Cuprins

Fișiere comune **2**
utils.h 2
matrix.h 3
vector.h 7
stack.h 9
queue.h 11
list.h 12
doubleList.h 17
graphEdgeList.h 21
graphAdjacencyList.h 22
ptrRange.h 25
dirGraphMat.h 30
dirGraphAdjacencyList.h 31
weightedDirGraphMat.h 34

Alocarea dinamica a memoriei. Tipuri specifice. **36**

Tablouri **42**

Liste liniare simplu înlănțuite. Stive și cozi **45**
stock.h 46
fixedPoint.h 53
inputHelper.h 57

Liste liniare dublu înlănțuite **60**

Grafuri neorientate **63**

Grafuri orientate. Parcurgerea grafurilor **67**

Parcurgerea grafurilor. Grafuri ponderate. Drumuri de cost minim **70**

Arbori **73**

Fișiere comune

utils.h

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

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;
```

```
}


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;
}

template<typename It>
constexpr auto average(It begin, It end) {
    std::remove_const_t<std::remove_reference_t<decltype(*begin)>> sum {};
    size_t count = 0;
    while (begin != end) {
        sum += *begin;
        ++begin;
        ++count;
    }
    return sum /count;
}
size_t fileLineCount(std::ifstream& f) {
    size_t count = 0;
    while (!f.eof()) {
        if (auto c = f.get(); c == '\n') ++count;
    }
    return count;
}
```

matrix.h

```
#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) {
```

```
16          getSize(list, m, n);
17          data = new double[m*n];
18
19          auto it = begin();
20          for (const auto& l : list) {
21              for (const auto& v : l) *(it++) = v;
22          }
23      }
24      Mat(Mat&& rhs) noexcept :
25          m(rhs.m), n(rhs.n),
26          data(std::exchange(rhs.data, nullptr)) {
27          //rhs.n = rhs.m = 0;
28      }
29
30      Mat(const Mat& rhs) : Mat(rhs.m, rhs.n) {
31          std::memcpy(data, rhs.data, sizeof(double) *m*n);
32      }
33      Mat& operator=(const Mat& rhs) {
34          setSize(rhs.m, rhs.n);
35          std::memcpy(data, rhs.data, sizeof(double) *m*n);
36          return *this;
37      }
38      Mat& operator=(Mat&& rhs) noexcept {
39          this->~Mat();
40          m = rhs.m;
41          n = rhs.n;
42          data = std::exchange(rhs.data, nullptr);
43
44          return *this;
45      }
46
47      ~Mat() {
48          delete[] data;
49      }
50      double& at(int i, int j) {
51          assert(i < m && j < n, "out of range");
52          return data[i *n +j];
53      }
54      double at(int i, int j) const { return data[i * n + j]; }
55      double* begin() { return data; }
56      const double* begin() const { return data; }
57      double* end() { return data + m*n; }
58      const double* end() const { return data + m*n; }
59
60      void setSize(int m, int n) {
61          if (this->m == m && this->m == n) return;
62          this->~Mat();
63          new (this) Mat(m, n);
64      }
65
```

```
66      static Mat read() {
67          Mat res(readSize("m"), readSize("n"));
68          for (auto& i : res)
69              std::cin >> i;
70          return res;
71      }
72      void print(const char* name) const {
73          std::cout << name << " = Mat " << m << "x" << n << "{\n";
74
75          for (int i = 0; i < m; ++i) {
76              for (int j = 0; j < n; ++j) {
77                  std::cout << at(i, j) << " ";
78              }
79              std::cout << "\n";
80          }
81          std::cout << "}\n";
82      }
83
84      friend std::ostream& operator<<(std::ostream& s, Mat& m) {
85          s << "Mat " << m.m << "x" << m.n << "{\n";
86          for (int i = 0; i < m.m; ++i) {
87              for (int j = 0; j < m.n; ++j) {
88                  s << m.at(i, j) << " ";
89              }
90              s << "\n";
91          }
92          return s<< "}\n";
93      }
94
95      double norm1() const { return normImpl<Type::Row>(n, m); }
96      double normInf() const { return normImpl<Type::Col>(m, n); }
97      double normF() const {
98          double res = 0;
99          for (int i = 0; i < m; ++i)
100             for (int j = 0; j < n; ++j)
101                 res += at(i, j) * at(i, j);
102
103         return std::sqrt(res);
104     }
105     bool isStrictlyRowDiagonallyDominant() const {
106         return isStrictlyDiagonallyDominantImpl<Type::Row>();
107     }
108
109     bool isStrictlyColDiagonallyDominant() const {
110         return isStrictlyDiagonallyDominantImpl<Type::Col>();
111     }
112
113 private:
114
115     enum class Type { Row, Col };
```

```
116     template<Type type>
117     bool isStrictlyDiagonallyDominantImpl() const {
118         assert(m == n, "Matrix must be square");
119         for (int i = 0; i < m; ++i) {
120             double val = std::abs(at(i, i));
121             double sum = -val;
122             for (int j = 0; j < m; ++j) {
123                 sum += std::abs(type == Type::Col? at(j, i): at(i, j));
124             }
125             if (sum >= val) {
126                 std::cout << "(" << sum << "," << val << ")";
127                 return false;
128             }
129         }
130         return true;
131     }
132     template<Type type>
133     double normImpl(int sz1, int sz2) const {
134         double max = -1;
135         for (int j = 0; j < sz1; ++j) {
136             double x = 0;
137             for (int i = 0; i < sz2; ++i) {
138                 x += std::abs(type == Type::Col? at(j, i) : at(i, j));
139             }
140             if (x > max) max = x;
141         }
142         return max;
143     }
144 };
145 
146 Mat& add(const Mat& a, const Mat& b, Mat& res) {
147     assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
148     res.setSize(a.m, a.n);
149     for (int i = 0; i < a.m; ++i)
150         for (int j = 0; j < a.n; ++j)
151             res.at(i, j) = a.at(i, j) + b.at(i, j);
152     return res;
153 }
154 
155 Mat& mul(double a, const Mat& b, Mat& res) {
156     res.setSize(b.m, b.n);
157 
158     for (int i = 0; i < res.m; ++i)
159         for (int j = 0; j < res.n; ++j)
160             res.at(i, j) = a * b.at(i, j);
161     return res;
162 }
163 
164 Mat& neg(const Mat& a, Mat& res) {
165     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();
        _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);
}
```

stack.h

```
#pragma once
#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

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

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 << "}";
    }
};
```

list.h

```
#pragma once
#include "utils.h"
#include "ptrRange.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;
    };
    struct It {
        const Node* n;
        constexpr It(const Node* n) : n(n) {}
        constexpr It& operator++() {
            n = n->next;
            return *this;
        }
        constexpr auto& operator*() { return n->val; }
        constexpr bool operator ==(const It& rhs) { return n == rhs.n; }
        constexpr bool operator !=(const It& rhs) { return n != rhs.n; }
    };
```

```
    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;

    constexpr It begin() const { return top; }
    constexpr It end() const { return nullptr; }

    constexpr const T& front() const { return top->val; }
    constexpr T& front() { return top->val; }
    constexpr size_t size() const {
        size_t sz = 0;
        Node* n = top;

        while (n) {
            ++sz;
            n = n->next;
        }
        return sz;
    }

    constexpr void copyTo(T* ptr) const {
        for (auto v : *this) {
            *(ptr++) = v;
        }
    }
    ManagedPtrRange<T> toManagedPtrRange() const {
        auto res = ManagedPtrRange<T>(size());
        copyToMemory(res.first);
        return res;
    }

    ~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);
    }

```

```
78      T& first() { return top->val; }
79      T& last() { return bot->val; }
80      T pop_front() {
81          assert(top, "Empty list");
82          Node* n = top;
83          top = top->next;
84          T res = n->val;
85          delete n;
86          return res;
87      }
88
89      void push_front(T val) {
90          Node *n = new Node{ val, top };
91          top = n;
92      }
93      void push_back(T val) {
94          Node *n = new Node{ val, nullptr };
95          if (top == nullptr) {
96              top = n;
97          }
98          else {
99              bot->next = n;
100
101         }
102         bot = n;
103     }
104
105     template <typename... Args>
106     void emplace_front(Args&&... args) {
107         Node *n = new Node{ T(std::forward<Args>(args)...), top };
108         top = n;
109     }
110     template <typename... Args>
111     void emplace_back(Args&&... args) {
112         Node *n = new Node{ T(std::forward<Args>(args)...), nullptr };
113         if (top == nullptr) {
114             top = n;
115         }
116         else {
117             bot->next = n;
118
119         }
120         bot = n;
121     }
122
123     bool empty() const { return top == nullptr; }
124
125     bool operator== (const List& rhs) {
126         for (auto it = top, rit = rhs.top;
127             it != nullptr;
```

```
128             it = it->next, rit = rit->next) {
129             if (it->val != rit->val) return false;
130         }
131         return true;
132     }
133 
134     static List read(const char* msg) {
135         List q;
136         std::cout << msg << ":\n";
137         int len1 = readSize("n");
138         for (int i = 0; i < len1; ++i) {
139             T s;
140             std::cin >> s;
141             q.push_front(s);
142         }
143         return std::move(q);
144     }
145 
146     friend std::ostream& operator<<(std::ostream& s, const List& st) {
147         if constexpr (std::is_same_v<T, char>) {
148             const Node* it = st.top;
149             while (it != nullptr) {
150                 s << it->val;
151                 it = it->next;
152             }
153             return s;
154         }
155         else {
156             if (st.empty()) return s << "{}";
157             s << "{";
158             const Node* it = st.top;
159             while (it->next != nullptr) {
160                 s << it->val << ", ";
161                 it = it->next;
162             }
163 
164             return s << it->val << "}";
165         }
166     }
167 
168     void remove(T& val) {
169         remove_if([&] (T& t) { return t ==val; });
170     }
171     // removes all elements that satisfy p
172     template<class P>
173     void remove_if(P p) {
174         apply_on(p, [] (Node* n) { delete n; } );
175     }
176     //applies f() on all nodes that satisfy the predicate p()
177     template<class P, class F>
```

