

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

Fişiere comune

utils.h

```
1  #pragma once
2  #include <initializer_list>
3  #include <iostream>
4  #include <fstream>
5
6  constexpr size_t getSize(std::initializer_list<double> l) {
7      size_t n = 0;
8      auto it = l.begin();
9      auto end = l.end();
10     while (it++ != end) ++n;
11     return n;
12 }
13
14 constexpr void getSize(std::initializer_list<
15     std::initializer_list<double>> list,
16     int& s1, int& s2) {
17     constexpr int invalid = -1;
18     s1 = 0;
19     s2 = invalid;
20
21     for (const auto& l: list) {
22         int i = getSize(l);
23         ++s1;
24         if (s2 != invalid && i != s2) {
25             //std::cerr << i << "!=" << s2 << "\n";
26             throw std::logic_error("Fixed line size expected");
27         }
28         s2 = i;
29     }
30     //std::cerr << "size: " << s1 << ", " << s2 << "\n";
31 }
32 constexpr int MAX_SZ = 256;
33 inline size_t readSize(const char* name, int a = 1, int b = MAX_SZ) {
34     int res;
35     do {
36         std::cout << name << ": ";
37         std::cin >> res;
38     } while (res < a || res > b);
39     return res;
```

```

40 }
41
42
43 inline void assert(bool cond, const char* msg) {
44     if (!cond) throw std::logic_error(msg);
45 }
46 template<typename T>
47 inline T read(const char* name) {
48     T res;
49     std::cout << name << ": ";
50     std::cin >> res;
51     return res;
52 }
53
54 template<typename It>
55 constexpr auto average(It begin, It end) {
56     std::remove_const_t<std::remove_reference_t<decltype(*begin)>> sum {};
57     size_t count = 0;
58     while (begin != end) {
59         sum += *begin;
60         ++begin;
61         ++count;
62     }
63     return sum / count;
64 }
65 size_t fileLineCount(std::ifstream& f) {
66     size_t count = 0;
67     while (!f.eof()) {
68         if (auto c = f.get(); c == '\n') ++count;
69     }
70     return count;
71 }

```

matrix.h

```

1  #pragma once
2  #include "utils.h"
3
4  #include <iostream>
5  #include <cmath>
6  #include <cstring>
7  #include <utility>
8
9  struct Mat {
10     int m, n;
11     double *data;
12     Mat() : m(0), n(0), data(nullptr) {}
13     Mat(int m, int n) : m(m), n(n) { data = new double[m*n]; }
14     //template<int m, int n>
15     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->n == 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);

```

```

166 }
167 Mat& sub(const Mat& a, const Mat& b, Mat& res) {
168     return add(a, neg(b, res), res);
169 }
170 Mat& mul(const Mat& a, const Mat& b, Mat& res) {
171     assert(a.n == b.m, "Sizes don't match, can't multiply");
172     res.setSize(a.m, b.n);
173
174     for (int i = 0; i < res.m; ++i)
175         for (int j = 0; j < res.n; ++j) {
176             res.at(i, j) = 0;
177             for (int k = 0; k < a.n; ++k)
178                 res.at(i, j) += a.at(i, k) * b.at(k, j);
179         }
180     return res;
181 }
182
183 Mat& trans(const Mat& a, Mat& res) {
184     assert(a.data != res.data, "Can't transpose inplace");
185     res.setSize(a.n, a.m);
186
187     for (int i = 0; i < res.m; ++i)
188         for (int j = 0; j < res.n; ++j)
189             res.at(i, j) = a.at(j, i);
190     return res;
191 }

```

vector.h

```

1  #pragma once
2  #include "utils.h"
3
4  #include <iostream>
5  #include <utility>
6  #include <cmath>
7
8  struct Vec {
9      double *_begin, *_end;
10
11      constexpr double* begin() { return _begin; }
12      constexpr const double* begin() const { return _begin; }
13
14      constexpr double* end() { return _end; }
15      constexpr const double* end() const { return _end; }
16
17      constexpr Vec() : _begin(nullptr), _end(nullptr) {}
18      explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
19      Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
20          auto it = _begin;
21          for (const auto& v : list) *(it++) = v;

