Cuprins

Fișiere comume

utils.h

```
#pragma once
#include <initializer_list>
#include <iostream>

constexpr size_t getSize(std::initializer_list<double> l) {
    size_t n = 0;
    auto it = l.begin();
    auto end = l.end();
    while (it++ != end) ++n;
    return n;
}

constexpr void getSize(std::initializer_list<
                           std::initializer_list<double>> list,
                       int& s1, int& s2) {
    constexpr int invalid = -1;
    s1 = 0;
    s2 = invalid;

    for (const auto& l: list) {
        int i = getSize(l);
        ++s1;
        if (s2 != invalid && i != s2) {
            //std::cerr << i << "!=" << s2 << "\n";
            throw std::logic_error("Fixed line size expected");
        }
        s2 = i;
    }
    //std::cerr << "size: " << s1 << ", " << s2 << "\n";
}
constexpr int MAX_SZ = 256;
inline size_t readSize(const char* name, int a = 1, int b = MAX_SZ) {
    int res;
    do {
        std::cout << name << ": ";
        std::cin >> res;
    } while (res < a || res > b);
    return res;
}
```

```cpp


inline void assert(bool cond, const char* msg) {
    if (!cond) throw std::logic_error(msg);
}
template<typename T>
inline T read(const char* name) {
    T res;
    std::cout << name << ": ";
    std::cin >> res;
    return res;
}
```