```
178     void apply_on(P p, F f) {
179         for (;;) {
180             auto* n = top;
181             if (!n) {
182                 return;
183             }
184             if (!p(n->val)) break;
185             top = n->next;
186             f(n);
187         }
188         auto* prev = top;
189         auto* it = prev->next;
190 
191         while (it) {
192             if (p(it->val)) {
193                 prev->next = it->next;
194                 if (prev->next == nullptr) {
195                     bot = prev;
196                 }
197                 f(it);
198                 it = prev;
199             } else {
200                 prev = it;
201                 it = it->next;
202             }
203         }
204     }
205     // P is a predicate on T
206     template<typename P>
207     Node* find(P p) {
208         auto n = top;
209         for (; n; n = n->next) {
210             if (p(n->val))
211                 return n;
212         }
213         return n;
214     }
215 
216     template<typename P>
217     const Node* find(P p) const {
218         auto n = top;
219         for (; n; n = n->next) {
220             if (p(n->val))
221                 return n;
222         }
223         return n;
224     }
225 
226     Node* findElem(T& e) {
227         return find([&](T& other) { return other == e; });
```

```
    }
    const Node* findElem(T& e) const {
        return find([&](T& other) { return other == e; });
    }
};

// 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;
}
```

doubleList.h

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

#include <utility>

template<typename T>
struct DoubleList {
    struct Node {
        T val;
        Node* next;
        Node* prev;
    };
    struct It {
        const Node* n;
        constexpr It(const Node* n) : n(n) {}
        constexpr It& operator++() { n = n->next; return *this; }
        constexpr It& operator--() { n = n->prev; return *this; }

        constexpr It operator++(int) { auto r = n; ++(*this); return r; }
        constexpr It operator--(int) { auto r = n; --(*this); return r; }
```

```
21
22          constexpr auto& operator*() { return n->val; }
23          constexpr auto& operator->() { return n->val; }
24          constexpr bool operator ==(const It& rhs) { return n == rhs.n; }
25          constexpr bool operator !=(const It& rhs) { return n != rhs.n; }
26          constexpr bool hasNext() const { return n->next; }
27          constexpr bool hasPrev() const { return n->prev; }
28      };
29      Node* top;
30      Node* bot;
31      constexpr DoubleList(Node* top = nullptr, Node* bot = nullptr)
32          : top(top), bot(bot) {}
33      DoubleList(std::initializer_list<T> l) : DoubleList() {
34          for (auto& val : l) push_back(val);
35      }
36      DoubleList(const DoubleList&) = delete;
37      DoubleList& operator=(const DoubleList&) = delete;
38
39      constexpr It begin() const { return top; }
40      constexpr It end() const { return nullptr; }
41
42      ~DoubleList() {
43          while (top) {
44              Node* n = top;
45              top = top->next;
46              delete n;
47          }
48      }
49      DoubleList(DoubleList&& rhs) noexcept
50          : top(std::exchange(rhs.top, nullptr)),
51            bot(std::exchange(rhs.bot, nullptr)) {}
52      DoubleList& operator=(DoubleList&& rhs) noexcept {
53          this->~DoubleList();
54          this->top = std::exchange(rhs.top, nullptr);
55          this->bot = std::exchange(rhs.bot, nullptr);
56      }
57
58      T& first() { return top->val; }
59      T& last() { return bot->val; }
60      T pop_front() {
61          assert(top, "Empty list");
62          Node* n = top;
63          top = top->next;
64          if (top == nullptr) bot = nullptr;
65          top->prev = nullptr;
66          T res = n->val;
67          delete n;
68          return res;
69      }
70
```

```
71      T pop_back() {
72          assert(bot, "Empty list");
73          Node* n = bot;
74          bot = bot->prev;
75          if (bot == nullptr) bot = nullptr;
76          bot->next = nullptr;
77          T res = n->val;
78          delete n;
79          return res;
80      }
81
82      void push_front(T val) {
83          Node *n = new Node{ val, top, nullptr };
84          if (top == nullptr) bot = n;
85          else top->prev = n;
86          top = n;
87      }
88      void push_back(T val) {
89          Node *n = new Node{ val, nullptr, bot };
90          if (top == nullptr) top = n;
91          else bot->next = n;
92          bot = n;
93      }
94      bool empty() const { return top == nullptr; }
95
96      bool operator==(const DoubleList& rhs) {
97          for (auto it = top, rit = rhs.top;
98              it != nullptr;
99              it = it->next, rit = rit->next) {
100             if (it->val != rit->val) return false;
101         }
102         return true;
103     }
104
105     static DoubleList read(const char* msg) {
106         DoubleList q;
107         std::cout << msg << ":\n";
108         int len1 = readSize("n");
109         for (int i = 0; i < len1; ++i) {
110             T s;
111             std::cin >> s;
112             q.push_back(s);
113         }
114         return std::move(q);
115     }
116
117     friend std::ostream& operator<<(std::ostream& s,
118                                     const DoubleList& st) {
119         if constexpr (std::is_same_v<T, char>) {
120             const Node* it = st.top;
```

```
121             while (it != nullptr) {
122                 s << it->val;
123                 it = it->next;
124             }
125             return s;
126         }
127         else {
128             if (st.empty()) return s << "{}";
129             s << "{";
130             const Node* it = st.top;
131             while (it->next != nullptr) {
132                 s << it->val << ", ";
133                 it = it->next;
134             }
135             return s << it->val << "}";
136         }
137     }
138 
139     void remove(T& val) {
140         remove_if([&] (T& t) { return t == val; });
141     }
142     void remove(Node* n) {
143         if (!n->prev) { std::cout << "FRONT\n"; pop_front(); return; }
144         if (!n->next) { std::cout << "BACK\n"; pop_back(); return; }
145         n->prev->next = n->next;
146         n->next->prev = n->prev;
147     }
148     // removes all elements that satisfy p
149     template<class P>
150     void remove_if(P p) {
151         apply_on(p, [] (Node* n) { delete n; } );
152     }
153     //applies f() on all nodes that satisfy the predicate p()
154     template<class P, class F>
155     void apply_on(P p, F f) {
156         for (;;) {
157             auto* n = top;
158             if (!n) {
159                 return;
160             }
161             if (!p(n->val)) break;
162             top = n->next;
163             f(n);
164         }
165         auto* prev = top;
166         auto* it = prev->next;
167 
168         while (it) {
169             if (p(it->val)) {
170                 prev->next = it->next;
```

```
                if (prev->next == nullptr) {
                    bot = prev;
                }
                f(it);
                it = prev;
            } else {
                prev = it;
                it = it->next;
            }
        }
    }
    // P is a predicate on T
    template<typename P>
    Node* find(P p) {
        auto n = top;
        for (; n; n = n->next) {
            if (p(n->val))
                return n;
        }
        return n;
    }
    void insert(Node* n, T val) {
        if (n == bot) { push_back(val); return; }
        Node* newN = new Node{ val, n->next, n};
        n->next->prev = newN;
        n->next = newN;
    }
    size_t size() const {
        auto it = top;
        size_t res = 0;
        while (it) { it = it->next; ++res; }
        return res;
    }
};
```

graphEdgeList.h

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

#include <iostream>
#include <fstream>

struct Graph_EdgeList {
    using Vertex = unsigned;
    struct Edge {
        Vertex a, b;
        constexpr Edge(Vertex a, Vertex b) :a(a), b(b) {}
        friend std::ostream& operator<< (std::ostream& s, Edge e) {
```

```
            return s << "(" << e.a << ", "<< e.b << ")";
        }

        friend std::istream& operator>> (std::istream& s, Edge& e) {
            return s >> e.a >>e.b;
        }
        constexpr bool operator ==(Edge rhs) const {
            return (rhs.a == a && rhs.b == b) ||
                (rhs.b == a && rhs.a == b);
        }
    };
    List<Edge> edges;
    size_t nodeSize;

    template <typename Path>
    static Graph_EdgeList fromFile(Path path) {
        std::ifstream f(path);
        size_t sz = 0;
        f >> sz;
        Graph_EdgeList g { {}, sz };
        Vertex a, b;
        while (f >> a >> b) { g.addEdge(a, b); }
        return g;
    }
    void addEdge(Vertex a, Vertex b) { edges.emplace_front(a, b); }
    void addEdge(Edge e) { edges.push_front(e); }

    friend std::ostream& operator<<(std::ostream& s,
                                    const Graph_EdgeList& g) {
        return s << g.edges;
    }
    constexpr bool hasEdge(Edge e) const {
        return edges.findElem(e) != nullptr;
    }

    constexpr bool hasEdge(Vertex a, Vertex b) const {
        return hasEdge({a, b});
    }
};
```

graphAdjacencyList.h

```
#pragma once

#include "utils.h"
#include "ptrRange.h"
#include "list.h"

#include <iostream>
#include <fstream>
```

```cpp
#include <algorithm>//std::count
#include <alloca.h>


struct Graph_AdjacencyList {
    using Vertex = unsigned;
private:

    Vertex** adjacencyList;
    Vertex* adjacencyData;

    size_t vertSize;
    size_t edgeSize;
    using Graph = Graph_AdjacencyList;
public:
    struct Iterator {
        Vertex** p;
        constexpr Iterator(Vertex** p) : p(p) {}

        constexpr Iterator operator+(int x) const { return p + x; }
        constexpr Iterator operator-(int x) const { return p + x; }
        constexpr Iterator& operator++() { ++p; return *this; }
        constexpr Iterator& operator--() { ++p; return *this; }

        constexpr Iterator operator++(int) { auto r = p; ++p; return r; }
        constexpr Iterator operator--(int) { auto r = p; --p; return r; }

        constexpr auto operator*() const {
            return PtrRange<const Vertex>(*p, *(p+1));
        }
        //constexpr auto& operator->() { return ; }
        constexpr bool operator ==(Iterator o) const { return p == o.p; }
        constexpr bool operator !=(Iterator o) const { return p != o.p; }
    };
    constexpr auto begin() const { return Iterator(adjacencyList); }
    constexpr auto end() const { return Iterator(adjacencyList+vertSize); }

    constexpr PtrRange<const Vertex> operator[](Vertex v) const {
        return *(begin() + v);
    }
    size_t vertCount() const { return vertSize; }
    size_t edgeCount() const { return edgeSize; }

private:
    Graph_AdjacencyList(Vertex** adjacencyList, Vertex* adjacencyData,
                        size_t vertSize, size_t edgeSize)
        : adjacencyList(adjacencyList), adjacencyData(adjacencyData),
          vertSize(vertSize), edgeSize(edgeSize) {
        adjacencyData = new Vertex[edgeSize*2];
    }
```

