Trimmed `https://cp-algorithms.com/` for NSSPC Usage

Contents

1. Algebra

1.1. Euclidean algorithm for computing the greatest common divisor

$$\gcd(a, b) = \begin{cases} a, & \text{if } b{=}0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

```
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

Note that since C++17, `gcd` is implemented as a standard function in C++.

1.2. n^{th} Fibonacci number using Fast Doubling Method

```
pair<int, int> fib (int n) {
    if (n == 0)
        return {0, 1};

    auto p = fib(n >> 1);
    int c = p.first * (2 * p.second - p.first);
    int d = p.first * p.first + p.second * p.second;
    if (n & 1)
        return {d, c + d};
    else
        return {c, d};
}
```

The above code returns F_n and F_{n+1} as a pair.

1.3. Sieve of Eratosthenes (finding prime numbers in a segment $[1; n]$)

1.3.1. Regular sieve

```
int n;
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i <= n; i++) {
    if (is_prime[i] && (long long)i * i <= n) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

Time complexity: $O(n \log \log n)$

Memory complexity: $O(n)$

1.3.2. Segmented sieve (counting primes)

```
int count_primes(int n) {
    const int S = 10000;

    vector<int> primes;
    int nsqrt = sqrt(n);
    vector<char> is_prime(nsqrt + 2, true);
    for (int i = 2; i <= nsqrt; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            for (int j = i * i; j <= nsqrt; j += i)
                is_prime[j] = false;
        }
    }

    int result = 0;
    vector<char> block(S);
    for (int k = 0; k * S <= n; k++) {
        fill(block.begin(), block.end(), true);
        int start = k * S;
        for (int p : primes) {
            int start_idx = (start + p - 1) / p;
            int j = max(start_idx, p) * p - start;
            for (; j < S; j += p)
```

```
                block[j] = false;
        }
        if (k == 0)
            block[0] = block[1] = false;
        for (int i = 0; i < S && start + i <= n; i++) {
            if (block[i])
                result++;
        }
    }
    return result;
}
```

Time complexity: $O(n \log \log n)$

Memory complexity: $O(\sqrt{n} + S)$

1.3.3. Finding primes in range

```
vector<char> segmentedSieve(long long L, long long R) {
    // generate all primes up to sqrt(R)
    long long lim = sqrt(R);
    vector<char> mark(lim + 1, false);
    vector<long long> primes;
    for (long long i = 2; i <= lim; ++i) {
        if (!mark[i]) {
            primes.emplace_back(i);
            for (long long j = i * i; j <= lim; j += i)
                mark[j] = true;
        }
    }

    vector<char> isPrime(R - L + 1, true);
    for (long long i : primes)
        for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
            isPrime[j - L] = false;
    if (L == 1)
        isPrime[0] = false;
    return isPrime;
}
```

Time complexity: $O\Big((R-L+1)\log\log(R) + \sqrt{R}\log\log\sqrt{R}\Big)$

1.4. Primality test

```
bool isPrime(int x) {
    for (int d = 2; d * d <= x; d++) {
        if (x % d == 0)
            return false;
    }
    return x >= 2;
}
```

This is the simplest form of a prime check. You can optimize this function quite a bit, for instance by only checking all odd numbers in the loop, since the only even prime number is 2.

1.5. Integer factorization

1.5.1. Trial division

1.5.1.1. Unoptimized trial division

If we cannot find any divisor in the range $[2; \sqrt{n}]$, then the number itself has to be prime.

```
vector<long long> trial_division1(long long n) {
    vector<long long> factorization;
    for (long long d = 2; d * d <= n; d++) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

1.5.1.2. Wheel factorization

Once we know that the number is not divisible by 2, we don't need to check other even numbers. This leaves us with only 50% of the numbers to check. After factoring out 2, and getting an odd number, we can simply start with 3 and only count other odd numbers.

```
vector<long long> trial_division2(long long n) {
    vector<long long> factorization;
    while (n % 2 == 0) {
        factorization.push_back(2);
        n /= 2;
    }
    for (long long d = 3; d * d <= n; d += 2) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

If the number is not divisible by 3, we can also ignore all other multiples of 3 in the future computations. So we only need to check the numbers $5, 7, 11, 13, 17, 19, 23, \ldots$. We can observe a pattern of these remaining numbers. We need to check all numbers with $d \bmod 6 = 1$ and $d \bmod 6 = 5$. So this leaves us with only 33.3% percent of the numbers to check. We can implement this by factoring out the primes 2 and 3 first, after which we start with 5 and only count remainders 1 and 5 modulo 6.