matrix.h

```cpp
#pragma once
#include "utils.h"

#include <iostream>
#include <cmath>
#include <cstring>
#include <utility>

struct Mat {
    int m, n;
    double *data;
    Mat() : m(0), n(0), data(nullptr) {}
    Mat(int m, int n) : m(m), n(n) { data = new double[m*n]; }
    //template<int m, int n>
    Mat(std::initializer_list<std::initializer_list<double>> list) {
        getSize(list, m, n);
        data = new double[m*n];

        auto it = begin();
        for (const auto& l : list) {
            for (const auto& v : l) *(it++) = v;
        }
    }
    Mat(Mat&& rhs) noexcept :
        m(rhs.m), n(rhs.n),
        data(std::exchange(rhs.data, nullptr)) {
        //rhs.n = rhs.m = 0;
    }

    Mat(const Mat& rhs) : Mat(rhs.m, rhs.n) {
        std::memcpy(data, rhs.data, sizeof(double) *m*n);
    }
    Mat& operator=(const Mat& rhs) {
        setSize(rhs.m, rhs.n);
        std::memcpy(data, rhs.data, sizeof(double) *m*n);
```

```
        return *this;
    }
    Mat& operator=(Mat&& rhs) noexcept {
        this->~Mat();
        m = rhs.m;
        n = rhs.n;
        data = std::exchange(rhs.data, nullptr);

        return *this;
    }

    ~Mat() {
        delete[] data;
    }
    double& at(int i, int j) {
        assert(i < m && j < n, "out of range");
        return data[i *n +j];
    }
    double at(int i, int j) const { return data[i * n + j]; }
    double* begin() { return data; }
    const double* begin() const { return data; }
    double* end() { return data + m*n; }
    const double* end() const { return data + m*n; }

    void setSize(int m, int n) {
        if (this->m == m && this->m == n) return;
        this->~Mat();
        new (this) Mat(m, n);
    }

    static Mat read() {
        Mat res(readSize("m"), readSize("n"));
        for (auto& i : res)
            std::cin >> i;
        return res;
    }
    void print(const char* name) const {
        std::cout << name << " = Mat " << m << "x" << n << "{\n";

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                std::cout << at(i, j) << " ";
            }
            std::cout << "\n";
        }
        std::cout << "}\n";
    }

    friend std::ostream& operator<<(std::ostream& s, Mat& m) {
        s << "Mat " << m.m << "x" << m.n << "{\n";
```

```
        for (int i = 0; i < m.m; ++i) {
            for (int j = 0; j < m.n; ++j) {
                s << m.at(i, j) << " ";
            }
            s << "\n";
        }
        return s<< "}\n";
    }

    double norm1() const { return normImpl<Type::Row>(n, m); }
    double normInf() const { return normImpl<Type::Col>(m, n); }
    double normF() const {
        double res = 0;
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
                res += at(i, j) * at(i, j);

        return std::sqrt(res);
    }
    bool isStrictlyRowDiagonallyDominant() const {
        return isStrictlyDiagonallyDominantImpl<Type::Row>();
    }

    bool isStrictlyColDiagonallyDominant() const {
        return isStrictlyDiagonallyDominantImpl<Type::Col>();
    }

private:

    enum class Type { Row, Col };
    template<Type type>
    bool isStrictlyDiagonallyDominantImpl() const {
        assert(m == n, "Matrix must be square");
        for (int i = 0; i < m; ++i) {
            double val = std::abs(at(i, i));
            double sum = -val;
            for (int j = 0; j < m; ++j) {
                sum += std::abs(type == Type::Col? at(j, i): at(i, j));
            }
            if (sum >= val) {
                std::cout << "(" << sum << "," << val << ")";
                return false;
            }
        }
        return true;
    }
    template<Type type>
    double normImpl(int sz1, int sz2) const {
        double max = -1;
        for (int j = 0; j < sz1; ++j) {
```

```
            double x = 0;
            for (int i = 0; i < sz2; ++i) {
                x += std::abs(type == Type::Col? at(j, i) : at(i, j));
            }
            if (x > max) max = x;
        }
        return max;
    }
};

Mat& add(const Mat& a, const Mat& b, Mat& res) {
    assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
    res.setSize(a.m, a.n);
    for (int i = 0; i < a.m; ++i)
        for (int j = 0; j < a.n; ++j)
            res.at(i, j) = a.at(i, j) + b.at(i, j);
    return res;
}

Mat& mul(double a, const Mat& b, Mat& res) {
    res.setSize(b.m, b.n);

    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j)
            res.at(i, j) = a * b.at(i, j);
    return res;
}

Mat& neg(const Mat& a, Mat& res) {
    return mul(-1, a, res);
}
Mat& sub(const Mat& a, const Mat& b, Mat& res) {
    return add(a, neg(b, res), res);
}
Mat& mul(const Mat& a, const Mat& b, Mat& res) {
    assert(a.n == b.m, "Sizes don't match, can't multiply");
    res.setSize(a.m, b.n);

    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j) {
            res.at(i, j) = 0;
            for (int k = 0; k < a.n; ++k)
                res.at(i, j) += a.at(i, k) * b.at(k, j);
        }
    return res;
}

Mat& trans(const Mat& a, Mat& res) {
    assert(a.data != res.data, "Can't transpose inplace");
    res.setSize(a.n, a.m);
```

```
    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j)
            res.at(i, j) = a.at(j, i);
    return res;
}
```