```
59  public:
60      Graph_AdjacencyList(const Graph&) = delete;
61      Graph_AdjacencyList(Graph&& rhs)
62          : adjacencyList(std::exchange(rhs.adjacencyList, nullptr)),
63            adjacencyData(std::exchange(rhs.adjacencyData, nullptr)),
64            vertSize(rhs.vertSize), edgeSize(rhs.edgeSize) {}
65      Graph& operator=(const Graph&) = delete;
66      Graph& operator=(Graph&& rhs) {
67          this->~Graph_AdjacencyList();
68          vertSize = rhs.vertSize;
69          edgeSize = rhs.edgeSize;
70          adjacencyList = std::exchange(rhs.adjacencyList, nullptr);
71          adjacencyData = std::exchange(rhs.adjacencyData, nullptr);
72          return *this;
73      }
74      ~Graph_AdjacencyList() {
75          delete[] adjacencyData;
76          delete[] adjacencyList;
77      }
78
79      template <typename Path>
80      static Graph_AdjacencyList fromFile(Path path) {
81          std::ifstream f(path);
82          size_t vertSize = 0;
83          f >> vertSize;
84
85          // std::cout << "Verts: " << vertSize << "\n";
86          static_assert(sizeof(intptr_t) == sizeof(Vertex*));
87          PtrRange<size_t> counts((size_t*)alloca(sizeof(size_t) * vertSize),
88                                  vertSize);
89          for (auto& it:counts) it = 0;
90
91          size_t edgeSize = 0;
92          Vertex v, w;
93          while (f >> v >> w) {
94              ++edgeSize;
95              // std::cout << "L" << v << " " << w << "\n";
96
97              ++counts[v];
98              ++counts[w];
99          }
100
101         std::cout << "edgeSize: " << edgeSize << "\n";
102         // std::cout << "counts: " << counts << "\n";
103         Vertex** list = new Vertex*[vertSize+1];
104
105         for (auto it = counts.begin()+1, end = counts.end()-1;
106              it < end; ++it) {
107             *it += *(it-1);
108         }
```

```
        // std::cout << "cumsum: " << counts << "\n";

        PtrRange<Vertex*> countsPtr((Vertex**) counts.begin(),
                                    (Vertex**) counts.end());
        Vertex* data = new Vertex[edgeSize*2];
        for (size_t i = vertSize-1; i >  0; --i) {
            countsPtr[i] = list[i] = data + counts[i-1];
        }
        countsPtr[0] = list[0] = data;
        list[vertSize] = data+(edgeSize*2);

        //std::cout << "thing: " << countsPtr << "\n";

        f.clear();
        f.seekg(0);
        f >> vertSize;
        Vertex a, b;
        while (f>> a>> b) {
            *(countsPtr[a]++) = b;
            *(countsPtr[b]++) = a;
        }
        return Graph_AdjacencyList(list, data, vertSize, edgeSize);
    }

    friend std::ostream& operator<<(std::ostream& s,
                                    const Graph_AdjacencyList& g) {
        for (size_t i = 0; i < g.vertSize; ++i) {
            s <<  "["<< i << "] = " << g[i] << "\n";
        }
        return s;
    }
    constexpr bool hasEdge(Vertex a, Vertex b) const {
        return (*this)[a].contains(b);
    }
};
```

ptrRange.h

```
#pragma once
#include <iostream>
#include <utility>
#include <iomanip>

template<typename T>
struct PtrRange {
    T* first;
    T* last;
    constexpr PtrRange(T* first, T* last)
        : first(first), last(last){}

```

```
    constexpr PtrRange(T* first, size_t sz)
        : first(first), last(first+sz){}
    constexpr T* begin() { return first; }
    constexpr T* end() { return last; }

    constexpr const T* begin() const { return first; }
    constexpr const T* end() const { return last; }

    constexpr T& front() { return *first; }
    constexpr T& back() { return *(last-1); }
    constexpr const T& front() const { return *first; }
    constexpr const T& back() const { return *(last-1); }


    friend std::ostream& operator<<(std::ostream& s, PtrRange v) {
        s << "(";
        T* it = v.first;
        for (T* end = v.last - 1; it < end; ++it)
            s << *it << ", ";

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

        return s << ")";
    }
    constexpr const T& operator[](size_t i) const { return first[i]; }
    constexpr T& operator[](size_t i) { return first[i]; }

    constexpr size_t size() const { return last - first; }

    constexpr T* find(const T& val) {
        for (auto it = first; it != last; ++it) {
            if (*it == val) return it;
        }
        return nullptr;
    }

    constexpr T* min() {
        auto smallest = first;
        for (auto it = first+1; it != last; ++it) {
            if (*it < *smallest)
                smallest = it;
        }
        return smallest;
    }

    // P is a predicate on T
    template<typename P>
    constexpr T* find_if(P p) {
        for (auto it = first; it != last; ++it) {
            if (p(it->val)) return it;
```

```
        }
        return nullptr;
    }

    template<typename P>
    constexpr T* find_neighbouring_if(P p) {
        for (auto it = first; it != (last-1); ++it) {
            if (p(it->val, (it+1)->val)) return it;
        }
        return nullptr;
    }

    template<typename P>
    constexpr T* find_neighbouring(const T& a, const T& b) {
        for (auto it = first; it != (last-1); ++it) {
            if (*it == a && *(it+1) == b) return it;
        }
        return nullptr;
    }
    constexpr bool contains(const T& val) const { return find(val) != nullptr; }

    template<typename P>
    constexpr T* contains_if(P p) const { return find(p) != nullptr; }

    template<typename P>
    constexpr T* contains_neighbouring_if(P p) const {
        return find_neighbouring_if(p) != nullptr;
    }
};

template<typename T>
struct ManagedPtrRange : public PtrRange<T> {
    ManagedPtrRange(size_t sz)
        : PtrRange<T>(new T[sz], sz) {}

    ManagedPtrRange(const ManagedPtrRange&) = delete;
    ManagedPtrRange(ManagedPtrRange&& rhs)
        : PtrRange<T>(std::exchange(rhs.first, nullptr),
                      std::exchange(rhs.last, nullptr)) {}
    ManagedPtrRange& operator=(const ManagedPtrRange&) = delete;
    ManagedPtrRange& operator=(ManagedPtrRange&& rhs) {
        this->~ManagedPtrRange();
        this->first = std::exchange(rhs.fist, nullptr);
        this->last = std::exchange(rhs.last, nullptr);
        return *this;
    }
    ~ManagedPtrRange() { delete[] this->first; }
};

template<typename T>
```

```cpp
struct MatrixPtrRange : public PtrRange<T> {
    size_t rows, cols;
    MatrixPtrRange(T* first, size_t rows, size_t cols)
        : PtrRange<T>(first, rows*cols), rows(rows), cols(cols) {}
protected:
    MatrixPtrRange(T* first, T* last, size_t rows, size_t cols)
        : PtrRange<T>(first, last), rows(rows), cols(cols) {}
public:
    constexpr T& operator()(size_t i, size_t j) {
        return this->first[cols*i + j];
    }
    constexpr const T& operator()(size_t i, size_t j) const {
        return this->first[cols*i + j];
    }
    friend std::ostream& operator<<(std::ostream& s, MatrixPtrRange p) {
        auto it = p.begin();
        auto end = p.end();
        auto endl = it + p.cols;
        auto w = s.width();
        for (;endl <= end; endl += p.cols) {
            for (;it < endl; ++it)
                s << std::setw(w) << *it << " ";
            s << "\n";
        }
        return s;
    }
};

template<typename T>
struct ManagedMatrixPtrRange : public MatrixPtrRange<T> {
    ManagedMatrixPtrRange(size_t rows, size_t cols)
        : MatrixPtrRange<T>(new T[rows*cols], rows, cols) {}

    ManagedMatrixPtrRange(const ManagedMatrixPtrRange<T>& rhs) :
        ManagedMatrixPtrRange(static_cast<const MatrixPtrRange<T>&>(rhs)) {}
    ManagedMatrixPtrRange(const MatrixPtrRange<T>& rhs)
        : ManagedMatrixPtrRange(rhs.rows, rhs.cols) {
        auto itL = this->begin();
        auto itR = rhs.begin();
        auto endR = rhs.end();
        for (; itR != endR; ++itR, ++itL)
            *itL = *itR;
    }
    ManagedMatrixPtrRange(ManagedMatrixPtrRange&& rhs)
        : MatrixPtrRange<T>(std::exchange(rhs.first, nullptr),
                            std::exchange(rhs.last, nullptr),
                            rhs.rows, rhs.cols) {}

    ManagedMatrixPtrRange& operator=(const ManagedMatrixPtrRange<T>& rhs) {
        *this = static_cast<const MatrixPtrRange<T>&>(rhs);
```

