int i = 0; i < edges; i++) {
38         cin >> u >> v;
39         adj[u].push_back(v);
40         adj[v].push_back(u); // <- Assuming undirected graph
41     }
42     breadth_first_search(0); // Start BFS from node x
43
44     return 0;
45 }
```

3.1.2 DFS

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 vector<bool> visited;
5 vector<vector<int>> adj;
6
7 void depth_first_search(int node) {
8     // DFS requires stack data structure, starting from a given initial
9     // node
10    visited[node] = true;
11    cout << node << '\n';
12    // For each child of node, if it hasn't been visited, call DFS
13    // function
14    for(int i = 0; i < adj[node].size(); i++) {
15        int child = adj[node][i];
16        if(!visited[child]) {
17            depth_first_search(child);
18        }
19    }
20 }
21
22 int main() {
23     int nodes, edges;
24     cin >> nodes >> edges;
25     // Initialize visited and adjacency list
26     visited.assign(nodes, false);
27     adj.assign(nodes, vector<int>());
28     // Values of nodes, given as pairs
29     for(int i = 0; i < edges; i++) {
30         int u, v;
31         cin >> u >> v;
32         adj[u].push_back(v);
33         adj[v].push_back(u); // <- Assuming undirected graph
34     }
35     // For each node, if it hasn't been visited, call DFS function
36     for(int i = 0; i < nodes; i++) {
37         if(!visited[i]) {
38             depth_first_search(i);
39         }
40     }
41 }
```

```

40     return 0;
41 }

```

3.1.3 BFS / DFS using queue and stack

```

1  #define GS 400040
2  vector<int> graph[GS];
3  bitset <GS> vis;
4
5  void bfs(int curr) {
6      queue<int> q;
7      q.push(curr);
8      while (q.size()) {
9          curr = q.front(); q.pop();
10         if (!q[curr]) {
11             q[curr] = 1;
12             for (int h : graph[curr]) q.push(h);
13         }
14     }
15 }
16
17 void dfs(int curr) {
18     stack<int> s;
19     s.push(curr);
20     while (s.size()){
21         curr = s.top(); s.pop();
22         if (!vis[curr]) {
23             vis[curr] = 1;
24             for (int h : graph[curr]) s.push(h);
25         }
26     }
27 }

```

3.2 Topological Sort

3.3 APSP: Floyd Warshall

3.4 SSSP

3.4.1 Lazy Dijkstra

```

1  // Lazy version of Dijkstra's algorithm usign priority queue
2  // Works with negative weights while there are no negative cycles
3  // If there are any negative cycles, the algorithm will not work
4  #include <bits/stdc++.h>

```

```

5  #define GS 1000
6  #define INF 100000000
7  using namespace std;
8
9  // Define the graph and the distance array
10 vector<pair<int, int>> graph[GS];
11 int distance[GS];
12
13 void dijkstra(int origin, int size) {
14     // Set all distances to INF
15     for (int i = 0; i <= size; i++) distance[i] = INF;
16     // Create the priority queue and the current pair
17     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<
18         int, int>>> pq;
19     pair<int, int> current;
20
21     // Set the distance to the origin to 0 and push it to the queue
22     pq.push(make_pair(0, origin));
23
24     // While the queue is not empty, get the top element and update the
25     // distances
26     while (!pq.empty()) {
27         // Get the top element and pop it
28         current = pq.top();
29         pq.pop();
30
31         // If the distance is already smaller, continue to next
32         // iteration
33         if (distance[current.second] < current.first) continue;
34         // Update the distance
35         distance[current.second] = current.first;
36
37         // Iterate over the neighbors and update the distances
38         for (pair<int, int> neighbor : graph[current.second]) {
39             // If the new distance is smaller, push it to the queue
40             if ((neighbor.second + current.first) < distance[neighbor.
41                 first]) {
42                 pq.push(make_pair(neighbor.second + current.first,
43                     neighbor.first));
44             }
45         }
46     }
47 }

```

3.4.2 Bellman-Ford

3.5 Strongly Connected Components: Kosaraju

3.6 Articulation Points and Bridges: ModTarjan

4 Math

4.1 Identities

Coefficientes binomiales.

$$(a+b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k$$

$$\binom{n}{k} = \binom{n}{n-k}$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$k \binom{n}{k} = n \binom{n-1}{k-1}$$

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

$$\sum_{k=0}^n (-1)^k \binom{n}{k} = 0$$

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

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

Números Catalanés.

