2. Используя класс queue из STL решите следующую задачу.
(2736) Вывести простые числа среди чисел от 2 до N, используя следующий алгоритм:
Первоначально очередь все числа от 2 до N.

1. Взять первый элемент X из входной очереди и напечатать.
2. В выходную очередь поместить числа из очереди, которые не кратны X.
3. Поменять входную и выходную очередь (swap).
4. Пока очередь не пуста, то повторять действия с шага 2.

Ввод содержит одно целое число N ($2 \leq N \leq 100000$).

Абстрактный тип данных (АТД) — это математическая модель для типов данных, где тип данных определяется поведением (семантикой) с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.
Конкретные реализации АТД называются структурами данных. В C++ структуры данных реализуются как классы. В задаче 3 необходимо определить АТД, не нужно писать реализацию класса C++!

```
#include <iostream>
#include <queue>

using namespace std;
int main()
{
    int n;
    cin >> n;
    queue<int> q;
    queue<int> out;
    for (int i = 2; i <= n; i++)
    {
        q.push(i);
    }
    int temp, qNum;
    int cou = 0;
    while (q.size())
    {
        qNum = q.front();
        q.pop();
        cout << qNum << " ";
        int count = q.size();
        for (int i = 0; i < count; i++){
            int temp = q.front();
            q.pop();
            if (temp % qNum != 0){
                out.push(temp);
            }
        }
        q.swap(out);
    }
    return 0;
}
```

5. Напишите функцию для обратного (post-order) обхода бинарного дерева, заданного следующей структурой:
`struct node { int value; node *left, *right; };`
К каждому значению, хранящемуся в дереве, функция применяет функцию, указанную в качестве аргумента:

```
void post-order(node *n, void (*f)(int));
struct node { int value; node *left, *right; };
void post-order(node *n, void (*op)(int))
{
    if (n != nullptr)
    {
        post-order(n->left, op);
        post-order(n->right, op);
        op(n->value);
    }
}
```

14. Используя поиск в ширину, решите задачу.
(1716) Стартовав с числа 1, нужно получить некоторое заданное число N. На каждом шаге можно добавлять к текущему числу один из его делителей, чтобы получить новое число. Например, для первого шага у нас только один вариант: добавить 1 к 1 и получить 2. На втором шаге можно выбрать один из двух делителей и получить число 2+1=3 или 2+2=4. От числа 4 на третьем шаге можно перейти к числам 5, 6 или 8 в зависимости от выбранного делителя.
Напишите программу, определяющую минимальное количество шагов для получения заданного числа N (2≤N≤10^5).

```
#include <iostream>
#include <queue>

using namespace std;
int main()
{
   queue<int> q;
   int n; cin >> n;
   vector<int> v(n + 1, 0);
   q.push(1);

   auto step = [&](int x, int k){
      if (x <= n && v[x] == 0)
      {
         v[x] = v[k] + 1;
         q.push(x);
      }
   };

   while(!q.empty())
   {
      int k = q.front();
      q.pop();
      for (int d = 1; d*d <= k; d++)
      {
         if (k % d == 0)
         {
            if (k + d <= n)
            {
               step(k+d, k);
               step(k+k/d, k);
            }
         }
      }
   }
   cout << v[n];
   return 0;
}
```

8. Используя map из STL напишите решение следующей задачи с эффективностью $O(NlogN)$.
Дана последовательность из n целых чисел. Найти непрерывную под последовательность максимальной длины, в которой нет одинаковых элементов. Вывести длину и начальный индекс найденной под последовательности.

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

int main()
{
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    map<int, int> m;
    pair<int, int> ans = {1, 0};

    int i = 0, j = 0;
    while (j < n)
    {
        if (m[arr[j]] == 0)
        {
            m[arr[j++]]++;
            if (j - i > ans.first)
                ans = {j - i, i};
        }
        else
            m[arr[i++]]--;
    }
    cout << ans.first << ' ' << ans.second << '\n';
    return 0;
}
```

11. Определить АТД Разреженная матрица, обеспечивающий метод get(i,j) для получения элемента матрицы и set(i,j,v) для изменения (добавления) ненулевого элемента. В конструкторе задаются размеры матрицы.
Реализовать АТД через словарь по ключам map<pair<int,int>, double>. Определить эффективность операций + и * в зависимости от количества ненулевых элементов K.