```
163     }
164
165     ManagedMatrixPtrRange& operator=(const MatrixPtrRange<T>& rhs) {
166         if (this->size() >= rhs.size()) {
167             this->last = this->first + rhs.size();
168             this->rows = rhs.rows;
169             this->cols = rhs.cols;
170         }
171         else {
172             this->~ManagedMatrixPtrRange();
173             new (this) ManagedMatrixPtrRange(rhs.rows, rhs.cols);
174         }
175         auto itL = this->begin();
176         auto itR = rhs.begin();
177         auto endR = rhs.end();
178         for (; itR != endR; ++itR, ++itL)
179             *itL = *itR;
180     }
181     ManagedMatrixPtrRange& operator=(ManagedMatrixPtrRange&& rhs) {
182         this->~ManagedMatrixPtrRange();
183         this->first = std::exchange(rhs.first, nullptr);
184         this->last = std::exchange(rhs.last, nullptr);
185         return *this;
186     }
187     ~ManagedMatrixPtrRange() { delete[] this->first; }
188 };
189
190 //template<typename T>
191 //using Vector = ManagedPtrRange<T>;
192
193 //template<typename T>
194 //using Matrix = ManagedMatrixPtrRange<T>;
195
196 template<typename T>
197 struct StaticVector : public ManagedPtrRange<T> {
198     T* allocEnd;
199     constexpr size_t capacity() const { return allocEnd - this->first; }
200     StaticVector(size_t capacity, size_t sz)
201         : ManagedPtrRange<T>(capacity) {
202         allocEnd = this->last;
203         this->last = this->first+sz;
204     }
205     StaticVector(size_t capacity) : StaticVector(capacity, 0) {}
206
207     void push_back(const T& val) {
208         assert(this->last != allocEnd, "Overflow");
209         *(this->last++) = val;
210     }
211     void insert(T* pos, const T& val) {
212         assert(this->last != allocEnd, "Overflow");
```

```
        for (auto it = this->last-1; it != pos; --it)
            *it = *(it-1);
        ++this->last;
        *pos = val;
    }
    void remove(T* pos) {
        assert(this->last != this->first, "Empty vector");
        for (auto it = pos; it != this->last; ++it)
            *it = *(it+1);
        --this->last;
    }
};
```

dirGraphMat.h

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

#include <iostream>
#include <fstream>
#include <utility>


struct DirGraph_Mat {
    using Graph = DirGraph_Mat;
    using Vertex = unsigned;
    bool* mat;
    size_t size;

    DirGraph_Mat(size_t size) : mat(new bool[size*size]), size(size) {}
    DirGraph_Mat(const Graph&) = delete;
    DirGraph_Mat(Graph&& rhs)
        : mat(std::exchange(rhs.mat, nullptr)), size(rhs.size) {}
    Graph& operator=(const Graph&) = delete;
    Graph& operator=(Graph&& rhs) {
        this->~DirGraph_Mat();
        mat = std::exchange(rhs.mat, nullptr);
        size = rhs.size;
        return *this;
    }
    ~DirGraph_Mat() {
        delete[] mat;
    }

    constexpr auto operator[] (Vertex v) const {
        return PtrRange<const bool>(mat+size*v, size);
    }

    template <typename Path>
```

```
    static Graph fromFile(Path path) {
        std::ifstream f(path);
        size_t sz = 0;
        f >> sz;
        Graph g(sz);
        Vertex a, b;
        while (f >> a >> b) { g.addArc(a, b); }
        return g;
    }
    constexpr void addArc(Vertex a, Vertex b) {
        mat[a * size + b] = true;
    }
    constexpr bool hasArc(Vertex a, Vertex b) const {
        return mat[a * size + b];
    }
    void printMatrix() const {
        bool* it = mat;
        bool* end = mat + size*size;
        bool* endl = mat + size;
        for (;endl < end; endl += size) {
            for (;it < endl; ++it)
                std::cout << *it << " ";
            std::cout << "\n";
        }
    }
};
```

dirGraphAdjacencyList.h

```
#pragma once

#include "utils.h"
#include "ptrRange.h"
#include "list.h"

#include <iostream>
#include <fstream>
#include <algorithm>//std::count
#include <alloca.h>


struct DirGraph_AdjacencyList {
    using Vertex = unsigned;
private:

    Vertex** adjacencyList;
    Vertex* adjacencyData;

    size_t vertSize;
    size_t arcSize;
```

```
22      using Graph = DirGraph_AdjacencyList;
23  public:
24      struct Iterator {
25          Vertex** p;
26          constexpr Iterator(Vertex** p) : p(p) {}
27
28          constexpr Iterator operator+(int x) const { return p + x; }
29          constexpr Iterator operator-(int x) const { return p + x; }
30          constexpr Iterator& operator++() { ++p; return *this; }
31          constexpr Iterator& operator--() { ++p; return *this; }
32
33          constexpr Iterator operator++(int) { auto r = p; ++p; return r; }
34          constexpr Iterator operator--(int) { auto r = p; --p; return r; }
35
36          constexpr auto operator*() const {
37              return PtrRange<const Vertex>(*p, *(p+1));
38          }
39          //constexpr auto& operator->() { return ; }
40          constexpr bool operator ==(Iterator o) const { return p == o.p; }
41          constexpr bool operator !=(Iterator o) const { return p != o.p; }
42      };
43      constexpr auto begin() const { return Iterator(adjacencyList); }
44      constexpr auto end() const { return Iterator(adjacencyList+vertSize); }
45
46      constexpr PtrRange<const Vertex> operator[](Vertex v) const {
47          return *(begin() + v);
48      }
49      size_t vertCount() const { return vertSize; }
50      size_t arcCount() const { return arcSize; }
51
52  private:
53      DirGraph_AdjacencyList(Vertex** adjacencyList, Vertex* adjacencyData,
54                             size_t vertSize, size_t edgeSize)
55          : adjacencyList(adjacencyList), adjacencyData(adjacencyData),
56            vertSize(vertSize), arcSize(edgeSize) {
57          adjacencyData = new Vertex[edgeSize];
58      }
59  public:
60      DirGraph_AdjacencyList(const Graph&) = delete;
61      DirGraph_AdjacencyList(Graph&& rhs)
62          : adjacencyList(std::exchange(rhs.adjacencyList, nullptr)),
63            adjacencyData(std::exchange(rhs.adjacencyData, nullptr)),
64            vertSize(rhs.vertSize), arcSize(rhs.arcSize) {}
65      Graph& operator=(const Graph&) = delete;
66      Graph& operator=(Graph&& rhs) {
67          this->~DirGraph_AdjacencyList();
68          vertSize = rhs.vertSize;
69          arcSize = rhs.arcSize;
70          adjacencyList = std::exchange(rhs.adjacencyList, nullptr);
71          adjacencyData = std::exchange(rhs.adjacencyData, nullptr);
```

```
72          return *this;
73      }
74      ~DirGraph_AdjacencyList() {
75          delete[] adjacencyData;
76          delete[] adjacencyList;
77      }
78
79      template <typename Path>
80      static Graph fromFile(Path path) {
81          std::ifstream f(path);
82          size_t vertSize = 0;
83          f >> vertSize;
84
85          // std::cout << "Verts: " << vertSize << "\n";
86          static_assert(sizeof(intptr_t) == sizeof(Vertex*));
87          PtrRange<size_t> counts((size_t*)alloca(sizeof(size_t) * vertSize),
88                                  vertSize);
89          for (auto& it:counts) it = 0;
90
91          size_t arcSize = 0;
92          Vertex v, w;
93          while (f >> v >> w) {
94              ++arcSize;
95              // std::cout << "L" << v << " " << w << "\n";
96              ++counts[v];
97          }
98
99          std::cout << "arcSize: " << arcSize << "\n";
100         // std::cout << "counts: " << counts << "\n";
101         Vertex** list = new Vertex*[vertSize+1];
102
103         for (auto it = counts.begin()+1, end = counts.end()-1;
104              it < end; ++it) {
105             *it += *(it-1);
106         }
107         // std::cout << "cumsum: " << counts << "\n";
108
109         PtrRange<Vertex*> countsPtr((Vertex**) counts.begin(),
110                                     (Vertex**) counts.end());
111         Vertex* data = new Vertex[arcSize*2];
112         for (size_t i = vertSize-1; i >  0; --i) {
113             countsPtr[i] = list[i] = data + counts[i-1];
114         }
115         countsPtr[0] = list[0] = data;
116         list[vertSize] = data+(arcSize);
117
118         //std::cout << "thing: " << countsPtr << "\n";
119
120         f.clear();
121         f.seekg(0);
```

```
        f >> vertSize;
        Vertex a, b;
        while (f>> a>> b) {
            *(countsPtr[a]++) = b;
        }
        return Graph(list, data, vertSize, arcSize);
    }

    friend std::ostream& operator<<(std::ostream& s,
                                    const Graph& g) {
        for (size_t i = 0; i < g.vertSize; ++i) {
            s <<  "["<< i << "] = " << g[i] << "\n";
        }
        return s;
    }
    /*
    constexpr bool hasArc(Vertex a, Vertex b) const {
        return (*this)[a].contains(b);
    }*/
};
```

weightedDirGraphMat.h

```
#include "utils.h"
#include "ptrRange.h"

#include <iostream>
#include <limits>
#include <utility>

struct WeightedDirGraph_Mat {
    using Graph = WeightedDirGraph_Mat;
    using Vertex = unsigned;
    using Weight = double;

    using Matrix = ManagedMatrixPtrRange<Weight>;
    static_assert(std::numeric_limits<Weight>::has_infinity);
    static constexpr Weight inf = std::numeric_limits<Weight>::infinity();
    static_assert(inf + 1 == inf);
    static_assert(inf - 1 == inf);
    static_assert(1 < inf);

    Matrix mat;
    constexpr size_t size() const { return mat.cols; }
    WeightedDirGraph_Mat(size_t size): mat(size, size) {
        for (size_t i = 0; i < size*size; ++i) {
            mat[i] = inf;
        }
        for (size_t i = 0; i < size; ++i)
            addArc(i, i, 0);
```

```
    }

    static WeightedDirGraph_Mat fromFile(const char* path) {
        std::ifstream f(path);
        size_t sz = 0;
        f >> sz;
        Graph g(sz);
        Vertex a, b;
        Weight w;

        while (f >> a >> b >> w) { g.addArc(a, b, w); }
        return g;
    }

    friend std::ostream& operator<< (std::ostream& s, const Graph& g) {
        return s << g.mat;
    }

    constexpr void addArc(Vertex a, Vertex b, Weight w) {
        mat(a, b) = w;
    }
    constexpr Weight arcWeight(Vertex a, Vertex b) const {
        return mat(a, b);
    }
};
```

Alocarea dinamică a memoriei. Tipuri specifice.