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

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

$$C_n \sim \frac{4^n}{n^{3/2} \sqrt{\pi}}$$

$$\Sigma(n) = O(\log(\log(n))) \text{ (number of divisors of } n)$$

$$F_{2n+1} = F_{n+1}^2 + F_n^2$$

$$F_{2n} = F_{n+1}^2 - F_{n-1}^2$$

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

$$F_{n+i} F_{n+j} - F_n F_{n+i+j} = (-1)^n F_i F_j$$

(Möbius Function)

0 if n is square-free

1 if n got even amount of distinct prime factors

0 if n got odd amount of distinct prime factors

(Möbius Inv. Formula)

Let $g(n) = \sum_{d|n} f(d)$, then $f(n) = \sum_{d|n} g(d) \mu\left(\frac{n}{d}\right)$.

Permutaciones objetos repetidos

$$P(n, k) = \frac{P(n, k)}{n_1! n_2! \dots}$$

Separadores, Ecuaciones lineales a variables = b

$$\binom{a}{b} = \binom{a+b-1}{b} = \binom{a+b-1}{a-1}$$

Teorema chino

sean $\{n_1, n_2, \dots, n_k\}$ primos relativos

$$P = n_1 \cdot n_2 \cdot \dots \cdot n_k$$

$$P_i = \frac{P}{n_i}$$

$$\begin{aligned} x &\cong a_1(n_1) \\ x &\cong a_2(n_2) \dots x \cong a_k(n_k) \\ P_1 S_1 &\cong 1(n_1) \text{ Donde } S \text{ soluciones.} \\ x &= P_1 S_1 a_1 + P_2 S_2 a_2 \dots P_k S_k a_k \end{aligned}$$

4.2 Binary Exponentiation and Modular Arithmetic

4.2.1 Binary Exponentiation

```

1 #include <bits/stdc++.h>
2 #define ll long long
3 using namespace std;
4
5 ll inf = 10000000007;
6
7 ll bitPow(ll a, ll e) {
8     ll res = 1;
9     a %= inf;
10
11     // while exponent is greater than zero
12     while (e > 0) {
13         // if exponent is odd, multiply result by base
14         if (e & 1)
15             // multiply result by base and take the remainder
16             res = (res * a) % inf;
17         // square the base and take the remainder
18         a = (a * a) % inf;
19         // divide the exponent by 2
20         e >>= 1;
21     }
22
23     return res;
24 }
```

4.2.2 Modular Arithmetic

Modular arithmetic is a system of arithmetic for integers, which considers the remainder. In modulus, numbers "wrap around" upon reaching a fixed value.

Congruence

A number $x \bmod N$ is the equivalent of the remainder of the division of x by N . Two numbers a and b are congruent modulo N if they have the same remainder upon division by N . We say that N if $a \bmod N = b \bmod N$.

- **For example:** $54 \equiv 24 \pmod{7}$

Both numbers are congruent modulo 7, since $54 \bmod 7 = 3$ and $24 \bmod 7 = 3$.

Another way of defining this is by saying that a and b are congruent modulo N if their difference $(a - b)$ is an integer multiple of n , that is, if $\frac{a-b}{n}$ has a remainder of 0.

- **For example:** $36 \equiv 10 \pmod{13}$

36 and 10 are congruent modulo 13, since their difference $36 - 10 = 26$ is a multiple of 13 ($n = 13$).