vector.h

```
#pragma once
#include "utils.h"

#include <iostream>
#include <utility>
#include <cmath>

struct Vec {
    double *_begin, *_end;

    constexpr double* begin() { return _begin; }
    constexpr const double* begin() const { return _begin; }

    constexpr double* end() { return _end; }
    constexpr const double* end() const { return _end; }

    constexpr Vec() : _begin(nullptr), _end(nullptr) {}
    explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
    Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
        auto it = _begin;
        for (const auto& v : list) *(it++) = v;
    }
    Vec(const Vec& rhs) : Vec(rhs.size()) {
        auto it = _begin;
        for (const auto& v : rhs) *(it++) = v;
    }
    Vec(Vec&& rhs) noexcept
        : _begin(std::exchange(rhs._begin, nullptr)),
        _end(std::exchange(rhs._end, nullptr)) {}

    Vec& operator=(const Vec& rhs) {
        if (size() != rhs.size()) {
            this->~Vec();
            new (this) Vec(rhs.size());
        }
        auto it = _begin;
        for (const auto& v : rhs) *(it++) = v;
        return *this;
    }
    Vec& operator=(Vec&& rhs) noexcept {
        this->~Vec();
```

```cpp
        _begin = std::exchange(rhs._begin, nullptr);
        _end = std::exchange(rhs._end, nullptr);
        return *this;
    }

    ~Vec() { delete[] _begin; }

    constexpr size_t size() const { return _end - _begin; }

    constexpr double& operator[](size_t i) { return _begin[i]; }
    constexpr double operator[](size_t i) const { return _begin[i]; }

    void setSize(size_t n) {
        if (size() == n) return;
        *this = Vec(n);
    }

    friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
        s << "(";
        double* it = v._begin;
        for (double* end = v._end - 1; it < end; ++it)
            s << *it << ", ";

        if (it < v._end) s << *it;

        return s << ")";
    }
    static Vec read() {
        Vec res(readSize("n"));
        for (auto& v: res) std::cin >> v;
        return res;
    }
    double norm() const;
};
void assertSizes(const Vec& a, const Vec& b) {
    assert(a.size() == b.size(), "Sizes don't match");
}

Vec& add(const Vec& a, const Vec& b, Vec& res) {
    assertSizes(a, b);
    res.setSize(a.size());
    auto aIt = a.begin();
    auto bIt = b.begin();
    for (auto& v : res) v = *(aIt++) + *(bIt++);
    return res;
}

Vec& mul(double a, const Vec& b, Vec& res) {
    res.setSize(b.size());
    auto it = b.begin();
```

```
    for (auto& v : res) v = a * (*(it++));
    return res;
}

Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }

Vec& sub(const Vec& a, const Vec& b, Vec& res) {
    return add(a, neg(b, res), res);
}
double dot(const Vec& a, const Vec& b) {
    double res = 0;
    assertSizes(a, b);
    auto bIt = b.begin();
    for (auto& v : a) res += v * (*(bIt++));
    return res;
}
double norm(const Vec& a) {
    return std::sqrt(dot(a, a));
}
double Vec::norm() const {
    return ::norm(*this);
}
```

list.h

```
#include "utils.h"

#include <type_traits>
#include <utility>


// We should never check for (bot == nullptr),
// so we don't update it when the list becomes empty.
template<typename T>
struct List {
    struct Node {
        T val;
        Node* next;
    };
    Node* top;
    Node* bot;
    constexpr List(Node* top = nullptr, Node* bot = nullptr)
        : top(top), bot(bot) {}
    List(const List&) = delete;
    List& operator=(const List&) = delete;
    ~List() {
        while (top) {
            Node* n = top;
            top = top->next;
            delete n;
```

```
        }
    }
    List(List&& rhs) noexcept
        : top(std::exchange(rhs.top, nullptr)),
          bot(std::exchange(rhs.bot, nullptr)) {}
    List& operator=(List&& rhs) noexcept {
        this->~List();
        this->top = std::exchange(rhs.top, nullptr);
        this->bot = std::exchange(rhs.bot, nullptr);
    }

    T& first() { return top->val; }
    T& last() { return bot->val; }
    T pop_front() {
        assert(top, "Empty list");
        Node* n = top;
        top = top->next;
        T res = n->val;
        delete n;
        return res;
    }

    void push_front(T val) {
        Node *n = new Node{ val, top };
        top = n;
    }
    void push_back(T val) {
        Node *n = new Node{ val, nullptr };
        if (top == nullptr) {
            top = n;
        }
        else {
            bot->next = n;

        }
        bot = n;
    }
    bool empty() const { return top == nullptr; }

    bool operator== (const List& rhs) {
        for (auto it = top, rit = rhs.top;
            it != nullptr;
            it = it->next, rit = rit->next) {
            if (it->val != rit->val) return false;
        }
        return true;
    }

    static List read(const char* msg) {
        List q;
```

```
        std::cout << msg << ":\n";
        int len1 = readSize("n");
        for (int i = 0; i < len1; ++i) {
            T s;
            std::cin >> s;
            q.push_front(s);
        }
        return std::move(q);
    }

    friend std::ostream& operator<<(std::ostream& s, const List& st) {
        if constexpr (std::is_same_v<T, char>) {
            const Node* it = st.top;
            while (it != nullptr) {
                s << it->val;
                it = it->next;
            }
            return s;
        }
        else {
            if (st.empty()) return s << "{}";
            s << "{";
            const Node* it = st.top;
            while (it->next != nullptr) {
                s << it->val << ", ";
                it = it->next;
            }

            return s << it->val << "}";
        }
    }

    void remove(T& val) {
        remove_if([&] (T& t) { return t ==val; });
    }
    // removes all elements that satisfy p
    template<class P>
    void remove_if(P p) {
        apply_on(p, [] (Node* n) { delete n; } );
    }
    //applies f() on all nodes that satisfy the predicate p()
    template<class P, class F>
    void apply_on(P p, F f) {
        for (;;) {
            auto* n = top;
            if (!n) {
                return;
            }
            if (!p(n->val)) break;
            top = n->next;
```

```
            f(n);
        }
        auto* prev = top;
        auto* it = prev->next;

        while (it) {
            if (p(it->val)) {
                prev->next = it->next;
                if (prev->next == nullptr) {
                    bot = prev;
                }
                f(it);
                it = prev;
            } else {
                prev = it;
                it = it->next;
            }
        }
    }
};

// keeps in l all the elements that don't satisfy the predicate p
// and retuns pair of:
//   - a reference to the original list
//   - a list containing the elements that satisfy p
template<typename T, typename P>
constexpr auto partition_split(List<T>& l, P p) {
    struct res_t {
        List<T>& notSatisfying;
        List<T> satisfying;
    } res = { l, {} };
    List<T>& sat = res.satisfying;
    auto insertNode = [&sat](auto* n) {
        n->next = sat.top;
        sat.top = n;
        if (sat.bot == nullptr) {
            sat.bot = n;
        }
    };
    l.apply_on(p, insertNode);
    return res;
}
```