Here is an implementation for the prime number 2, 3 and 5:

```
vector<long long> trial_division3(long long n) {
    vector<long long> factorization;
    for (int d : {2, 3, 5}) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
```

```
        }
    }
    static array<int, 8> increments = {4, 2, 4, 2, 4, 6, 2, 6};
    int i = 0;
    for (long long d = 7; d * d <= n; d += increments[i++]) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
        if (i == 8)
            i = 0;
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

1.5.1.3. Precomputed primes

```
vector<long long> primes;

vector<long long> trial_division4(long long n) {
    vector<long long> factorization;
    for (long long d : primes) {
        if (d * d > n)
            break;
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

1.5.2. Pollard's $p-1$ method

```
long long pollards_p_minus_1(long long n) {
    int B = 10;
    long long g = 1;
    while (B <= 1000000 && g < n) {
        long long a = 2 + rand() %  (n - 3);
        g = gcd(a, n);
        if (g > 1)
            return g;

        // compute a^M
        for (int p : primes) {
            if (p >= B)
                continue;
            long long p_power = 1;
            while (p_power * p <= B)
                p_power *= p;
            a = power(a, p_power, n);

            g = gcd(a - 1, n);
            if (g > 1 && g < n)
```

```
                return g;
            }
            B *= 2;
        }
        return 1;
}
```

Time complexity: $O(B \log B \log^2 n)$ per iteration

1.5.3. Pollard's rho algorithm

1.5.3.1. Floyd's cycle-finding algorithm

```
long long mult(long long a, long long b, long long mod) {
    return (__int128)a * b % mod;
}

long long f(long long x, long long c, long long mod) {
    return (mult(x, x, mod) + c) % mod;
}

long long rho(long long n, long long x0=2, long long c=1) {
    long long x = x0;
    long long y = x0;
    long long g = 1;
    while (g == 1) {
        x = f(x, c, n);
        y = f(y, c, n);
        y = f(y, c, n);
        g = gcd(abs(x - y), n);
    }
    return g;
}
```

If GCC is not available, you can using a similar idea as binary exponentiation:

```
long long mult(long long a, long long b, long long mod) {
    long long result = 0;
    while (b) {
        if (b & 1)
            result = (result + a) % mod;
        a = (a + a) % mod;
        b >>= 1;
    }
    return result;
}
```

1.5.3.2. Brent's algorithm

```
long long brent(long long n, long long x0=2, long long c=1) {
    long long x = x0;
    long long g = 1;
    long long q = 1;
    long long xs, y;

    int m = 128;
    int l = 1;
    while (g == 1) {
        y = x;
        for (int i = 1; i < l; i++)
```

```
            x = f(x, c, n);
        int k = 0;
        while (k < l && g == 1) {
            xs = x;
            for (int i = 0; i < m && i < l - k; i++) {
                x = f(x, c, n);
                q = mult(q, abs(y - x), n);
            }
            g = gcd(q, n);
            k += m;
        }
        l *= 2;
    }
    if (g == n) {
        do {
            xs = f(xs, c, n);
            g = gcd(abs(xs - y), n);
        } while (g == 1);
    }
    return g;
}
```

1.6. Number/Sum of divisors

1.6.1. Number of divisors

```
long long numberOfDivisors(long long num) {
    long long total = 1;
    for (int i = 2; (long long)i * i <= num; i++) {
        if (num % i == 0) {
            int e = 0;
            do {
                e++;
                num /= i;
            } while (num % i == 0);
            total *= e + 1;
        }
    }
    if (num > 1) {
        total *= 2;
    }
    return total;
}
```

1.6.2. Sum of divisors

```
long long SumOfDivisors(long long num) {
    long long total = 1;

    for (int i = 2; (long long)i * i <= num; i++) {
        if (num % i == 0) {
            int e = 0;
            do {
                e++;
                num /= i;
            } while (num % i == 0);

            long long sum = 0, pow = 1;
```

```
            do {
                sum += pow;
                pow *= i;
            } while (e-- > 0);
            total *= sum;
        }
    }
    if (num > 1) {
        total *= (1 + num);
    }
    return total;
}
```

1.7. Gray code

1.7.1. Finding gray code

```
int g (int n) {
    return n ^ (n >> 1);
}
```

1.7.2. Finding inverse gray code

```
int rev_g (int g) {
  int n = 0;
  for (; g; g >>= 1)
    n ^= g;
  return n;
}
```