```
#include <iostream>
#include <map>

using namespace std;
class SparseMatrix
{
    int n, m;
    map<pair<int, int>, double> matrix;
public:
    SparseMatrix(int n, int m) : n(n), m(m) {}
    double get(int i, int j)
    {
        if (i < 0 || i >= n || j < 0 || j >= m)
            throw runtime_error("Invalid index");

        if (matrix.count({i, j}))
            return matrix[{i, j}];
        return 0;
    }
    void set(int i, int j, int v)
    {
        if (i < 0 || i >= n || j < 0 || j >= m)
            throw runtime_error("Invalid index");
        if (v == 0)
            matrix.erase({i, j});
        else
            matrix[{i, j}] = v;
    }
};

int main()
{
    return 0;
}
```

12. Определить АТД Матрица, обеспечивающий метод [i,j] для доступа к элементам матрицы. В конструкторе задаются размеры матрицы. Реализовать матрицу через vector размером $N \cdot M$. Определить операцию +. Сравнить время сложения матриц размером 1000×1000, меняя порядок циклов (строки/столбцы и столбцы/строки) для уровня оптимизации O3. Результаты записать в таблицу, в которой будет указан порядок выполнения циклов и время выполнения в мкс.

```
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;
class Matrix
{
    vector<double> v;
    int n, m;
public:
    Matrix(int n, int m) : n(n), m(m)
    {
        v.resize(n * m);
    }
    double &operator[](pair<int, int> index)
    {
        auto [i, j] = index;
        if (i < 0 || i > n || j < 0 || j > m)
            throw runtime_error("Invalid Index");

        return v[i * n + j];
    }
    Matrix operator+(const Matrix &other)
    {
        if (n != other.n || m != other.m)
            throw runtime_error("Invalid Sizes");

        Matrix answer(n, m);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                answer[{i, j}] = v[i * n + j] + other.v[i * n + j];
        return answer;
    }
};
int main()
{
    Matrix m(1000, 1000), m2(1000, 1000);
    long long cnt = 0;
    for (int i = 0; i < 100; i++)
    {
        chrono::steady_clock::time_point begin =
chrono::steady_clock::now();
        Matrix ans = m + m2;
        chrono::steady_clock::time_point end =
chrono::steady_clock::now();
        cnt += chrono::duration_cast<std::chrono::microseconds>(end -
begin).count();
    }
    cout << cnt / 100 << " microseconds\n";
    return 0;
}
```

20. Сравните время сортировки с помощью sort, stable_sort, make_heap/sort_heap для вектора из 10^6 случайных чисел. Результаты оформить в виде таблицы.

```
#include <iostream>
#include <chrono>
#include <algorithm>
#include <vector>
#include <ctime>

using namespace std;
vector<int> generate_vector()
{
    vector<int> ans;
    for (int i = 0; i < 1e6; i++)
        ans.push_back(rand());
    return ans;
}

int test(vector<int> &arr)
{
    std::chrono::steady_clock::time_point begin, end;
    begin = std::chrono::steady_clock::now();

    sort(arr.begin(), arr.end());
    // stable_sort(arr.begin(), arr.end());
    // make_heap(arr.begin(), arr.end());
    // sort_heap(arr.begin(), arr.end());

    end = std::chrono::steady_clock::now();

    return chrono::duration_cast<std::chrono::microseconds>(end -
begin).count();
}

int main()
{
    int sum = 0;
    for (int i = 0; i < 10; i++)
    {
        vector<int> arr = generate_vector();
        int cnt = test(arr);
        sum += cnt;
    }
    cout << "Avarage: " << sum / 10 << '\n';