stack.h

```
#include "utils.h"

template<typename T>
struct Stack {
    struct Node {
```

```
        T val;
        Node* next;
    };

    Stack(const Stack&) = delete;
    Stack& operator=(const Stack&) = delete;
    Node* top;
    Stack(Node* top = nullptr) : top(top) {}
    ~Stack() {
        while (top) {
            Node* n = top;
            top = top->next;
            delete n;
        }
    }
    void push(T val) {
        Node *n = new Node{ val, top };
        top = n;
    }
    void quickPop() {
        Node* n = top;
        top = top->next;
        delete n;
    }
    T pop() {
        assert(top, "Empty stack");
        Node* n = top;
        top = top->next;
        T res = n->val;
        delete n;
        return res;
    }
    bool empty() const { return top == nullptr; }
    friend std::ostream& operator<<(std::ostream& s, const Stack& st) {
        if (st.empty()) return s << "{}";
        s << "{";
        const Node* it = st.top;
        while (it->next != nullptr) {
            s << it->val << ", ";
            it = it->next;
        }

        return s << it->val << "}";
    }
};
```

queue.h

```
#include "utils.h"

```

```cpp
template<class T>
struct Queue {
    struct Node {
        T val;
        Node* next;
    };
    Node* top;
    Node* bot;
    Queue(Node* top = nullptr, Node* bot = nullptr) : top(top), bot(bot) {}
    /*
    Queue(const Queue&) = delete;
    Queue& operator=(const Queue&) = delete;
    ~Queue() {
        while (top) {
            Node* n = top;
            top = top->next;
            delete n;
        }
        }*/
    void push(T val) {
        Node *n = new Node{ val, nullptr };
        if (bot == nullptr) {
            top = n;
        }
        else {
            bot->next = n;
        }
        bot = n;
    }

    T pop() {
        assert(top, "Empty queue");
        Node* old = top;
        T res = old->val;
        top = top->next;
        if (top == nullptr)
            bot = nullptr;
        delete old;
        return res;
    }
    bool empty() const { return top == nullptr; }
    static Queue read(const char* msg) {
        Queue q;
        std::cout << msg  << ":\n";
        int len1 = readSize("n");
        for (int i = 0; i < len1; ++i) {
            T s;
            std::cin >> s;
            q.push(s);
        }
```

```
        return std::move(q);
    }

    friend std::ostream& operator<<(std::ostream& s, const Queue& st) {
        if (st.empty()) return s << "{}";
        s << "{";
        const Node* it = st.top;
        while (it->next != nullptr) {
            s << it->val << ", ";
            it = it->next;
        }

        return s << it->val << "}";
    }
};
```

Alocarea dinamică a memoriei. Tipuri specifice.

