Bogosort

Scripts for competitive programming

```
// Binary search
const int RESULT = 100;

int bsearch() {
    int low = 0;
    int high = 1e9;
    while (low < high) {
        int mid = (low + high) / 2;
        if (mid == RESULT) low = high = mid;
        else if (mid < RESULT) low = mid;
        else high = mid;
    }
    return low;
}
```

```
// BFS
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> adjacent;

int bfs(int start, int end) {
    vector<bool> visited(adjacent.size());
    queue<pair<int, int>> q;
    q.push({start, 0});

    while(!q.empty()) {
        auto p = q.front();
        q.pop();
        int vertex = p.first;
        int distance = p.second;

        if (vertex == end) return distance;
        if (visited[vertex]) continue;
        visited[vertex] = true;
        for (int v : adjacent[vertex]) {
            q.push({v, distance + 1});
        }
    }
}
```

```cpp
// DFS
// Suited for trees. If you want to use this with general undirected graphs,
// use a `visited` vector

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

vector<vector<int>> adjacent;
vector<bool> visited;

bool dfs(int vertex, int parent, int goal) {
    if (vertex == goal) return true;

    bool found = false;
    for (auto v : adjacent[vertex]) {
        if (v == parent) continue;
        if (!dfs(v, vertex, goal)) continue;
        found = true;
        break;
    }
}
```

```cpp
// Segment tree
#include <bits/stdc++.h>
using namespace std;

// Maximum segment tree
struct STree {
    int n;
    vector<int> elements;

    STree(int size) {
        n = best2power(size);
        elements = vector<int>(n * 2);
    }

    void update(int k, int x) {
        k += n;
        elements[k] = x;
        for (k /= 2; k >= 1; k /= 2) {
            int a = k * 2;
            int b = a + 1;
            elements[k] = max(elements[a], elements[b]);
        }
    }

    int get(int k) {
        return elements[k + n];
    }

    int max_element(int l, int r) {
        l += n;
        r += n;
        int result = -1;
        while (n <= r) {
            if (l & 1) {
                result = max(result, elements[l]);
                l++;
            }
            if ((r & 1) == 0) {
                result = max(result, elements[r]);
                r--;
            }
            l /= 2;
            r /= 2;
        }
        return result;
    }
```

```
    int best2power(int n) {
        int i = 1;
        while (i < n) i *= 2;
        return i;
    }
};
```

```
//
#include <bits/stdc++.h>
using namespace std;

struct UFDS {
    map<int, int> parent;
    map<int, int> size;

    void enter(int p) {
        if (parent.count(p)) return;
        parent[p] = p;
        size[p] = 1;
    }

    void connect(int u, int v) {
        if (size[u] > size[v]) swap(u, v);
        parent[u] = v;
        size[v] += size[u];
    }

    int get_size(int p) {
        return size[root(p)];
    }

    int root(int p) {
        if (parent[p] = p) return p;
        int tmp = root(parent[p]);
        parent[p] = tmp;
        return tmp;
    }
};
```

```
// PBDS
#include <ext/pb_ds/detail/standard_policies.hpp>
// Or
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>


typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> pbds;

pbds X;
X.insert(1);
X.insert(2);

cout << *X.find_by_order(1) << "\n";
cout << X.order_of_key(2) << "\n";
```

```
// Subset generation
void search(int k) {
    if (k == n) {
        // process subset
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

```
// Generating permutations
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(),permutation.end()));
```

```
// Prime factors
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

```
// Sieve of Eratosthenes
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        sieve[u] = x;
    }
}
```