    return 0;
}
```

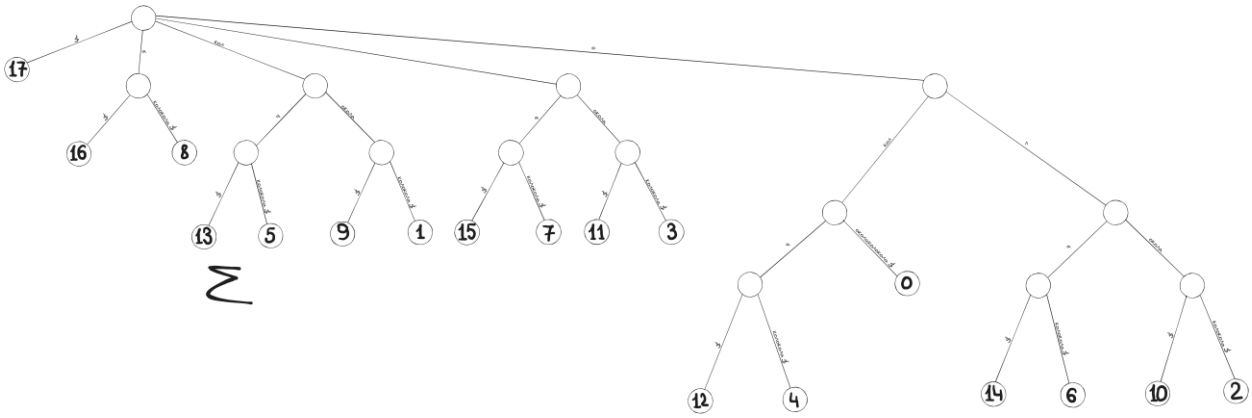
Результаты в микросекундах:

sort	stable_sort	make_heap/sort_heap
241206	278574	542811

1. Реализуйте АТД Стек на односвязном списке (forward_list).

```
template <typename T>
class Queue {
private:
    std::forward_list<T> q;
    typename std::forward_list<T>::iterator it;

public:
    Queue() : it(q.before_begin()) {}

    int size() const {
        return std::distance(q.begin(), q.end());
    }

    bool isEmpty() const {
        return size() == 0;
    }

    void push(const T& element) {
        q.insert_after(it, element);
        ++it;
    }

    T front() const {
        return q.front();
    }

    void pop() {
        q.pop_front();
    }
};
```

22. Постройте сжатое суффиксное дерево для строки "околоколаколокола" и найдите количество различных подстрок в этой строке. Объясните способ подсчета с использованием суффиксного дерева.
$-защитный символ.
Количество различных подстрок в этой строке = 153
Идём по каждому ребру и суммируем длину строки в этом ребре, когда доходим до листа, мы вычетаем накопленную длину строки.
Пример:
околоколаколокола – длина 17.
Идём налево и получаем всю длину строки = 17.
Идём вниз и налево. Накопленная длина 1 => 17-1 = 16.
Идём вниз и направо. Накопленная длина 1+8=9 => 17-9=8.
И т.д.



13. Используя поиск в глубину, определите число компонент связности в графе, задаваемом следующим образом:
Как матрица N×M из символов латинского алфавита. Клетки считаются связными, если в них находится одинаковая буква и они имеют общую границу.

```
#include <iostream>
#include <vector>

using namespace std;
// Проверка находится ли данная вершина внутри границ матрицы
bool isValid(int i, int j, int rows, int cols) {
    return (i >= 0 && i < rows && j >= 0 && j < cols);
}

// DFS для поиска компоненты связности
void dfs(vector<vector<char>>& matrix, int i, int j, char target,
vector<vector<bool>>& visited) {
    // Помечаем текущую клетку как посещенную
    visited[i][j] = true;