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.

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

```

[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 $|a_{ii}| > \sum\limits_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|$, pentru orice $i = 1, ..., n$.

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

```
    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 {
```

```
78          return isStrictlyDiagonallyDominantImpl<Type::Col>();
79      }
80  };
81
82  Mat& add(const Mat& a, const Mat& b, Mat& res) {
83      assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
84      res.setSize(a.m, a.n);
85      for (int i = 0; i < a.m; ++i)
86          for (int j = 0; j < a.n; ++j)
87              res.at(i, j) = a.at(i, j) + b.at(i, j);
88      return res;
89  }
90  Mat& mul(double a, const Mat& b, Mat& res) {
91      res.setSize(b.m, b.n);
92
93      for (int i = 0; i < res.m; ++i)
94          for (int j = 0; j < res.n; ++j)
95              res.at(i, j) = a * b.at(i, j);
96      return res;
97  }
98  Mat& neg(const Mat& a, Mat& res) { return mul(-1, a, res); }
99  Mat& sub(const Mat& a, const Mat& b, Mat& res) {
100     return add(a, neg(b, res), res);
101 }
102 Mat& mul(const Mat& a, const Mat& b, Mat& res) {
103     assert(a.n == b.m, "Sizes don't match, can't multiply");
104     res.setSize(a.m, b.n);
105
106     for (int i = 0; i < res.m; ++i)
107         for (int j = 0; j < res.n; ++j) {
108             res.at(i, j) = 0;
109             for (int k = 0; k < a.n; ++k)
110                 res.at(i, j) += a.at(i, k) * b.at(k, j);
111         }
112     return res;
113 }
114 Mat& trans(const Mat& a, Mat& res) {
115     assert(a.data != res.data, "Can't calculate the transpose inplace");
116     res.setSize(a.n, a.m);
117
118     for (int i = 0; i < res.m; ++i)
119         for (int j = 0; j < res.n; ++j)
120             res.at(i, j) = a.at(j, i);
121     return res;
122 }
```

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));
}
```

```
100 double Vec::norm() const {
101     return ::norm(*this);
102 }
```

Tablouri

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.

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

```cpp
#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;
    Vec b;

```

```
30      System(int n, int m) : A(n, m), b(n) {}
31
32      System(std::initializer_list<std::initializer_list<double>> A,
33             std::initializer_list<double> b) : A(A), b(b) {
34             assert(std::size_t(this->A.m) == this->b.size(),
35                    "sizes don't match");
36      }
37
38      friend std::ostream& operator<<(std::ostream& s, const System& sys) {
39          s << "System " << sys.A.m << "x" << sys.A.n << ": \n";
40          auto& A = sys.A;
41          for (int i = 0; i < A.m; ++i) {
42              s << "{";
43              for (int j = 0; j < A.n; ++j) {
44                  //showpos shows a '+' in front of positive numbers
45                  if (std::abs(A.at(i, j)) > eps)
46                      s << std::showpos << A.at(i, j)
47                        << std::noshowpos << "*x" << (j+1) << " ";
48              }
49              s<< "= " << sys.b[i] << "\n";
50          }
51
52          return s;
53      }
54
55      // L_i += f * L_j
56      void addLines(int i, double f, int j) {
57          for (int k = 0; k < A.n; ++k) {
58              A.at(i, k) += f * A.at(j, k);
59          }
60          b[i] += f * b[j];
61      }
62
63      // L_i *= f
64      void multiplyLine(int i, double f) {
65          for (int k = 0; k < A.n; ++k) {
66              A.at(i, k) *= f;
67          }
68          b[i] *= f;
69      }
70
71      Vec solveTriangulated() {
72          for (int i = A.m-1; i > 0; --i) {
73              addLines(i-1, -A.at(i-1, i), i);
74              multiplyLine(i, 1 / A.at(i, i));
75          }
76          return b;
77      }
78
79      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

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.

```
#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;
}
```

11. Creați o listă liniară simplu înlănțuită în nodurile căreia sunt memorate numere naturale. Separați 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 cantității unui produs, în cazul vânzării), calculul valorii stocului la un moment dat, listare stoc.

stock.h

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

```cpp
#include "list.h"

#include <iostream>
#include <iomanip>

constexpr size_t precision = 2;
using FP = FixedPoint<precision>;

template<class T>
constexpr void setMax(T& a, T b) {
    if (b > a) a = b;
}
struct Product {
    std::string name;
    FP quantity;
    std::string unit;
    FP unitPrice;
    constexpr auto totalPrice() const {
        return quantity * unitPrice;
    }
};

struct Stock {
    List<Product> products;

    //return value: was the operation was successful
    bool add(Product p) {
        auto* res = products.find(
            [&] (Product& other) { return other.name == p.name; });
        if (res != nullptr) {
            auto& v = res->val;
            std::cout << "Product '"<< p.name << "' already exists.\n"
                "Resuplying.\n";
            if (v.unit != p.unit) {
                std::cout << "Units did not match ('"
                          << v.unit <<"' != '" <<p.unit << "')\n";
                return false;
            }
            v.unitPrice +=(v.quantity*v.unitPrice + p.quantity*p.unitPrice)
                / (v.quantity +p.quantity);
            v.quantity += p.quantity;
        } else {
            products.push_front(p);
        }
        return true;
    }
    //retun value: was the operation was successful
    bool resupply(const std::string& name, FP quantity) {
        auto* res = products.find(
            [&] (Product& other) { return other.name == name; });
```

```
53          if (res == nullptr){
54              std::cout << "Product '" << name << "' not found.\n";
55              return false;
56          }
57          res->val.quantity += quantity;
58          return true;
59      }
60      //retun value: was the operation was successful
61      bool sell(const std::string& name, FP quantity) {
62          auto* res = products.find(
63              [&] (Product& other) { return other.name == name; });
64          if (res == nullptr) {
65              std::cout << "Product '" << name << "' not found.\n";
66              return false;
67          }
68          auto& v = res->val;
69          if (quantity > v.quantity) {
70              std::cout << "Quantity too high. Can't sell. (" << quantity
71                        << " > " << v.quantity << ")\n";
72              return false;
73          }
74          v.quantity -= quantity;
75          return true;
76      }
77      constexpr FP value() const {
78          FP total = 0;
79          for (auto& p : products)
80              total += p.totalPrice();
81          return total;
82      }
83      void print() const {
84          if (products.empty()) {
85              std::cout << "No products\n";
86              return;
87          }
88          auto ph = PrintHelper(*this);
89          ph.printHeader();
90          FP total = 0;
91          for (auto& p : products) {
92              ph.printProd(p);
93              total += p.totalPrice();
94          }
95          ph.printFooter(total);
96      }
97  private:
98
99      class PrintHelper {
100         static constexpr std::string_view fields[] = {
101             "Name", "Quantity", "Unit Price", "Total Price"
102         };
```

```
103         struct Longest {
104             size_t name = fields[0].size();
105             size_t quantity = 0;// fields[1].size();
106             size_t unit = 0;
107             size_t unitPrice = fields[2].size();
108             size_t totalPrice = fields[3].size();
109         } longest;
110     public:
111         constexpr PrintHelper(const Stock& s) {
112             auto& l = longest;
113             for (auto& p : s.products) {
114                 setMax(l.name, p.name.size());
115                 setMax(l.quantity, p.quantity.textLen());
116                 setMax(l.unit, p.unit.size());
117                 setMax(l.unitPrice, p.unitPrice.textLen());
118                 setMax(l.totalPrice, p.totalPrice().textLen());
119             }
120 
121             longest.quantity = std::max(fields[1].size() - longest.unit -1,
122                                         longest.quantity);
123         }
124         void printProd(const Product& p) const {
125             std::cout << "| " << std::left;
126             printPadded(longest.name, p.name);
127             std::cout << " | " << std::right;
128             printPadded(longest.quantity, p.quantity);
129             std::cout << " " << std::left;
130             printPadded(longest.unit, p.unit);
131             std::cout << " | " << std::right;
132             printPadded(longest.unitPrice, p.unitPrice);
133             std::cout << " | ";
134             printPadded(longest.totalPrice, p.totalPrice());
135             std::cout << " |\n";
136         }
137         void printHeader() const {
138             printLine();
139             std::cout << "| ";
140             printCentered(longest.name, fields[0]);
141             std::cout << " | ";
142             printCentered(longest.quantity + 1 + longest.unit, fields[1]);
143             std::cout << " | ";
144             printCentered(longest.unitPrice, fields[2]);
145             std::cout << " | ";
146             printCentered(longest.totalPrice, fields[3]);
147             std::cout << " |\n";
148             printLine();
149         }
150         void printFooter(FixedPoint<precision> totalPrice) const {
151             printLine();
152             std::cout << "| ";
```

```
            printPadded(longest.name, "");
            std::cout << " | ";
            printPadded(longest.quantity + 1 + longest.unit, "");
            std::cout << " | ";
            printPadded(longest.unitPrice, "");
            std::cout << " | "<<std::right;
            printPadded(longest.totalPrice, totalPrice);
            std::cout << " |\n";
            printLine();
        }

    private:
        template<typename T>
        static void printPadded(size_t len, T v) {
            std::cout << std::setw(len) << v;
        }
        static void printCentered(size_t len, std::string_view s) {
            //we can safely assume that len >= s.len();
            int total = len - s.size();
            int left = total / 2;
            int right = total - left;
            hline(left, ' ');
            std::cout << std::setw(0) << s;
            hline(right, ' ');
        }

        static void hline(int len, char c) {
            while (--len >= 0)
                std::cout << c;
        }

        void printLine() const {
            std::cout << "+";
            hline(longest.name+2, '-');
            std::cout << "+";
            hline(longest.quantity + 3 + longest.unit, '-');
            std::cout << "+";
            hline(longest.unitPrice+2, '-');
            std::cout << "+";
            hline(longest.totalPrice+2, '-');
            std::cout << "+\n";
        }
    };
};
```

stock.cpp

```
#include "stock.h"