Laborator 1

16. Scrieți funcții pentru implemetarea operațiilor specifice pe matrice de numere reale cu m linii și n coloane: suma, diferența și produsul al două matrice, produsul dintre o matrice și un scalar real, transpusa unei matrice, norme matriceale specifice[1], citirea de la tastatură a componentelor unei matrice, afișarea componentelor matricei. Pentru cazul particular al unei matrice patratice de ordin n, să se testeze dacă aceasta satisface criteriul de dominanță pe linii[2] sau pe coloane[3]. Se vor folosi tablouri bidimensionale alocate static.

```
#include "utils.h"

#include <iostream>
#include <cmath>

struct Mat {
    double data[MAX_SZ][MAX_SZ] {};
    int m, n;

    Mat() : m(0), n(0) {}
    Mat(int m, int n) : m(m), n(n) {}

    static Mat read() {
        Mat res(readSize("m"), readSize("n"));

        for (int i = 0; i < res.m; ++i)
            for (int j = 0; j < res.n; ++j)
                std::cin >> res.data[i][j];
        return res;
    }
    void setSize(int m, int n) {
        this->m = m;
        this->n = n;
    }
```

[1] Dacă $A \in \mathcal{M}_{m\times n}(\mathbb{R})$, atunci $||A||_1 = \max\limits_{1\le j\le n} \sum\limits_{i=1}^{m} |a_{ij}|$, $||A||_\infty = \max\limits_{1\le i\le m} \sum\limits_{j=1}^{n} |a_{ij}|$, $||A||_F = \sqrt{\sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n} a_{ij}^2}$.

[2] $A \in \mathcal{M}_n(\mathbb{R})$ este strict diagonal dominantă pe linii dacă $|a_{ii}| > \sum\limits_{\substack{j=1\\ j\ne i}}^{n} |a_{ij}|$, pentru orice $i = 1, ..., n$.

[3] $A \in \mathcal{M}_n(\mathbb{R})$ este strict diagonal dominantă pe colonane dacă $|a_{jj}| > \sum\limits_{\substack{i=1\\ i\ne j}}^{n} |a_{ij}|$, pentru orice $j = 1, ..., n$.

```cpp
    double& at(int i, int j) { return data[i][j]; }
    double at(int i, int j) const { return data[i][j]; }

    void print(const char* name) const {
        std::cout << name << " = Mat " << m << "x" << n << "{\n";
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j)
                std::cout << at(i, j) << " ";
            std::cout << "\n";
        }
        std::cout << "}\n";
    }
private:
    enum class Type { Row, Col };
    template<Type type>
    bool isStrictlyDiagonallyDominantImpl() const {
        assert(m == n, "Matrix must be square");
        for (int i = 0; i < m; ++i) {
            double val = std::abs(at(i, i));
            double sum = -val;
            for (int j = 0; j < m; ++j)
                sum += std::abs(type == Type::Col ? at(j, i) : at(i, j));
            if (sum >= val) return false;
        }
        return true;
    }

    template<Type type>
    double normImpl(int sz1, int sz2) const {
        double max = -1;
        for (int j = 0; j < sz1; ++j) {
            double x = 0;
            for (int i = 0; i < sz2; ++i)
                x += std::abs(type == Type::Col ? at(j, i) : at(i, j));
            if (x > max) max = x;
        }
        return max;
    }
public:
    double norm1() const { return normImpl<Type::Row>(n, m); }
    double normInf() const { return normImpl<Type::Col>(m, n); }
    double normF() const {
        double res = 0;
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
                res += at(i, j) * at(i, j);

        return std::sqrt(res);
    }
    bool isStrictlyRowDiagonallyDominant() const {
```

```
        return isStrictlyDiagonallyDominantImpl<Type::Row>();
    }
    bool isStrictlyColDiagonallyDominant() const {
        return isStrictlyDiagonallyDominantImpl<Type::Col>();
    }
};

Mat& add(const Mat& a, const Mat& b, Mat& res) {
    assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
    res.setSize(a.m, a.n);
    for (int i = 0; i < a.m; ++i)
        for (int j = 0; j < a.n; ++j)
            res.at(i, j) = a.at(i, j) + b.at(i, j);
    return res;
}
Mat& mul(double a, const Mat& b, Mat& res) {
    res.setSize(b.m, b.n);

    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j)
            res.at(i, j) = a * b.at(i, j);
    return res;
}
Mat& neg(const Mat& a, Mat& res) { return mul(-1, a, res); }
Mat& sub(const Mat& a, const Mat& b, Mat& res) {
    return add(a, neg(b, res), res);
}
Mat& mul(const Mat& a, const Mat& b, Mat& res) {
    assert(a.n == b.m, "Sizes don't match, can't multiply");
    res.setSize(a.m, b.n);

    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j) {
            res.at(i, j) = 0;
            for (int k = 0; k < a.n; ++k)
                res.at(i, j) += a.at(i, k) * b.at(k, j);
        }
    return res;
}
Mat& trans(const Mat& a, Mat& res) {
    assert(a.data != res.data, "Can't calculate the transpose inplace");
    res.setSize(a.n, a.m);

    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j)
            res.at(i, j) = a.at(j, i);
    return res;
}
```