    // Проверяем соседние клетки
    static const int dr[] = {-1, 0, 1, 0};
    static const int dc[] = {0, -1, 0, 1};
    for (int k = 0; k < 4; ++k) {
        int ni = i + dr[k];
        int nj = j + dc[k];
        if (isValid(ni, nj, matrix.size(), matrix[0].size()) &&
matrix[ni][nj] == target && !visited[ni][nj]) {
            dfs(matrix, ni, nj, target, visited);
        }
    }
}

// Функция для подсчета числа компонент связности
int countConnectedComponents(vector<vector<char>>& matrix) {
    int rows = matrix.size();
    int cols = matrix[0].size();
    vector<vector<bool>> visited(rows, vector<bool>(cols, false)); //
Массив для отслеживания посещенных вершин
    int count = 0; // Счетчик компонент связности

    // Проходим по каждой клетке матрицы
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            if (!visited[i][j]) {
                // Если клетка не посещена, запускаем DFS из неё
                dfs(matrix, i, j, matrix[i][j], visited);
                ++count; // Увеличиваем счетчик компонент связности
            }
        }
    }
    return count;
}
```

25. Напишите функцию бинарного возведения в степень по модулю M.

```
typedef unsigned long long ull;

ull fast_pow(
    ull num,
    ull n,
    ull mod
) {
    if (n > 1) {
        ull t = fast_pow(num, n / 2, mod);
        if (n % 2 == 1)
            return (t * t * num) % mod;
        return (t * t) % mod;
    }
    if (n == 1)
        return num % mod;
    return 1;
}
```

15. Напишите функцию для проверки, что в орграфе, заданном через матрицу смежности, существует эйлеров путь (путь, проходящий по всем дугам графа). Сам путь находить не нужно.

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;
//Проверяем, является ли граф сильно связным.
bool inSC(const vector<vector<int>>& matrix) {
    int n = matrix.size();
    vector<bool> visited(n, false);

    auto bfs = [&](int start) {
        fill(visited.begin(), visited.end(), false);
        queue<int> q;
        q.push(start);
        visited[start] = true;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int u = 0; u < n; ++u) {
                if (matrix[v][u] > 0 && !visited[u]) {
                    visited[u] = true;
                    q.push(u);
                }
            }
        }
    };

    // Проверяем достижимость любой вершины ко всем остальным
вершинам.
    bfs(0); //Взяли нулевую вершину.
    if (find(visited.begin(), visited.end(), false) != visited.end())
return false;

    // Проверяем достижимость всех остальных вершин до любой вершины.
    vector<vector<int>> revmatrix(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            revmatrix[j][i] = matrix[i][j];
        }
    }
    bfs(0);
    if (find(visited.begin(), visited.end(), false) != visited.end())
return false;
    return true;
}

//Проверяем есть ли Эйлеров путь.
bool hasEulPath(const vector<vector<int>>& matrix) {
    int n = matrix.size();
    if (!inSC(matrix)) return false;

    vector<int> inDegree(n, 0), outDegree(n, 0);
    for (int u = 0; u < n; ++u) {
```

```
        for (int v = 0; v < n; ++v) {
            if (matrix[u][v] > 0) {
                outDegree[u] += matrix[u][v];
                inDegree[v] += matrix[u][v];
            }
        }
    }

    int startNodes = 0, endNodes = 0;
    for (int i = 0; i < n; ++i) {
        if (outDegree[i] - inDegree[i] == 1) startNodes++;
        else if (inDegree[i] - outDegree[i] == 1) endNodes++;
        else if (inDegree[i] != outDegree[i]) return false;
    }

    return (startNodes == 1 && endNodes == 1) || (startNodes == 0 &&
endNodes == 0);
}

int main() {
    vector<vector<int>> matrix = {
        {0, 1, 0, 0},
        {0, 0, 1, 1},
        {1, 0, 0, 0},
        {0, 0, 1, 0}
    };

    if (hasEulPath(matrix)) cout << "В графе присутствует Эйлеров
путь.\n";
   else cout << "В графе отсутствует Эйлеров путь.\n";
   return 0;
}
```