Addition

Properties of addition in Modular Arithmetic:

1. If $a + b = c$ then $a \pmod{N} + b \pmod{N} \equiv c \pmod{N}$.
2. If $a \equiv b \pmod{N}$, then $a + k \equiv b + k \pmod{N}$ for any integer k .
3. If $a \equiv b \pmod{N}$ and $c \equiv d \pmod{N}$, then $a + c \equiv b + d \pmod{N}$.
4. If $a \equiv b \pmod{N}$, then $-a \equiv -b \pmod{N}$.

- **For example:** Find the sum of 31 and 148 in modulo 24.

31 in modulo 24 is 7 and 148 in modulo 24 is 4. Thus, $31 + 148 \equiv 7 + 4 \equiv 11 \pmod{24}$.

- **Another example:** Find the remainder when $123 + 234 + 32 + 56 + 22 + 12 + 78$ is divided by 3.

We know that $123 \bmod 3 = 0$, $234 \bmod 3 = 0$, $32 \bmod 3 = 2$, $56 \bmod 3 = 2$, $22 \bmod 3 = 1$, $12 \bmod 3 = 0$, and $78 \bmod 3 = 0$. Thus, the sum of all these numbers is $0 + 0 + 2 + 2 + 1 + 0 + 0 = 5$, and $5 \bmod 3 = 2$.

Multiplication

Properties of multiplication in Modular Arithmetic:

1. If $a \cdot b = c$, then $a \pmod{N} \cdot b \pmod{N} \equiv c \pmod{N}$.
2. If $a \equiv b \pmod{N}$, then $a \cdot k \equiv b \cdot k \pmod{N}$ for any integer k .
3. If $a \equiv b$ and $c \equiv d \pmod{N}$, then $a \cdot c \equiv b \cdot d \pmod{N}$.

- **For example:** What is $(8 \cdot 16) \pmod{7}$.

Since $8 \equiv 1 \pmod{7}$ and $16 \equiv 2 \pmod{7}$, then $(8 \cdot 16) \equiv (1 \cdot 2) \equiv 2 \pmod{7}$.

- **Another example:** What is the remainder when $123 \cdot 234 \cdot 32 \cdot 56 \cdot 22 \cdot 12 \cdot 78$ is divided by 3.

We know that $123 \equiv 0$, $134 \equiv 2$, $23 \equiv 2$, $49 \equiv 1$, $235 \equiv 1$ and $13 \equiv 1$, therefore: $123 \cdot 234 \cdot 32 \cdot 56 \cdot 22 \cdot 12 \cdot 78 \equiv 1 \cdot 2 \cdot 2 \cdot 1 \cdot 1 \cdot 1 \equiv 4 \equiv 1 \pmod{3}$. Leaving a remainder of 1.

```

1 void modArithmetic (int a, int b, int x) {
2     // If the result of adding a and b is greater than x, take the
3     // remainder of the division by x
4     (a + b) % x;
5     // If the result of subtracting a and b is less than 0, add x to the
6     // result and take the modulus again
7     (a - b %x + x) % x;
8     // If the result of multiplying a and b is greater than x, take the
9     // remainder of the division by x
10    (a * b) % x;
11 }

```

4.3 Modular Inverse

The modular inverse of an integer a modulo m is an integer x such that $ax \equiv 1 \pmod{m}$.

- If a and N are integers such that $\gcd(a, N) = 1$, then there exists an integer x such that $ax \equiv 1 \pmod{N}$.
 x is called the modular inverse of a modulo N .

However, $\frac{a}{b} \pmod{N}$ is not the same as $(\frac{a \bmod N}{b \bmod N}) \pmod{N}$.

- Lets take $a = 10$, $b = 2$, and $N = 3$.
 $\frac{10}{2} \pmod{3} = 5 \pmod{3} = 2$; $(\frac{10 \bmod 3}{2 \bmod 3}) \pmod{3} = (\frac{1}{2}) \pmod{3} = 0.5$.
This discrepancy is due to the fact that division is not always compatible with modular arithmetic.