#include "inputHelper.h"
```

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

#include <iostream>
#include <iomanip>

struct Command {
    char shortName;
    std::string_view name;
    std::string_view args;
    bool (*f)(Stock& s, MultiInputHelper& ih);//returns true if should exit
    std::string_view description;
    void print() const {
        std::cout << "  " << std::setw(10) << std::left << name
                  << " - " << shortName << " "
                  << std::setw(25) << args << " - " << description << "\n";
    }
};
using IH = MultiInputHelper;

void printHelp();
bool printHelp(Stock&, IH&) {
    printHelp();
    return true;
}
bool quit(Stock&, IH&) { return false; }
bool print(Stock& s, IH&) {
    s.print();
    return true;
}

bool add(Stock& s, IH& ih) {
    Product p;
    ih.readName(p.name, "name");
    ih.readFP(p.quantity, "quantity");
    ih.readString(p.unit, "unit");
    ih.readFP(p.unitPrice, "unit price");
    s.add(p);
    return true;
}

bool sell(Stock& s, IH& ih) {
    std::string name;
    FP quantity;
    ih.readName(name, "name");
    ih.readFP(quantity, "quantity");

    s.sell(name, quantity);
    return true;
}
```

```
54
55  bool resupply(Stock& s, IH& ih) {
56      std::string name;
57      FP quantity;
58      ih.readName(name, "name");
59      ih.readFP(quantity, "quantity");
60
61      s.resupply(name, quantity);
62      return true;
63  }
64
65  bool value(Stock& s, IH&) {
66      std::cout << "Total stock value: " << s.value() << "\n";
67      return true;
68  }
69  bool init(Stock& s, IH& ih) {
70      std::string str;
71      for (;;) {
72          ih.getLine("+ ");
73          add(s, ih);
74          ih.getLine("Add more products (y/N)? ");
75          if (tolower(ih.readChar()) != 'y') break;
76      }
77      return true;
78  }
79  constexpr Command cmds[] = {
80      { 'H', "help", "", printHelp, "Show help" },
81      { 'A', "add", "name quantity unit price", add, "Add product" },
82      { 'I', "init", "", init, "Add multiple elements" },
83      { 'S', "sell", "name quantity", sell, "Sell product" },
84      { 'R', "resupply", "name quantity", resupply, "Resupply product" },
85      { 'V', "value", "", value, "Print stock total value" },
86      { 'P', "print", "", print, "Print a table of products" },
87      { 'Q', "quit", "", quit, "Quit the program" },
88  };
89  void printHelp() {
90      std::cout << "\nFormat of commands: \n"
91                   "name - shortName args - description\n";
92
93      for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i)
94          cmds[i].print();
95  }
96  bool eval(Stock& stock, MultiInputHelper& ih) {
97      ih.getLine("> ");
98      auto s = ih.readStringView("> ");
99      if (s.size() == 1) {
100         for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i) {
101             if (cmds[i].shortName == toupper(s[0]))
102                 return cmds[i].f(stock, ih);
103         }
```

```
    }
    for (size_t i = 0; i < (sizeof(cmds) /sizeof(cmds[0])); ++i) {
        if (cmds[i].name == s) return cmds[i].f(stock, ih);
    }
    std::cout << "Invalid command!\n";
    printHelp();
    std::cout << "!";
    return true;
}

int main() {
    Stock stock;
    printHelp();
    MultiInputHelper ih;
    for (;;) {
        if (!eval(stock, ih)) break;
    }
    return 0;
}
```

fixedPoint.h

```
#pragma once
#include "utils.h"
#include <iostream>

constexpr char decimalSeparator = '.';

constexpr long pow(long b, long e) {
    long res = 1;
    while (--e >= 0) res *= b;
    return res;
}
template<size_t decimals = 2>
class FixedPoint {
    static_assert(decimals < 18);
    static constexpr long factor = pow(10, decimals);
    int64_t val;
    using FP = FixedPoint<decimals>;

public:
    static constexpr FP make(int64_t v) {
        FP r;
        r.val = v;
        return r;
    }
    constexpr FixedPoint(long double v) : val(v * factor) {}
    constexpr FixedPoint(double v) : val(v * factor) {}
    constexpr FixedPoint(float v) : val(v * factor) {}
    constexpr FixedPoint(long v) : val(v * factor) {}
```

```
29      constexpr FixedPoint(int v) : val(v * factor) {}
30      constexpr FixedPoint() : val(0) {}
31
32      constexpr FP operator+(FP rhs) const { return make(val + rhs.val); }
33      constexpr FP operator-(FP rhs) const { return make(val - rhs.val); }
34      constexpr FP operator+() const { return *this; }
35      constexpr FP operator-() const { return make(-val); }
36      constexpr FP operator*(FP rhs) const {
37          return make((val * rhs.val) / factor);
38      }
39      constexpr FP operator/(FP rhs) const {
40          return make((val * factor) / rhs.val);
41      }
42
43      constexpr FP& operator+=(FP rhs) { return *this = *this + rhs; }
44      constexpr FP& operator-=(FP rhs) { return *this = *this - rhs; }
45      constexpr FP& operator/=(FP rhs) { return *this = *this / rhs; }
46      constexpr FP& operator*=(FP rhs) { return *this = *this * rhs; }
47
48      constexpr bool operator< (FP rhs) const { return val < rhs.val; }
49      constexpr bool operator> (FP rhs) const { return val > rhs.val; }
50      constexpr bool operator<=(FP rhs) const { return val <= rhs.val; }
51      constexpr bool operator>=(FP rhs) const { return val >= rhs.val; }
52      constexpr size_t textLen(bool showSign = false) const {
53          size_t baseline = 1 + decimals + showSign; // 1 for the dot
54          if (val < 0) return make(-val).textLen(true);
55          // a int64_t can only store 19 digits
56          if (val < factor * 1) return baseline + 0;
57          if (val < factor * 10) return baseline + 1;
58          if constexpr (decimals <= 16)
59          if (val < factor * 100) return baseline + 2;
60          if constexpr (decimals <= 15)
61          if (val < factor * 1000) return baseline + 3;
62          if constexpr (decimals <= 14)
63          if (val < factor * 10000) return baseline + 4;
64          if constexpr (decimals <= 13)
65          if (val < factor * 100000) return baseline + 5;
66          if constexpr (decimals <= 12)
67          if (val < factor * 1000000) return baseline + 6;
68          if constexpr (decimals <= 11)
69          if (val < factor * 10000000) return baseline + 7;
70          if constexpr (decimals <= 10)
71          if (val < factor * 100000000) return baseline + 8;
72          if constexpr (decimals <= 9)
73          if (val < factor * 1000000000) return baseline + 9;
74          if constexpr (decimals <= 8)
75          if (val < factor * 10000000000) return baseline + 10;
76          if constexpr (decimals <= 7)
77          if (val < factor * 100000000000) return baseline + 11;
78          if constexpr (decimals <= 6)
```

```
        if (val < factor * 1000000000000) return baseline + 12;
        if constexpr (decimals <= 5)
        if (val < factor * 10000000000000) return baseline + 13;
        if constexpr (decimals <= 4)
        if (val < factor * 100000000000000) return baseline + 14;
        if constexpr (decimals <= 3)
        if (val < factor * 1000000000000000) return baseline + 15;
        if constexpr (decimals <= 2)
        if (val < factor * 10000000000000000) return baseline + 16;
        if constexpr (decimals <= 1)
        if (val < factor * 100000000000000000) return baseline + 17;
        if constexpr (decimals <= 0)
        if (val < factor * 1000000000000000000) return baseline + 18;
        return baseline + (19 - decimals);
    }

    //a buffer of size at most 22 is needed
    //(19 for digits, 1 for the dot, 1 for sign, and 1 for \0)
    static constexpr size_t MaxBuffSize = 22;

    // returns the string length (no \0)
    constexpr size_t toString(char* buf, bool showSign) const {
        auto len = textLen(showSign);
        char* p = buf + len;
        auto i = val;
        if (val < 0) {
            *buf = '-';
            i = -val;
        }
        else if (showSign) *buf = '+';
        *p-- = '\0';
        char* decimalPoint = p - decimals;
        for (; p != decimalPoint; --p) {
            *p = (i % 10) + '0';
            i /= 10;
        }
        *p = decimalSeparator;
        while (i) {
            *(--p) = (i % 10) + '0';
            i /= 10;
        }
        return len;
    }
    static constexpr auto isDigit(char c) {
        return c >= '0' && c <= '9';
    }
    // if the return value is nullptr it means we didn't read anything good
    //else we return a pointer to the end of the read FixedPoint
    static constexpr const char* fromString(const char* str, FP& res) {
        long sign = 1;
```

```
129         if (*str == '+') ++str;
130         else if (*str == '-') { ++str; sign = -1; }
131         auto p = str;
132 
133         res.val = 0;
134         while (isDigit(*p)) {
135             res.val *= 10;
136             res.val += *p - '0';
137             ++p;
138         }
139         if (*p == '.' || *p == decimalSeparator) {
140             ++p;
141             if (str == p-1)
142                 str = p;
143             [&] {
144                 size_t i = 0;
145                 for (; i < decimals; ++i) {
146                     if (!isDigit(*p)) {
147                         for (;i < decimals; ++i) res.val *= 10;
148                         return;
149                     }
150                     res.val *= 10;
151                     res.val += *p - '0';
152                     ++p;
153                 }
154                 if (isDigit(*p)) {
155                     if (*p - '0' >= 5) ++res.val;
156                     ++p;
157                 }
158                 while (isDigit(*p)) ++p;
159             }();
160         } else {
161             res.val *= factor;
162         }
163         if (p == str) { return nullptr; std::cout << "NULL"; }
164         res.val *= sign;
165         return p;
166     }
167     static constexpr auto fromString(const char* str) {
168         struct {
169             FP res;
170             const char* str;
171         } res;
172         res.str = fromString(str, res.res);
173         return res;
174     }
175 
176     friend std::ostream& operator<<(std::ostream& s, FP fp) {
177         char buf[MaxBuffSize];
178         fp.toString(buf, s.flags() & s.showpos);
```

```
        return s << buf;
    }

    friend std::istream& operator>>(std::istream& s, FP& fp) {
        long double ld;
        s >> ld;
        fp = ld;
        return s;
    }
};
```