9. Сравните время работы set и unordered_set из STL для операций поиска с количеством элементов N=100,10000,10^6,10^7 (например, измерить время поиска 10 существующих значений в наборе и 10 несуществующих). Ключами являются числа от 1 до 10^9. Результат оформить в виде таблицы, время в ns. Привести код, использованный для измерения времени для одного значения N.

```
#include <iostream>
#include <algorithm>
#include <set>
#include <unordered_set>
#include <vector>
#include <chrono>
#include <cstdlib>

using namespace std;
void measureTime(int size) {
    srand(time(nullptr));
    set<int> s;
    unordered_set<int> us;
    vector<int> testValues(20);

    // Заполнение set и unordered_set и выбор тестовых значений
    for (int i = 0; i < size; ++i) {
        int val = rand() % 1000000000 + 1;
        s.insert(val);
        us.insert(val);
        if (i < 10) testValues[i] = val;
    }

    for (int i = 10; i < 20; ++i) {
        int val = rand() % 1000000000 + 1;
        if (s.find(val) == s.end()) testValues[i] = val;
    }

    // Измерение времени поиска в set
    long long sum = 0;
    for (int i = 0; i < 20; ++i) {
        auto start = chrono::steady_clock::now();
        s.find(testValues[i]);
        auto end = chrono::steady_clock::now();
        sum += chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    }
    cout << size << " elements in set - " << sum / 20 << " ns\n";

    // Измерение времени поиска в unordered_set
    sum = 0;
    for (int i = 0; i < 20; ++i) {
        auto start = chrono::steady_clock::now();
        us.find(testValues[i]);
        auto end = chrono::steady_clock::now();
        sum += chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    }
    cout << size << " elements in unordered_set - " << sum / 20 << "
ns\n";
}
```

```
int main() {
    vector<int> sizes = {100, 10000, 1000000, 10000000};
    for (int size : sizes) {
        measureTime(size);
        cout << '\n';
    }
    return 0;
}
```

	100	1000	10^6	10^7
set	180	150	175	185
unordered_set	190	195	135	140

16. Есть n ($2 \le n \le 10000$) городов, заданных своими координатами (x_i, y_i), $i \in 1 \dots n$, а также m ($2 \le m \le 100000$) дорог, соединяющих города. Нужно построить дополнительное число дорог (возможно, нулевое), так чтобы из любого города можно было доехать в любой другой, двигаясь по дорогам. При этом сумма длин построенных дорог должна быть минимально возможной.
Укажите какой алгоритм построения минимального остовного дерева является более эффективным для решения этой задачи и обоснуйте свой выбор.

В контексте данной задачи, алгоритм Прима с использованием приоритетной очереди, возможно, будет предпочтительнее из-за его гибкости и прямого подхода к работе с вершинами и ребрами.

Характеристики:

- Хорошо работает для плотных графов (графов с большим числом ребер).
- Не требует заранее отсортированных ребер.
- Разреженные графы (m = O(n)): O((n + m) log n) = O(n log n).
- Плотные графы (m = O(n^2)): O((n + m) log n) = O(n^2 log n).

19. Напишите функцию для поиска строки в большом текстовом файле. Функции передается имя файла с текстом и строка длиной до 10^6 символов. Вы можете хранить в памяти не более 2·10^6 символов из файла и можете считывать файл только последовательно. Оцените эффективность вашего алгоритма.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

// Функция для построения префикс-функции для строки pattern
vector<int> PreF(const string& pattern) {
    int m = pattern.size(), k = 0;
    vector<int> pi(m, 0);
    for (int i = 1; i < m; ++i) {
        while (k > 0 && pattern[k] != pattern[i]) k = pi[k - 1];
        if (pattern[k] == pattern[i]) k++;
    }
    return pi;
}