```

```

22 }
23 Vec(const Vec& rhs) : Vec(rhs.size()) {
24     auto it = _begin;
25     for (const auto& v : rhs) *(it++) = v;
26 }
27 Vec(Vec&& rhs) noexcept
28     : _begin(std::exchange(rhs._begin, nullptr)),
29     _end(std::exchange(rhs._end, nullptr)) {}
30
31 Vec& operator=(const Vec& rhs) {
32     if (size() != rhs.size()) {
33         this->~Vec();
34         new (this) Vec(rhs.size());
35     }
36     auto it = _begin;
37     for (const auto& v : rhs) *(it++) = v;
38     return *this;
39 }
40 Vec& operator=(Vec&& rhs) noexcept {
41     this->~Vec();
42     _begin = std::exchange(rhs._begin, nullptr);
43     _end = std::exchange(rhs._end, nullptr);
44     return *this;
45 }
46
47 ~Vec() { delete[] _begin; }
48
49 constexpr size_t size() const { return _end - _begin; }
50
51 constexpr double& operator[](size_t i) { return _begin[i]; }
52 constexpr double operator[](size_t i) const { return _begin[i]; }
53
54 void setSize(size_t n) {
55     if (size() == n) return;
56     *this = Vec(n);
57 }
58
59 friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
60     s << "(";
61     double* it = v._begin;
62     for (double* end = v._end - 1; it < end; ++it)
63         s << *it << ", ";
64
65     if (it < v._end) s << *it;
66
67     return s << ")";
68 }
69 static Vec read() {
70     Vec res(readSize("n"));
71     for (auto& v: res) std::cin >> v;

```



```

72         return res;
73     }
74     double norm() const;
75 };
76 void assertSizes(const Vec& a, const Vec& b) {
77     assert(a.size() == b.size(), "Sizes don't match");
78 }
79
80 Vec& add(const Vec& a, const Vec& b, Vec& res) {
81     assertSizes(a, b);
82     res.setSize(a.size());
83     auto aIt = a.begin();
84     auto bIt = b.begin();
85     for (auto& v : res) v = *(aIt++) + *(bIt++);
86     return res;
87 }
88
89 Vec& mul(double a, const Vec& b, Vec& res) {
90     res.setSize(b.size());
91     auto it = b.begin();
92     for (auto& v : res) v = a * *(it++);
93     return res;
94 }
95
96 Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }
97
98 Vec& sub(const Vec& a, const Vec& b, Vec& res) {
99     return add(a, neg(b, res), res);
100 }
101 double dot(const Vec& a, const Vec& b) {
102     double res = 0;
103     assertSizes(a, b);
104     auto bIt = b.begin();
105     for (auto& v : a) res += v * *(bIt++);
106     return res;
107 }
108 double norm(const Vec& a) {
109     return std::sqrt(dot(a, a));
110 }
111 double Vec::norm() const {
112     return ::norm(*this);
113 }

```

stack.h

```

1  #pragma once
2  #include "utils.h"
3
4  template<typename T>
5  struct Stack {

```

```

6      struct Node {
7          T val;
8          Node* next;
9      };
10
11      Stack(const Stack&) = delete;
12      Stack& operator=(const Stack&) = delete;
13      Node* top;
14      Stack(Node* top = nullptr) : top(top) {}
15      ~Stack() {
16          while (top) {
17              Node* n = top;
18              top = top->next;
19              delete n;
20          }
21      }
22      void push(T val) {
23          Node *n = new Node{ val, top };
24          top = n;
25      }
26      void quickPop() {
27          Node* n = top;
28          top = top->next;
29          delete n;
30      }
31      T pop() {
32          assert(top, "Empty stack");
33          Node* n = top;
34          top = top->next;
35          T res = n->val;
36          delete n;
37          return res;
38      }
39      bool empty() const { return top == nullptr; }
40      friend std::ostream& operator<<(std::ostream& s, const Stack& st) {
41          if (st.empty()) return s << "{}";
42          s << "{";
43          const Node* it = st.top;
44          while (it->next != nullptr) {
45              s << it->val << ", ";
46              it = it->next;
47          }
48
49          return s << it->val << "}";
50      }
51 };