2. String Processing

2.1. String hashing

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

2.2. Task - Finding repetitions

Main-Lorentz algorithm:

```
vector<int> z_function(string const& s) {
    int n = s.size();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; i++) {
        if (i <= r)
            z[i] = min(r-i+1, z[i-l]);
        while (i + z[i] < n && s[z[i]] == s[i+z[i]])
            z[i]++;
        if (i + z[i] - 1 > r) {
            l = i;
```

```
            r = i + z[i] - 1;
        }
    }
    return z;
}

int get_z(vector<int> const& z, int i) {
    if (0 <= i && i < (int)z.size())
        return z[i];
    else
        return 0;
}

vector<pair<int, int>> repetitions;

void convert_to_repetitions(int shift, bool left, int cntr, int l, int k1, int k2) {
    for (int l1 = max(1, l - k2); l1 <= min(l, k1); l1++) {
        if (left && l1 == l) break;
        int l2 = l - l1;
        int pos = shift + (left ? cntr - l1 : cntr - l - l1 + 1);
        repetitions.emplace_back(pos, pos + 2*l - 1);
    }
}

void find_repetitions(string s, int shift = 0) {
    int n = s.size();
    if (n == 1)
        return;

    int nu = n / 2;
    int nv = n - nu;
    string u = s.substr(0, nu);
    string v = s.substr(nu);
    string ru(u.rbegin(), u.rend());
    string rv(v.rbegin(), v.rend());

    find_repetitions(u, shift);
    find_repetitions(v, shift + nu);

    vector<int> z1 = z_function(ru);
    vector<int> z2 = z_function(v + '#' + u);
    vector<int> z3 = z_function(ru + '#' + rv);
    vector<int> z4 = z_function(v);

    for (int cntr = 0; cntr < n; cntr++) {
        int l, k1, k2;
        if (cntr < nu) {
            l = nu - cntr;
            k1 = get_z(z1, nu - cntr);
            k2 = get_z(z2, nv + 1 + cntr);
        } else {
            l = cntr - nu + 1;
            k1 = get_z(z3, nu + 1 + nv - 1 - (cntr - nu));
            k2 = get_z(z4, (cntr - nu) + 1);
        }
        if (k1 + k2 >= l)
```

```
            convert_to_repetitions(shift, cntr < nu, cntr, l, k1, k2);
    }
}
```

3. Combinatorics

3.1. Finding Power of Factorial Divisor

You are given two numbers n and k. Find the largest power of k x such that $n!$ is divisible by k^x.

```
int fact_pow (int n, int k) {
    int res = 0;
    while (n) {
        n /= k;
        res += n;
    }
    return res;
}
```

4. Graphs

4.1. Breadth first search

The algorithm takes as input an unweighted graph and the id of the source vertex s. The input graph can be directed or undirected, it does not matter to the algorithm.

The algorithm can be understood as a fire spreading on the graph: at the zeroth step only the source $s$is on fire. At each step, the fire burning at each vertex spreads to all of its neighbors. In one iteration of the algorithm, the "ring of fire" is expanded in width by one unit (hence the name of the algorithm).

More precisely, the algorithm can be stated as follows: Create a queue q which will contain the vertices to be processed and a Boolean array used[] which indicates for each vertex, if it has been lit (or visited) or not.

Initially, push the source s to the queue and setused$[s] = \text{true}$, and for all other vertices$v$set used$[v] = \text{false}$. Then, loop until the queue is empty and in each iteration, pop a vertex from the front of the queue. Iterate through all the edges going out of this vertex and if some of these edges go to vertices that are not already lit, set them on fire and place them in the queue.

As a result, when the queue is empty, the "ring of fire" contains all vertices reachable from the source s, with each vertex reached in the shortest possible way. You can also calculate the lengths of the shortest paths (which just requires maintaining an array of path lengths d[]) as well as save information to restore all of these shortest paths (for this, it is necessary to maintain an array of "parents"p[], which stores for each vertex the vertex from which we reached it).

```
vector<vector<int>> adj;  // adjacency list representation
int n; // number of nodes
int s; // source vertex

queue<int> q;
vector<bool> used(n);
vector<int> d(n), p(n);

q.push(s);
used[s] = true;
p[s] = -1;
```

```
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

If we have to restore and display the shortest path from the source to some vertex u, it can be done in the following manner:

```
if (!used[u]) {
    cout << "No path!";
} else {
    vector<int> path;
    for (int v = u; v != -1; v = p[v])
        path.push_back(v);
    reverse(path.begin(), path.end());
    cout << "Path: ";
    for (int v : path)
        cout << v << " ";
}
```

4.2. Depth first search

The idea behind DFS is to go as deep into the graph as possible, and backtrack once you are at a vertex without any unvisited adjacent vertices.

It is very easy to describe / implement the algorithm recursively: We start the search at one vertex. After visiting a vertex, we further perform a DFS for each adjacent vertex that we haven't visited before. This way we visit all vertices that are reachable from the starting vertex.

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<bool> visited;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
}
```

4.3. Finding connected components

```
int n;
vector<vector<int>> adj;
vector<bool> used;
vector<int> comp;

void dfs(int v) {
    used[v] = true ;
```

```
    comp.push_back(v);
    for (int u : adj[v]) {
        if (!used[u])
            dfs(u);
    }
}

void find_comps() {
    fill(used.begin(), used.end(), 0);
    for (int v = 0; v < n; ++v) {
        if (!used[v]) {
            comp.clear();
            dfs(v);
            cout << "Component:" ;
            for (int u : comp)
                cout << ' ' << u;
            cout << endl ;
        }
    }
}
```

4.4. Dijkstra Algorithm

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

Here the graph adjis stored as adjacency list: for each vertex v

adj$[v]$contains the list of edges going from this vertex, i.e. the list of `pair<int,int>` where the first element in the pair is the vertex at the other end of the edge, and the second element is the edge weight.

The function takes the starting vertex s and two vectors that will be used as return values.

First of all, the code initializes arrays: distances $d[]$, labels $u[]$ and predecessors $p[]$. Then it performs n iterations. At each iteration the vertex v is selected which has the smallest distance$d[v]$among all the unmarked vertices. If the distance to selected vertex v is equal to infinity, the algorithm stops. Otherwise the vertex is marked, and all the edges going out from this vertex are checked. If relaxation along the edge is possible (i.e. distance $d[\text{to}]$ can be improved), the distance$d[\text{to}]$ and predecessor $p[\text{to}]$are updated.

After performing all the iterations array $d[]$ stores the lengths of the shortest paths to all vertices, and array $p[]$stores the predecessors of all vertices (except starting vertex s). The path to any vertex t can be restored in the following way:

```
vector<int> restore_path(int s, int t, vector<int> const& p) {
    vector<int> path;

    for (int v = t; v != s; v = p[v])
        path.push_back(v);
    path.push_back(s);

    reverse(path.begin(), path.end());
    return path;
}
```

5. Miscellaneous

5.1. Longest increasing subsequence

We are given an array with n numbers: $a[0...n-1]$. The task is to find the longest, strictly increasing, subsequence in a.

Formally we look for the longest sequence of indices $i_1, ...i_k$such that

i_1 < i_2 < dots < i_k,quad a[i_1] < a[i_2] < dots < a[i_k]

5.1.1. Solution in $O(n^2)$ with dynamic programming

5.1.1.1. Finding the length

```
int lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i])
                d[i] = max(d[i], d[j] + 1);
        }
    }

    int ans = d[0];
    for (int i = 1; i < n; i++) {
        ans = max(ans, d[i]);
    }
```

```
    return ans;
}
```

5.1.1.2. Restoring the subsequence

```
vector<int> lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1), p(n, -1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i] && d[i] < d[j] + 1) {
                d[i] = d[j] + 1;
                p[i] = j;
            }
        }
    }

    int ans = d[0], pos = 0;
    for (int i = 1; i < n; i++) {
        if (d[i] > ans) {
            ans = d[i];
            pos = i;
        }
    }

    vector<int> subseq;
    while (pos != -1) {
        subseq.push_back(a[pos]);
        pos = p[pos];
    }
    reverse(subseq.begin(), subseq.end());
    return subseq;
}
```

5.1.2. Solution in $O(n \log n)$ with dynamic programming and binary search

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[l-1] < a[i] && a[i] < d[l])
            d[l] = a[i];
    }

    int ans = 0;
    for (int l = 0; l <= n; l++) {
        if (d[l] < INF)
            ans = l;
    }
    return ans;
}
```

It is also possible to restore the subsequence using this approach. This time we have to maintain two auxiliary arrays. One that tells us the index of the elements in $d[]$. And again we have to create an

array of "ancestors" $p[i]$. $p[i]$ will be the index of the previous element for the optimal subsequence ending in element i.

It's easy to maintain these two arrays in the course of iteration over the array $a[]$ alongside the computations of $d[]$. And at the end it is not difficult to restore the desired subsequence using these arrays.