Laborator 2

18. Scrieți funcții pentru implementarea operațiilor specifice pe vectori din $\mathbb{R}^n$: suma, diferența și produsul scalar al doi vectori, produsul dintre un vector și un scalar real, negativarea unui vector, norma euclidiană a unui vector, citirea de la tastură a celor n componente ale unui vector, afișarea componentelor vectorului sub forma unui n-uplu de elemente. Se vor folosi tablouri unidimensionale alocate dinamic.

```cpp
#include "utils.h"

#include <iostream>
#include <utility>
#include <cmath>

struct Vec {
    double *_begin, *_end;

    constexpr double* begin() { return _begin; }
    constexpr const double* begin() const { return _begin; }


    constexpr double* end() { return _end; }
    constexpr const double* end() const { return _end; }

    constexpr Vec() : _begin(nullptr), _end(nullptr) {}
    explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
    Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
        auto it = _begin;
        for (const auto& v : list) *(it++) = v;
    }
    Vec(const Vec&) = delete;
    Vec(Vec&& rhs) noexcept
        : _begin(std::exchange(rhs._begin, nullptr)),
        _end(std::exchange(rhs._end, nullptr)) {}

    Vec& operator=(const Vec&) = delete;
    Vec& operator=(Vec&& rhs) noexcept {
        this->~Vec();
        _begin = std::exchange(rhs._begin, nullptr);
        _end = std::exchange(rhs._end, nullptr);
        return *this;
    }

    ~Vec() { delete[] _begin; }

    constexpr size_t size() const { return _end - _begin; }

    constexpr double& operator[](size_t i) { return _begin[i]; }
    constexpr double operator[](size_t i) const { return _begin[i]; }

    void setSize(size_t n) {
```

```
        if (size() == n) return;
        *this = Vec(n);
    }

    friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
        s << "(";
        double* it = v._begin;
        for (double* end = v._end - 1; it < end; ++it)
            s << *it << ", ";

        if (it < v._end) s << *it;

        return s << ")";
    }
    static Vec read() {
        Vec res(readSize("n"));
        for (auto& v: res) std::cin >> v;
        return res;
    }
    double norm() const;
};
void assertSizes(const Vec& a, const Vec& b) {
    assert(a.size() == b.size(), "Sizes don't match");
}

Vec& add(const Vec& a, const Vec& b, Vec& res) {
    assertSizes(a, b);
    res.setSize(a.size());
    auto aIt = a.begin();
    auto bIt = b.begin();
    for (auto& v : res) v = *(aIt++) + *(bIt++);
    return res;
}

Vec& mul(double a, const Vec& b, Vec& res) {
    res.setSize(b.size());
    auto it = b.begin();
    for (auto& v : res) v = a * (*(it++));
    return res;
}

Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }

Vec& sub(const Vec& a, const Vec& b, Vec& res) {
    return add(a, neg(b, res), res);
}
double dot(const Vec& a, const Vec& b) {
    double res = 0;
    assertSizes(a, b);
    auto bIt = b.begin();
```

```
    for (auto& v : a) res += v * (*(bIt++));
    return res;
}
double norm(const Vec& a) {
    return std::sqrt(dot(a, a));
}
double Vec::norm() const {
    return ::norm(*this);
}
```