inputHelper.h

```
#pragma once
#include "fixedPoint.h"
#include <iostream>

class MultiInputHelper {
public:
    std::string line;
    const char* p;

public:
    MultiInputHelper() {}
    void getLine(std::string_view msg = "") {
        if (msg!=""){
            std::cout << msg;
            if (msg.back() != ' ') std::cout << ": ";
        }
        std::getline(std::cin, line);
        p = line.c_str();
    }

    void getLineAfterInvalid(std::string_view msg) {
        std::cout << "!";
        getLine(msg);
    }
    void readName(std::string& res, std::string_view msg) {
        eatWhiteSpace(msg);
        if (*p == '"') {
            const char* beg = ++p;
            res = "";
            for (;;) {
                if (!*p) {
                    getLine/*AfterInvalid*/(msg);
                    return readName(res, msg);
                } else if (*p++ == '"') {
                    if (*(p-2) == '\\') {
                        res += std::string(beg, p-beg-2) + '"';
                        beg = p;
```

```
                    } else {
                        res += std::string(beg, p-beg-1);
                        return;
                    }
                }
            }
        } else {
            readString(res, msg);
        }
    }

    void readString(std::string& res, std::string_view msg) {
        eatWhiteSpace(msg);
        const char* beg = p;
        while (*p && !isspace(*p)) {
            ++p;
        }
        auto sz = beg-p;
        if (sz == 0) {
            getLine/*AfterInvalid*/(msg);
            return readString(res, msg);
        } else {
            res = std::string(beg, p - beg);
        }
    }
    char readChar(std::string_view msg = "") {
        eatWhiteSpace(msg);
        return *p++;
    }

    std::string_view readStringView(std::string_view msg) {
        eatWhiteSpace(msg);
        const char* beg = p;
        while (*p && !isspace(*p)) ++p;

        auto sz = beg-p;
        if (sz == 0) {
            getLineAfterInvalid(msg);
            return readStringView(msg);
        } else {
            return std::string_view(beg, p - beg);
        }
    }
    template<size_t precision>
    void readFP(FixedPoint<precision>& res, std::string_view msg) {
        eatWhiteSpace(msg);
        auto str = FixedPoint<precision>::fromString(p, res);
        if (str == nullptr) {
            getLineAfterInvalid(msg);
            readFP(res, msg);
```

```
            return;
        } else {
            p = str;
        }
    }

private:
    void eatWhiteSpace(std::string_view message) {
        for (;;) {
            if (*p == 0) getLine(message);
            if (isspace(*p)) ++p;
            else return;
        }
    }
};
```

Liste liniare dublu înlănțuite

4. Creați o LLDI care să memoreze următoarele informații despre studenții unei grupe: numele, prenumele și trei note (reprezentate prin numere reale de la 1 la 10). Afișați numele, prenumele și media fiecărui student. Scrieți o funcție care calculează și returnează media grupei.

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

#include <iostream>

struct Student {
    std::string firstName;
    std::string lastName;
    double grades[3];
    double average() const {
        return ::average(grades, grades+3);
    }
    friend std::ostream& operator <<(std::ostream& os, const Student& s) {
        return os << s.firstName << " " << s.lastName
                  << " - average: " << s.average();
    }

    friend std::istream& operator >>(std::istream& stream, Student& s) {
        return stream >> s.firstName >> s.lastName
                  >> s.grades[0] >> s.grades[1] >> s.grades[2];
    }
};

void printStudents(const DoubleList<Student>& students) {
    std::cout << "students:\n";
    for (auto& s : students) std::cout << s << "\n";
}
double getAverage(const DoubleList<Student>& students) {
    double res = 0;
    int count = 0;
    for (auto& s : students) {
        res += s.average();
        ++count;
    }
    return res/count;
}

```

```cpp
int main() {
    auto students = DoubleList<Student>::read("Students");
    printStudents(students);
    std::cout << "Average: " << getAverage(students) << "\n";

    return 0;
}
```

5. Creați o LLDI care să memoreze numere întregi citite de la tastatură.

   (a) Scrieți o funcție care primește ca parametru adresa primului nod al listei și o afișează în ambele sensuri.

   (b) Scrieți o funcție care primește ca parametru adresa `p` a unui nod al listei și un număr întreg `x` și adaugă după nodul indicat de `p`, un nod cu informația utilă `x`.

   (c) Scrieți o funcție care primește ca parametru adresa `p` a unui nod și șterge nodul indicat de `p`.

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

#include <iostream>

template<typename T>
void printBothWays(typename DoubleList<T>::Node *n) {
    if (n == nullptr) return;
    typename DoubleList<T>::It it(n);
    std::cout << "Forward:\n";

    while (it.hasNext()) {
        std::cout << *it++ << ", ";
    }
    std::cout << *it << "\n";
    std::cout << "Reverse:\n";
    while (it.hasPrev()) {
        std::cout << *it << ", ";
        --it;
    }
    std::cout << *it << "\n";
}

template<typename T>
void insert(typename DoubleList<T>::Node *n, T val) {
    using Node = typename DoubleList<T>::Node;
    Node* newN = new Node{ val, n->next, n};
    n->next->prev = newN;
    n->next = newN;
}

template<typename T>
void remove(typename DoubleList<T>::Node *n) {
    assert(n->prev && n->next, "Please call list.remove(n).");
```

```
    n->prev->next = n->next;
    n->next->prev = n->prev;
}

int main() {
    DoubleList<int> list = {1, 2, 5, 4 };
    insert(list.top->next, 3);
    std::cout << list << "\n";
    remove<int>(list.bot->prev);

    printBothWays<int>(list.top);

    return 0;
}
```

Grafuri neorientate

7. Reprezentați în memorie un graf neorientat folosind lista muchiilor.

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

#include <iostream>
#include <fstream>

struct Graph_EdgeList {
    using Vertex = unsigned;
    struct Edge {
        Vertex a, b;
        constexpr Edge(Vertex a, Vertex b) :a(a), b(b) {}
        friend std::ostream& operator<< (std::ostream& s, Edge e) {
            return s << "(" << e.a << ", "<< e.b << ")";
        }

        friend std::istream& operator>> (std::istream& s, Edge& e) {
            return s >> e.a >>e.b;
        }
        constexpr bool operator ==(Edge rhs) const {
            return (rhs.a == a && rhs.b == b) ||
                (rhs.b == a && rhs.a == b);
        }
    };
    List<Edge> edges;
    size_t nodeSize;

    template <typename Path>
    static Graph_EdgeList fromFile(Path path) {
        std::ifstream f(path);
        size_t sz = 0;
        f >> sz;
        Graph_EdgeList g { {}, sz };
        Vertex a, b;
        while (f >> a >> b) { g.addEdge(a, b); }
        return g;
    }
    void addEdge(Vertex a, Vertex b) { edges.emplace_front(a, b); }
    void addEdge(Edge e) { edges.push_front(e); }

```

```
    friend std::ostream& operator<<(std::ostream& s,
                                    const Graph_EdgeList& g) {
        return s << g.edges;
    }
    constexpr bool hasEdge(Edge e) const {
        return edges.findElem(e)!= nullptr;
    }

    constexpr bool hasEdge(Vertex a, Vertex b) const {
        return hasEdge({a, b});
    }
};

int main() {
    auto g = Graph_EdgeList::fromFile("nodes.txt");
    std::cout << "g:" << g << "\n";
    std::cout << "has: "<< g.hasEdge(3, 2);
    return 0;
}
```

8. Afișați toate lanțurile elementare dintr-un graf dat.

```
#include "utils.h"
#include "graphAdjacencyList.h"

#include <iostream>

using Graph = Graph_AdjacencyList;

// A simple path or a simple cycle is a path or cycle that has no
// repeated vertices and consequently no repeated edges.
void printSimplePaths(const Graph& g, PtrRange<Graph::Vertex> verts) {
    for (auto v : g[verts.back()]) {
        //if (verts.front() == v) {
        //*verts.last++ = v;
        //    continue;
        // } else
        if (verts.contains(v)) { continue; }
        auto res = PtrRange<Graph::Vertex>(verts.first, verts.last+1);
        *verts.last = v;
        std::cout << res << "\n";
        printSimplePaths(g, res);
    }
}
void printSimplePaths(const Graph& g) {
    // I assume that a node is not a simple path.
    std::cout << "Simple Paths: \n";
    if (g.edgeCount() == 0) {
        std::cout << "No simple paths found\n";
        return;
```

```
    }
    using Vertex = Graph::Vertex;
    Vertex* verts = (Vertex*) alloca(sizeof(Vertex) * g.vertCount());
    for (Vertex v = 0; v < g.vertCount(); ++v) {
        verts[0] = v;
        printSimplePaths(g, PtrRange<Vertex>(verts, verts+1));
    }
}

int main() {
    auto g = Graph::fromFile("nodes.txt");
    std::cout << "g:\n" << g << "\n";
    printSimplePaths(g);
    return 0;
}
```

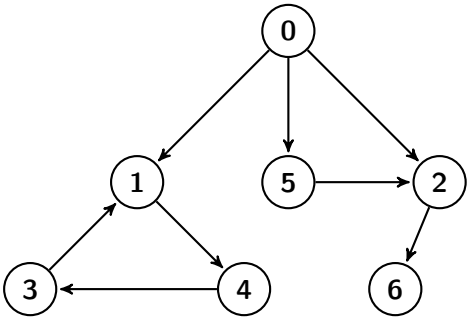
9. Se consideră un graf neorientat fără vârfuri izolate. Determinați dacă graful este eulerian și, în caz afirmativm afișați un ciclu eulerian.