On the other hand, using the extended Euclidean algorithm, we can find the modular inverse of a modulo N :

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int gcdExtended(int a, int b, int& x, int& y) {
5     // Base Case
6     if (b == 0) {
7         x = 1;
8         y = 0;
9         return a;
10    }
11
12    int x1, y1;

```



```

13     int gcd = gcdExtended(b%a, a, &x1, &y1);
14
15     x = y1;
16     y = x1 - y1 * (a / b);
17
18     return gcd;
19 }

```

4.4 Modular Binomial Coefficient and Permutations

4.5 Non-Mod Binomial Coefficient and Permutations

4.6 Modular Catalan Numbers

4.7 Fractional Ceiling

```

1 long long int ceil(long long int numerator, long long int denominator) {
2     return (numerator + denominator - 1) / denominator;
3 }

```

4.8 Fibonacci Numbers

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int fibonacci(int x) {
5     if (x == 0) return 0;
6     if (x == 1) return 1;
7     return fibonacci(x - 1) + fibonacci(x - 2);
8 }

```

4.9 Sieve Of Eratosthenes

```

1 #include <bits/stdc++.h>
2 #define MAX 1000001
3 using namespace std;
4
5 // Define both prime and pfix arrays
6 bool prime[MAX];
7 int pfix[MAX];
8
9 void sieve() {
10     // Set all numbers as prime
11     memset(prime, true, sizeof(prime));

```

```

12     // 0 and 1 are not prime
13     prime[0] = prime[1] = false;
14     // Iterate over all numbers
15     for (int p = 2; p * p < MAX; p++)
16         if (prime[p]) for (int i = p * p; i < MAX; i += p) prime[i] =
17             false;
18     // Calculate prefix sum of prime numbers
19     for (int i = 2; i < MAX; i++) {pfix[i] = pfix[i - 1] + prime[i];}

```

4.10 Sieve-based Factorization

```

1 #include <bits/stdc++.h>
2 #define MAX 1000001
3 using namespace std;
4
5 void sieveFactorization() {
6     // smallest_prime[i] stores the smallest prime factor of i
7     int smallest_prime[MAX];
8
9     // Initialize the samllest prime factor of each number
10    for (int i = 2; i < MAX; i++)
11        // If i is prime, then the smallest prime factor of i is i,
12        // otherwise is the smallest prime factor of i
13        smallest_prime[i] = (i % 2 == 0 ? 2 : i);
14
15    // Iterate over all odd numbers
16    for (int i = 3; i * i < MAX; i += 2)
17        if (smallest_prime[i] == i)
18            // Marks the smallest prime factor of all multiples of i as
19            // i, but only if it is the smallest prime factor
20            for (int j = i * i; j < MAX; j += i)
21                smallest_prime[j] = min(smallest_prime[j],
22                    smallest_prime[i]);

```

4.11 Cycle Finding

4.12 Berlekamp Massey

4.13 Modular Berlekamp Massey

4.14 Matrix exponentiation

4.15 Ecuaciones Diofantinas

4.16 Pollard-Rho, Stolen from GGDem

4.17 FFT, Stolen from GGDem

4.18 Euler Totient Function

5 Geometry

6 Strings

6.1 Explode by token

```

1 vector<string> explode_by_token(string const& s, char delimiter) {
2     vector<string> result;
3     // Create a string stream from the string, allowing to perform input
4     // output operations on strings.
5     stringstream iss(s);
6     // Read the string stream, tokenizing it by the delimiter
7     for(string token; getline(iss, token, delimiter);) {
8         // Split the string by the delimiter and push it to the result
9         // vector
10        result.push_back(move(token));
11    }
12    // Return the result vector
13    return result;
14 }
```