Tablouri

Laborator 3

7. Folosind structurile de date `VECTOR` și `MATRICE` definite la curs și funcțiile necesare, rezolvați următorul sistem algebric liniar cu n ecuații și n necunoscute folosind metoda lui Gauß de eliminare.

$$\begin{cases} 2x_1 - x_2 = 1 \\ -x_1 + 2x_2 - x_3 = 1 \\ -x_2 + 2x_3 - x_4 = 1 \\ \dots\dots\dots\dots \\ -x_{n-2} + 2x_{n-1} - x_n = 1 \\ -x_{n-1} + 2x_n = 1, \quad n \in \mathbb{N}, 2 \le n \le 50 \end{cases}.$$

```
#include "utils.h"

#include "vector.h"
#include "matrix.h"

#include <iostream>
#include <cmath>

constexpr double eps = 1e-7;

Vec& mul(const Mat& m, const Vec& v, Vec& res) {
    assert(v.begin() != res.begin(), "Can't multiply inplace");
    assert(v.size() == size_t(m.n), "Sizes don't match");
    res.setSize(m.m);


    for (size_t i = 0; i < res.size(); ++i) {
        res[i] = 0;
        for (size_t k = 0; k < v.size(); ++k)
            res[i] += m.at(i, k) * v[k];
    }
    return res;
}

// A * X = b
struct System {
    Mat A;
```

```cpp
    Vec b;

    System(int n, int m) : A(n, m), b(n) {}

    System(std::initializer_list<std::initializer_list<double>> A,
           std::initializer_list<double> b) : A(A), b(b) {
           assert(std::size_t(this->A.m) == this->b.size(),
                  "sizes don't match");
    }

    friend std::ostream& operator<<(std::ostream& s, const System& sys) {
        s << "System " << sys.A.m << "x" << sys.A.n << ": \n";
        auto& A = sys.A;
        for (int i = 0; i < A.m; ++i) {
            s << "{";
            for (int j = 0; j < A.n; ++j) {
                //showpos shows a '+' in front of positive numbers
                if (std::abs(A.at(i, j)) > eps)
                    s << std::showpos << A.at(i, j)
                      << std::noshowpos << "*x" << (j+1) << " ";
            }
            s<< "= " << sys.b[i] << "\n";
        }

        return s;
    }

    // L_i += f * L_j
    void addLines(int i, double f, int j) {
        for (int k = 0; k < A.n; ++k) {
            A.at(i, k) += f * A.at(j, k);
        }
        b[i] += f * b[j];
    }

    // L_i *= f
    void multiplyLine(int i, double f) {
        for (int k = 0; k < A.n; ++k) {
            A.at(i, k) *= f;
        }
        b[i] *= f;
    }

    Vec solveTriangulated() {
        for (int i = A.m-1; i > 0; --i) {
            addLines(i-1, -A.at(i-1, i), i);
            multiplyLine(i, 1 / A.at(i, i));
        }
        return b;
    }
```

```
    bool checkSolution(const Vec& x) const {
        Vec r;
        mul(A, x, r);
        sub(b, r, r);
        for (auto& v : r) {
            if (std::abs(v) > eps) return false;
        }
        return true;
    }

    static Vec solveCustom(int n) {
        System s(n, n);
        for (auto& v : s.A) v = 0;
        s.A.at(0,0) = 2;
        s.A.at(0,1) = -1;
        s.b[0] = 1;
        for (int i = 1; i < n - 1; ++i) {
            s.b[i] = 1;

            s.A.at(i,i-1) = -1;
            s.A.at(i,i) = 2;
            s.A.at(i,i+1) = -1;
        }
        s.b[n-1] = 1;
        s.A.at(n-1, n-2) = -1;
        s.A.at(n-1, n-1) = 2;
        s.customTriangulate();
        return s.solveTriangulated();
    }
    void customTriangulate() {
        multiplyLine(0, 1 / A.at(0, 0));
        for (int i = 1; i < A.m; ++i) {
            addLines(i, 1, i-1);
            multiplyLine(i, 1 / A.at(i, i));
        }
    }
};

int main() {
    try {
        int n = readSize("n", 2, 51);
        std::cout << "x = "<< System::solveCustom(n) << "\n";
    } catch (std::exception& e) {
        std::cerr << "Error" << e.what() << "\n";
        return 1;
    }
    return 0;
}
```