```
#include "utils.h"
#include "graphEdgeList.h"

#include <iostream>

using Graph = Graph_EdgeList;

using Vertex = Graph::Vertex;
using Edge = Graph::Edge;
// has a cycle that visits every edge once
bool isEulerian(Edge* begin,
                Edge* current,
                Edge* end) {
    /*
      edges right of current are unvisited,
      edges on the left form a chain
     */
    if (end - current == 0) {
        if (begin->a == (end-1)->b) {
            std::cout << "Is Eulerian With solution: ";
            for (auto* it = begin; it != end; ++it) {
                std::cout << it->a << " ";
            }
            std::cout << (end-1)->b;
            std::cout << "\n";
            return true;
        }
    }
    auto last = (current-1)->b;
    for (auto* it = current; it != end; ++it) {
        if (it->a == last) ;
```

```
32          else if (it->b == last) { std::swap(it->b, it->a); }
33          else continue;
34          std::swap(*current, *it);
35          if (isEulerian(begin, current+1, end)) {
36              return true;
37          }
38          std::swap(*current, *it);
39      }
40      return false;
41  }
42  void printIsEulerian(const Graph& g) {
43      size_t sz = g.edges.size();
44      assert(sz != 0, "size can't be 0");
45      //allocate memory on the stack
46      Edge *edges = (Edge*) alloca(sizeof(Edge)*sz);
47      g.edges.copyTo(edges);
48      if (!isEulerian(edges, edges+1, edges + sz)) {
49          std::cout << "Graph is not eulerian.\n";
50      }
51  }
52
53  int main() {
54      auto g = Graph::fromFile("nodes.txt");
55      std::cout << "g:\n" << g << "\n";
56      printIsEulerian(g);
57      return 0;
58  }
```

Grafuri orientate
Parcurgerea grafurilor

10. Parcurgeți în lățime și în adâncime considerând drept nod de start, fiecare nod al grafului. Scrieți rezultatele obținute.




```cpp
#include "utils.h"
#include "dirGraphMat.h"
#include "dirGraphAdjacencyList.h"

#include <iostream>

void visitDepthFirst(const DirGraph_Mat& g,
                     DirGraph_Mat::Vertex current,
                     DirGraph_Mat::Vertex* visited) {
    std::cout << current << " ";
    visited[current] = true;
    auto line = g[current];
    for (size_t i = 0; i < g.size; ++i) {
        if (line[i]) {
            if (visited[i]) continue;
            visitDepthFirst(g, i, visited);
        }
    }
}

void visitDepthFirst(const DirGraph_Mat& g) {
    using Graph = DirGraph_Mat;
    using Vertex = Graph::Vertex;

    Vertex* visited = (Vertex*)alloca(sizeof(Vertex) * g.size);
    for (Vertex i = 0; i < g.size; ++i) {
        std::cout << "Nodes from " << i << ":\n";
```

```
        for (Vertex j = 0; j <= g.size; ++j) visited[j] = 0;
        visitDepthFirst(g, i, visited);
        std::cout << "\n";
    }
    std::cout << "\n";
}

template<typename T>
struct SimpleQueue {
    T* first;
    T* last;
    constexpr void push(const T& val) {
        *(last++) = val;
    }

    T pop() {
        assert(first < last, "Queue empty");
        return *(first++);
    }
    constexpr bool empty() const {
        return first == last;
    }
};

void visitBreadthFirst(const DirGraph_AdjacencyList& g,
                       DirGraph_AdjacencyList::Vertex current,
                       SimpleQueue<DirGraph_AdjacencyList::Vertex>& queue,
                       DirGraph_AdjacencyList::Vertex* visited) {
    using Graph = DirGraph_AdjacencyList;
    using Vertex = Graph::Vertex;
    queue.push(current);
    visited[current] = true;
    while (!queue.empty()) {
        Vertex i = queue.pop();

        std::cout << i << " ";
        for (auto& v : g[i]) {
            if (visited[v]) continue;
            visited[v] = true;
            queue.push(v);
        }
    }
}
void visitBreadthFirst(const DirGraph_AdjacencyList& g) {
    using Graph = DirGraph_Mat;
    using Vertex = Graph::Vertex;

    Vertex* visited = (Vertex*)alloca(sizeof(Vertex) * g.vertCount());
    Vertex* queueData = (Vertex*)alloca(sizeof(Vertex) * g.arcCount());
    for (Vertex i = 0; i < g.vertCount(); ++i) {
```

```
78          std::cout << "Nodes from " << i << ":\n";
79          SimpleQueue<Vertex> q { queueData, queueData };
80
81          for (Vertex j = 0; j <= g.vertCount(); ++j) visited[j] = 0;
82          visitBreadthFirst(g, i, q, visited);
83          std::cout << "\n";
84      }
85      std::cout << "\n";
86  }
87
88  int main() {
89      std::cout << "Depth first:\n";
90      auto g = DirGraph_Mat::fromFile("nodes.txt");
91      g.printMatrix();
92      std::cout << "\n";
93      visitDepthFirst(g);
94
95      std::cout << "Breadth first:\n";
96
97      auto g2 = DirGraph_AdjacencyList::fromFile("nodes.txt");
98      std::cout << g2 << "\n";
99      visitBreadthFirst(g2);
100     return 0;
101 }
```

Parcurgerea grafurilor. Grafuri ponderate
Drumuri de cost minim

4. Presupunem că dispunem de o hartă cu n orașe. Unele dintre acestea sunt unite prin șosele, acestea putând fi sau nu cu sens unic. Pentru fiecare șosea ce unește două orașe se cunoaște lungimea în kilometri.

   (a) Un turist se află într-un oraș s și vrea să ajungă cu mașina în orașul f. Se cere să se afle traseul de lungime minimă dintre cele două orașe.

   (b) Care sunt traseele de lungime minimă între s și toate celelalte orașe?

   (c) Se cer traseele de lungime minimă între oricare două orașe de pe hartă.

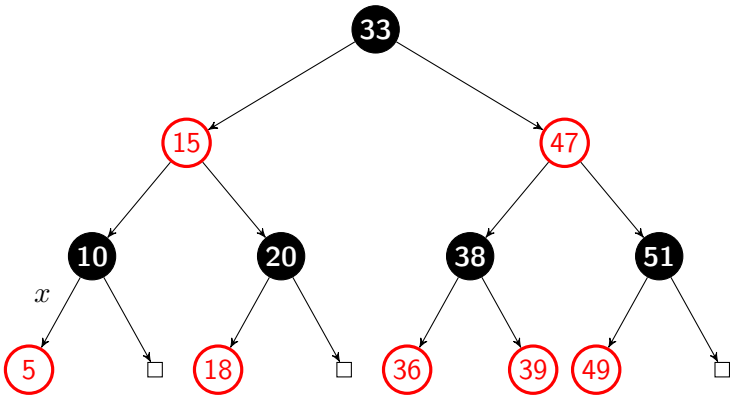
```cpp
#include "weightedDirGraphMat.h"
#include <iostream>
#include <iomanip>

using Graph = WeightedDirGraph_Mat;
using Vertex = Graph::Vertex;
using Weight = Graph::Weight;

constexpr Vertex invalidVert = -1;

template<typename T>
using Matrix = ManagedMatrixPtrRange<T>;

struct DijkstraRes {
    ManagedPtrRange<Weight> dist;
    ManagedPtrRange<Vertex> prev;
    Vertex from;

    bool printPath(Vertex u) {
        StaticVector<Vertex> s(prev.size());
        while (u != from) {
            if (prev[u] == invalidVert) return false;
            s.push_back(u);
            u = prev[u];
        }
        std::cout << from<< " ";
        for (auto it = s.last-1; it > s.first -1; --it) {
            std::cout << *it << " ";
        }
        std::cout << "\n";
        return true;
```

```
    }
};
auto dijkstra(const Graph& g, Vertex from) {
    DijkstraRes res = { {g.size()}, { g.size()}, from};
    StaticVector<Vertex> q(g.size(), g.size());
    for (Vertex v = 0; v < g.size(); ++v) {
        res.dist[v] = Graph::inf;
        res.prev[v] = invalidVert;
        q[v] = v;
    }
    res.dist[from] = 0;

    while (q.size() != 0) {
        auto it = q.min();
        auto u = *it;
        q.remove(it);
        for (Vertex v = 0; v < g.size(); ++v) {
            if (v == u) continue;
            auto val = g.mat(u, v);
            if (val != Graph::inf) {
                auto alt = res.dist[u] + val;
                if (alt < res.dist[v]) {
                    res.dist[v] = alt;
                    res.prev[v] = u;
                }
            }
        }
    }
    return res;
}

struct RoyFloydRes {
    Matrix<Weight> dist;
    Matrix<Vertex> next;

    bool printPath(Vertex u, Vertex v) {
        if (next(u, v) == invalidVert) {
            std::cout << "No path\n";
            return false;
        }
        while (u != v) {
            std::cout << u << " ";
            u = next(u, v);
        }
        std::cout << v << "\n";
        return true;
    }
};

RoyFloydRes royFloyd(const Graph& g) {
```

```
    auto n = g.size();
    RoyFloydRes res = {g.mat, {n,n} };

    for (Vertex u = 0; u < n; ++u) {
        for (Vertex v = 0; v < n; ++v) {
            res.next(u, v) = res.dist(u, v) != Graph::inf? v : invalidVert;
        }
    }
    for (Vertex k = 0; k < n;++k)
        for (Vertex i = 0; i < n;++i)
            for (Vertex j = 0; j < n; ++j) {
                if (res.dist(i, j) > res.dist(i, k) + res.dist(k, j)) {
                    res.dist(i, j) = res.dist(i, k) + res.dist(k, j);
                    res.next(i, j) = res.next(i, k);
                }
            }
    return res;
}

int main() {
    Graph g = Graph::fromFile("nodes.txt");
    std::cout << std::setw(3) << g;

    auto s = read<Vertex>("Start");
    auto e = read<Vertex>("End");
    auto dij = dijkstra(g, s);
    dij.printPath(e);

    std::cout << "\nR-F:\n";

    auto res = royFloyd(g);
    std::cout << std::setw(3) << res.dist;
    std::cout << "Next:\n" << res.next << "\n";
    res.printPath(1, 2);

    auto res2 = dijkstra(g, 1);

    res2.printPath(2);
    return 0;
}
```

Arbori

7. Fie arborele:



1