6.2 Multiple Hashings DS

6.3 Permute chars of string

6.4 Longest common subsequence

6.5 KMP

6.6 Suffix Array

6.7 STL Suffix Array

7 Classics

7.1 Job scheduling

7.1.1 One machine, linear penalty

7.1.2 One machine, deadlines

7.1.3 One machine, profit

7.1.4 Two machines, min time

8 Flow

8.1 Dinic

```

1 #define pb push_back
2 #define mp make_pair
3 #define fst first
4 #define snd second
5 #define ALL(s) s.begin(),s.end()
6 #define SZ(x) int((x).size())
7 #define fore(i, a, b) for(int i = a, to = b; i < to; ++i)
8 using namespace std;
9 typedef long long ll;
10
11 #define INF (1LL<<62)
12 // Min cut: nodes with dist >= 0 vs nodes with dist<0
13 // Matching MVC: left nodes with dist < 0 + right nodes with dist > 0
14
15 struct Dinic{
16     int nodes, src, dst;
17     vector<int> dist, q, work;
18     struct edge {int to, rev; ll f, cap;};
```

```

19 vector<vector<edge>> g;
20 Dinic(int x):nodes(x), g(x), dist(x), q(x), work(x){}
21 void add_edge(int s, int t, ll cap){
22     g[s].pb((edge){t, SZ(g[t]), 0, cap});
23     g[t].pb((edge){s, SZ(g[s]) - 1, 0, 0});
24 }
25 bool dinic_bfs(){
26     fill(ALL(dist), -1); dist[src] = 0;
27     int qt = 0; q[qt++] = src;
28     for(int qh = 0; qh < qt; qh++){
29         int u = q[qh];
30         fore(i, 0, SZ(g[u])){
31             edge &e = g[u][i]; int v = g[u][i].to;
32             if(dist[v] < 0 && e.f < e.cap) dist[v] = dist[u] + 1, q[qt++] =
33                 v;
34         }
35     }
36     return dist[dst] >= 0;
37 }
38 ll dinic_dfs(int u, ll f){
39     if(u == dst) return f;
40     for(int &i = work[u]; i < SZ(g[u]); i++){
41         edge &e = g[u][i];
42         if(e.cap <= e.f) continue;
43         int v = e.to;
44         if(dist[v] == dist[u] + 1){
45             ll df = dinic_dfs(v, min(f, e.cap - e.f));
46             if(df > 0){e.f += df; g[v][e.rev].f -= df; return df;}
47         }
48     }
49     return 0;
50 }
51 ll max_flow(int _src, int _dst){
52     src = _src; dst = _dst;
53     ll result = 0;
54     while(dinic_bfs()){
55         fill(ALL(work), 0);
56         while(ll delta = dinic_dfs(src, INF)) result += delta;
57     }
58     return result;
59 };
60

```

```

61 int main() {
62     ios_base::sync_with_stdio(false); cout.tie(NULL); cin.tie(NULL);
63     //l set, r set
64     int n, m;
65     cin >> n >> m;
66     m += n;
67     Dinic d(n + m + 2);
68     for(int i = 1; i <= n; i++) d.add_edge(0, i, 1);
69     for(int i = n + 1; i <= m; i++) d.add_edge(i, m + 1, 1);
70
71     int fin, q;
72     for(int i = 1; i <= n; i++){
73         cin >> q;
74         while(q--){
75             cin >> fin;
76             d.add_edge(i, n + fin, 1);
77         }
78     }
79     int res = d.max_flow(0, m + 1);
80     m -= n;
81     cout << m - res << endl;
82 }

```

9 Miscellaneous

9.1 Policy Based Data Structures

```

1 #include "bits/stdc++.h"
2 #include <bits/extc++.h>
3 using namespace __gnu_pbds;
4 using namespace std;
5 // Defines a new type which is an Order Statistic Tree where each node
6 // stores a pair of integers, ordered by the pair in ascending order.
7 typedef tree<pair<int,int>, null_type, less<pair<int,int>>, rb_tree_tag,
8     tree_order_statistics_node_update> ost;
9 using namespace std;
10
11 int main(){
12     // Creates an instance of OST named tree
13     ost tree;
14     // Inserts 5 elements, each one with a different id
15     int n = 5;
16     for(int id = 1; id <= n; id++)