Liste liniare simplu înlănțuite
Stive și cozi

Laborator 4

5. Se citește un text de la tastatura (poate conține orice caracter, inclusiv spații) și se încarcă în două stive: o stivă va conține doar litere mici, iar cealaltă doar litere mari. Se citește de la tastatură o vocală a alfabetului englez (literă mare sau mică). Ștergeți stiva corespunzătore până la întâlnirea vocalei citite.

```cpp
#include "utils.h"
#include "stack.h"

#include <iostream>
#include <cstdlib>

bool isVowel(char c) {
    c = tolower(c);
    return c == 'a' || c== 'e' || c== 'i' || c=='o' || c=='u';
}

int main() {
    std::string str;
    std::cout << "str: ";
    std::getline(std::cin, str);

    char v;
    do {
        v = read<char>("vowel");
    } while (!isVowel(v));

    Stack<char> lower;
    Stack<char> upper;
    auto printStacks = [&] (const char* s) {
                            std::cout << s;
                            std::cout << "lower:" << lower << "\n";
                            std::cout << "upper:" << upper << "\n";
                        };

    printStacks("Before:\n");
    for (auto& c : str) {
        if (islower(c)) lower.push(c);
        else if (isupper(c)) upper.push(c);
```

```
    }

    printStacks("After Adding:\n");
    Stack<char>& stack = isupper(v)? upper: lower;
    while (!stack.empty()) {
        if (char c = stack.pop(); c == v) {
            break;
        }
    }
    printStacks("Result:\n");
    return 0;
}
```

Laborator 5

11. Creați o listă liniară simplu înlănțuită în nodurile căreia sunt memorate numere naturale. Sepratați numerele naturale memorate în listă, în două liste, una corespunzătore numerelor pare și cealaltă, numerelor impare. Afișați cele două liste. Ștergeți din lista numerelor pare, o valoare pară x, citită de la tastatură, ori de câte ori aprare în listă.

```
#include "utils.h"
#include "list.h"

#include <iostream>

int main() {
    auto nums = List<int>::read("numbers");

    auto [evens, odds] = partition_split(nums, [](int n) {return n % 2;});
    std::cout << "odd:  " << odds << "\n";
    std::cout << "even: " << evens << "\n";

    int x;
    do {
        std::cout << "x (must be even): ";
        std::cin >> x;
    } while (x % 2);

    std::cout << "removing...\n";
    evens.remove(x);

    std::cout << "even: " << evens << "\n";
    return 0;
}
```

17. Modelați printr-o LLSI un stoc de produse caracterizate prin: denumire, unitate de măsură, cantitate și preț unitar. Implementați principalele operații pe stoc: crearea stocului, introducerea unui produse nou, eliminarea unui produs in cazul în care acesta a fost vândut în întregime, modificare informații despre un produs (de exemplu, modificarea cantitații unui produs, în cazul vânzării), calculul valorii stocului la un moment dat, listare stoc.

```
#include <iostream>
```