// Функция для поиска строки в большом текстовом файле
bool search(const string& filename, const string& pattern) {
    ifstream file(filename);
    if (!file.is_open()) {
        cout << "Ошибка при открытии файла.\n";
        return false;
    }

    const size_t bufferSize = 2 * 1000000; // 2 * 10^6 символов
    vector<char> buffer(bufferSize);

    vector<int> pi = PreF(pattern);
    int m = pattern.size(), k = 0; // k-счетчик совпавших символов

    while (file) {
        file.read(buffer.data(), bufferSize);
        size_t bytesRead = file.gcount();

        for (size_t i = 0; i < bytesRead; ++i) {
            while (k > 0 && pattern[k] != buffer[i]) k = pi[k - 1];
            if (pattern[k] == buffer[i]) k++;
            if (k == m) { // Найдено совпадение
                file.close();
                return true;
            }
        }
    }
    file.close();
    return false;
}
```

```
int main() {
    string filename = "text_file.txt";
    string pattern = "pattern";

    bool found = search(filename, pattern);

    if (found) cout << "Строка найдена.\n";
    else cout << "Строка не найдена.\n";
    return 0;
}
```

4. Предложите структуры данных для представления АТД из задания 3. Перечислите поля, их типы и комментарии к каждому полю. Укажите оценку эффективности (амортизированную или среднюю) для каждого метода с учетом использованных структур данных. Хранимая в структуре информация не должна дублироваться.

Для представления XML-документа можно использовать дерево, где каждый узел представляет элемент XML. Каждый узел содержит информацию о теге, атрибутах, вложенных элементах и указатели на родительский элемент, предыдущий и следующий элементы на том же уровне.

```
#include <string>
#include <unordered_map>
#include <vector>

struct XmlNode {
    std::string tagName; // Имя тега
    std::unordered_map<std::string, std::string> attributes; // Список
атрибутов (пара имя-значение)
    std::vector<XmlNode*> children; // Последовательность вложенных
элементов
    XmlNode* parent; // Указатель на родительский элемент
    XmlNode* prev; // Указатель на предыдущий элемент на том же уровне
    XmlNode* next; // Указатель на следующий элемент на том же уровне

    // Конструктор
    XmlNode(const std::string& name)
        : tagName(name), parent(nullptr), prev(nullptr), next(nullptr)
{}
};
```

- Поля структуры XmlNode:

- tagName (std::string): имя тега элемента XML.
- attributes (std::unordered_map<std::string, std::string>): ассоциативный массив для хранения атрибутов элемента в виде пар имя-значение.
- children (std::vector<XmlNode*>): вектор указателей на дочерние элементы, представляющие вложенные теги.
- parent (XmlNode*): указатель на родительский элемент, используется для навигации вверх по дереву.
- prev (XmlNode*): указатель на предыдущий элемент на том же уровне, используется для последовательной навигации.
- next (XmlNode*): указатель на следующий элемент на том же уровне, используется для последовательной навигации.

- Операции и их эффективность:

Вставка элемента

Вставка дочернего элемента:
Добавление дочернего элемента в вектор children родительского узла.
Временная сложность: O(1) (амортизированная) для добавления в конец вектора.

Вставка на том же уровне:
Настройка указателей prev и next для вставляемого элемента и его соседей.
Временная сложность: O(1).

Удаление элемента
Обновление указателей prev, next, parent и удаление элемента из вектора children родительского узла.
Временная сложность: O(n) для поиска элемента в векторе children родителя, где n — количество детей у родителя. В случае, если есть прямой доступ к элементу и его позиции, сложность будет O(1) для обновления указателей.

Поиск элемента
Поиск по имени тега:
Линейный поиск по всем узлам дерева.
Временная сложность: O(n), где n — количество узлов в дереве.
Поиск по атрибуту:
Линейный поиск с проверкой атрибутов у каждого узла.
Временная сложность: O(n * m), где n — количество узлов, m — среднее количество атрибутов в узле.

Доступ к родительскому, предыдущему и следующему элементу
Временная сложность: O(1), так как используется прямой указатель.