```

```

15 // Inserts a pair with the value and the id
16 for(int val = 0; val < n; val++)
17     tree.insert({val,id});
18 // Returns the smallest value, in case of a tie it returns the
19 // smallest id
20 cout << (*tree.find_by_order(0)).first << " " << (*tree.
21 // find_by_order(0)).second << endl;
22 // Returns the index (0 indexed) of the first occurrence of .first
23 cout << tree.order_of_key({1,-1}) << endl;
24 }

```

9.2 Bit Manipulation

```

1 #include "bits/stdc++.h"
2 using namespace std;
3
4 // Bitmasks are represented from 30 to 62 bits using signed int and
5 // signed long long int to avoid problems with two's complement
6 int main() {
7     signed int a, b;
8
9     // To multiply a number by two, just apply a left shift
10    a = 1;
11    a = a << 3;
12
13    // To divide a number by two, just apply a right shift
14    a = 32;
15    a = a >> 3;
16
17    // To turn on the n-th bit of a number, just apply a bitwise OR with
18    // 2^(n-1), turns on the third bit
19
20    a = 1;
21    b = 1 << 2;
22    a = a | b;
23
24    // To turn off the n-th bit of a number, just apply a bitwise AND with
25    // the complement of ~2^(n-1), turns off the third bit
26
27    a = 5;
28    b = 1 << 2;
29    a &= ~b;
30
31    // To check if the n-th bit of a number is on, just apply a bitwise
32    // AND with 2^(n-1) and check if the result is turned on

```

```

27 a = 5;
28 b = 1 << 2;
29 a = a & b;
30 cout << (a ? "YES" : "NO") << endl;
31
32 // To reverse the n-th bit of a number, just power the n-th bit with
33 // 2^(n-1)
34 a = 5;
35 b = 1 << 2;
36 a = a ^ b;
37
38 // To obtain the least significant bit of a number that is turned on
39 // , just apply a bitwise AND with the complement of the number and
40 // add one
41 a = 12;
42 log2(a & ((-1) * a)) + 1
43
44 // To turn on all bits of a number
45 a = (1<<4)-1;
46 }

```

9.2.1 Bitmasking

Bitmasking is a technique used in computer programming to solve problems using individual or groups of bits within a binary number.

It is a powerful tool to solve problems that involve subsets, permutations, and combinations; using bitwise operations such as AND, OR, XOR and NOT.

Some bitmasking utilities are:

- **Memory efficiency:** Bitmasks can be used for compact and memory efficient storage big collections of data.
- **Subset, permutation and combination generation:** Can be used to generate all possible subsets, permutations and combinations of a set.
- **Set operations:** Can be used to perform set operations such as union, intersection, and difference.
- **Data masking and filtering:** By selectively turning on or off bits, we can filter out or mask certain data.
- **Optimization:** Algorithm optimization can be achieved using bitmasking, substituting bit-level operations for arithmetic operations.

10 Testing

10.1 Gen and AutoRun testcases

10.1.1 Gen.cpp

10.1.2 Stress testing

10.1.3 Autorun

10.2 Highly Composite Numbers

Particularly useful when testing number theoretical solutions.

1	1	1	
2	2	2	2
3	4	3	2^2
4	6	4	$2*3$
5	12	6	2^2*3
6	24	8	2^3*3
7	36	9	2^2*3^2
8	48	10	2^4*3
9	60	12	2^2*3*5
10	120	16	2^3*3*5
11	180	18	2^2*3^2*5
12	240	20	2^4*3*5
13	360	24	2^3*3^2*5
14	720	30	2^4*3^2*5
15	840	32	$2^3*3*5*7$
16	1260	36	2^2*3^2*5*7
17	1680	40	$2^4*3*5*7$
18	2520	48	2^3*3^2*5*7
19	5040	60	2^4*3^2*5*7
20	7560	64	2^3*3^3*5*7
21	10080	72	2^5*3^2*5*7
22	15120	80	2^4*3^3*5*7
23	20160	84	2^6*3^2*5*7
24	25200	90	$2^4*3^2*5^2*7$
25	27720	96	$2^3*3^3*5*7*11$
26	45360	100	2^4*3^4*5*7
27	50400	108	$2^5*3^2*5^2*7$
28	55440	120	$2^4*3^2*5*7*11$
29	83160	128	$2^3*3^3*5*7*11$
30	110880	144	$2^5*3^2*5*7*11$
31	166320	160	$2^4*3^3*5*7*11$

32	221760	168	$2^6*3^2*5*7*11$
33	277200	180	$2^4*3^2*5^2*7*11$
34	332640	192	$2^5*3^3*5*7*11$
35	498960	200	$2^4*3^4*5*7*11$
36	554400	216	$2^5*3^2*5^2*7*11$
37	665280	224	$2^6*3^3*5*7*11$
38	720720	240	$2^4*3^2*5*7*11*13$
39	1081080	256	$2^3*3^3*5*7*11*13$
40	1441440	288	$2^5*3^2*5*7*11*13$
41	2162160	320	$2^4*3^3*5*7*11*13$
42	2882880	336	$2^6*3^2*5*7*11*13$
43	3603600	360	$2^4*3^2*5^2*7*11*13$
44	4324320	384	$2^5*3^3*5*7*11*13$
45	6486480	400	$2^4*3^4*5*7*11*13$
46	7207200	432	$2^5*3^2*5^2*7*11*13$
47	8648640	448	$2^6*3^3*5*7*11*13$
48	10810800	480	$2^4*3^3*5^2*7*11*13$
49	14414400	504	$2^6*3^2*5^2*7*11*13$
50	17297280	512	$2^7*3^3*5*7*11*13$
51	21621600	576	$2^5*3^3*5^2*7*11*13$
52	32432400	600	$2^4*3^4*5^2*7*11*13$
53	36756720	640	$2^4*3^3*5*7*11*13*17$
54	43243200	672	$2^6*3^3*5^2*7*11*13$
55	61261200	720	$2^4*3^2*5^2*7*11*13*17$
56	73513440	768	$2^5*3^3*5*7*11*13*17$
57	110270160	800	$2^4*3^4*5*7*11*13*17$
58	122522400	864	$2^5*3^2*5^2*7*11*13*17$
59	147026880	896	$2^6*3^3*5*7*11*13*17$
60	183783600	960	$2^4*3^3*5^2*7*11*13*17$
61	245044800	1008	$2^6*3^2*5^2*7*11*13*17$
62	294053760	1024	$2^7*3^3*5*7*11*13*17$
63	367567200	1152	$2^5*3^3*5^2*7*11*13*17$
64	551350800	1200	$2^4*3^4*5^2*7*11*13*17$
65	698377680	1280	$2^4*3^3*5*7*11*13*17*19$
66	735134400	1344	$2^6*3^3*5^2*7*11*13*17$
67	1102701600	1440	$2^5*3^4*5^2*7*11*13*17$
68	1396755360	1536	$2^5*3^3*5*7*11*13*17*19$
69	2095133040	1600	$2^4*3^4*5*7*11*13*17*19$
70	2205403200	1680	$2^6*3^4*5^2*7*11*13*17$
71	2327925600	1728	$2^5*3^2*5^2*7*11*13*17*19$
72	2793510720	1792	$2^6*3^3*5*7*11*13*17*19$
73	3491888400	1920	$2^4*3^3*5^2*7*11*13*17*19$
74	4655851200	2016	$2^6*3^2*5^2*7*11*13*17*19$

75	5587021440	2048	$2^7 \cdot 3^3 \cdot 5^7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	118	130429015516800	18432	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$
76	6983776800	2304	$2^5 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	119	195643523275200	20160	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$
77	10475665200	2400	$2^4 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	120	260858031033600	20736	$2^8 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$
78	13967553600	2688	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	121	288807105787200	21504	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
79	20951330400	2880	$2^5 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	122	391287046550400	23040	$2^7 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$
80	27935107200	3072	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	123	577614211574400	24576	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
81	41902660800	3360	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	124	782574093100800	25920	$2^8 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$
82	48886437600	3456	$2^5 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	125	866421317361600	26880	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
83	64250746560	3584	$2^6 \cdot 3^3 \cdot 5^7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	126	1010824870255200	27648	$2^5 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
84	73329656400	3600	$2^4 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	127	1444035528936000	28672	$2^6 \cdot 3^3 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
85	80313433200	3840	$2^4 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	128	1516237305382800	28800	$2^4 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
86	97772875200	4032	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	129	1732842634723200	30720	$2^7 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
87	128501493120	4096	$2^7 \cdot 3^3 \cdot 5^7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	130	2021649740510400	32256	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
88	146659312800	4320	$2^5 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	131	2888071057872000	32768	$2^7 \cdot 3^3 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
89	160626866400	4608	$2^5 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	132	3032474610765600	34560	$2^5 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
90	240940299600	4800	$2^4 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	133	4043299481020800	36864	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
91	293318625600	5040	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$	134	6064949221531200	40320	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
92	321253732800	5376	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	135	8086598962041600	41472	$2^8 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
93	481880599200	5760	$2^5 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	136	10108248702552000	43008	$2^6 \cdot 3^3 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
94	642507465600	6144	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	137	12129898443062400	46080	$2^7 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
95	963761198400	6720	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	138	18194847664593600	48384	$2^6 \cdot 3^5 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
96	1124388064800	6912	$2^5 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	139	20216497405104000	49152	$2^7 \cdot 3^3 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
97	1606268664000	7168	$2^6 \cdot 3^3 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	140	24259796886124800	51840	$2^8 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
98	1686582097200	7200	$2^4 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	141	30324746107656000	53760	$2^6 \cdot 3^4 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
99	1927522396800	7680	$2^7 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	142	36389695329187200	55296	$2^7 \cdot 3^5 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
100	2248776129600	8064	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	143	48519593772249600	57600	$2^9 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
101	3212537328000	8192	$2^7 \cdot 3^3 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	144	60649492215312000	61440	$2^7 \cdot 3^4 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
102	3373164194400	8640	$2^5 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	145	72779390658374400	62208	$2^8 \cdot 3^5 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31$
103	4497552259200	9216	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	146	74801040398884800	64512	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
104	6746328388800	10080	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	147	106858629141264000	65536	$2^7 \cdot 3^3 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
105	8995104518400	10368	$2^8 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	148	112201560598327200	69120	$2^5 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
106	9316358251200	10752	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$	149	149602080797769600	73728	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
107	13492656777600	11520	$2^7 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	150	224403121196654400	80640	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
108	18632716502400	12288	$2^7 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$	151	299204161595539200	82944	$2^8 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
109	26985313555200	12960	$2^8 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	152	374005201994424000	86016	$2^6 \cdot 3^3 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
110	27949074753600	13440	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$	153	448806242393308800	92160	$2^7 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
111	32607253879200	13824	$2^5 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$	154	673209363589963200	96768	$2^6 \cdot 3^5 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
112	46581791256000	14336	$2^6 \cdot 3^3 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$	155	748010403988848000	98304	$2^7 \cdot 3^3 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
113	48910880818800	14400	$2^4 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$	156	897612484786617600	103680	$2^8 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
114	55898149507200	15360	$2^7 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$				
115	65214507758400	16128	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$				
116	93163582512000	16384	$2^7 \cdot 3^3 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$				
117	97821761637600	17280	$2^5 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$				