```

queue.h

```

1  #pragma once
2  #include "utils.h"
3
4  template<class T>
5  struct Queue {
6      struct Node {
7          T val;
8          Node* next;
9      };
10     Node* top;
11     Node* bot;
12     Queue(Node* top = nullptr, Node* bot = nullptr) : top(top), bot(bot) {}
13     /*
14     Queue(const Queue&) = delete;
15     Queue& operator=(const Queue&) = delete;
16     ~Queue() {
17         while (top) {
18             Node* n = top;
19             top = top->next;
20             delete n;
21         }
22     }*/
23     void push(T val) {
24         Node *n = new Node{ val, nullptr };
25         if (bot == nullptr) {
26             top = n;
27         }
28         else {
29             bot->next = n;
30         }
31         bot = n;
32     }
33
34     T pop() {
35         assert(top, "Empty queue");
36         Node* old = top;
37         T res = old->val;
38         top = top->next;
39         if (top == nullptr)
40             bot = nullptr;
41         delete old;
42         return res;
43     }
44     bool empty() const { return top == nullptr; }
45     static Queue read(const char* msg) {
46         Queue q;
47         std::cout << msg << ":\n";
48         int len1 = readSize("n");

```

```

49     for (int i = 0; i < len1; ++i) {
50         T s;
51         std::cin >> s;
52         q.push(s);
53     }
54     return std::move(q);
55 }
56
57 friend std::ostream& operator<<(std::ostream& s, const Queue& st) {
58     if (st.empty()) return s << "{}";
59     s << "{";
60     const Node* it = st.top;
61     while (it->next != nullptr) {
62         s << it->val << ", ";
63         it = it->next;
64     }
65
66     return s << it->val << "}";
67 }
68 };

```

list.h

```

1  #pragma once
2  #include "utils.h"
3  #include "ptrRange.h"
4
5  #include <type_traits>
6  #include <utility>
7
8
9  // We should never check for (bot == nullptr),
10 // so we don't update it when the list becomes empty.
11 template<typename T>
12 struct List {
13     struct Node {
14         T val;
15         Node* next;
16     };
17     struct It {
18         const Node* n;
19         constexpr It(const Node* n) : n(n) {}
20         constexpr It& operator++() {
21             n = n->next;
22             return *this;
23         }
24         constexpr auto& operator*() { return n->val; }
25         constexpr bool operator==(const It& rhs) { return n == rhs.n; }
26         constexpr bool operator!=(const It& rhs) { return n != rhs.n; }
27 };

```

```

28 Node* top;
29 Node* bot;
30 constexpr List(Node* top = nullptr, Node* bot = nullptr)
31     : top(top), bot(bot) {}
32 List(const List&) = delete;
33 List& operator=(const List&) = delete;
34
35 constexpr It begin() const { return top; }
36 constexpr It end() const { return nullptr; }
37
38 constexpr const T& front() const { return top->val; }
39 constexpr T& front() { return top->val; }
40 constexpr size_t size() const {
41     size_t sz = 0;
42     Node* n = top;
43
44     while (n) {
45         ++sz;
46         n = n->next;
47     }
48     return sz;
49 }
50
51 constexpr void copyTo(T* ptr) const {
52     for (auto v : *this) {
53         *(ptr++) = v;
54     }
55 }
56 ManagedPtrRange<T> toManagedPtrRange() const {
57     auto res = ManagedPtrRange<T>(size());
58     copyToMemory(res.first);
59     return res;
60 }
61
62 ~List() {
63     while (top) {
64         Node* n = top;
65         top = top->next;
66         delete n;
67     }
68 }
69 List(List&& rhs) noexcept
70     : top(std::exchange(rhs.top, nullptr)),
71       bot(std::exchange(rhs.bot, nullptr)) {}
72 List& operator=(List&& rhs) noexcept {
73     this->~List();
74     this->top = std::exchange(rhs.top, nullptr);
75     this->bot = std::exchange(rhs.bot, nullptr);
76 }
77

```

```

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             return s << it->val << "}";
164         }
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; });

```



```

228     }
229     const Node* findElem(T& e) const {
230         return find([&](T& other) { return other == e; });
231     }
232 };
233
234 // keeps in l all the elements that don't satisfy the predicate p
235 // and returns pair of:
236 // - a reference to the original list
237 // - a list containing the elements that satisfy p
238 template<typename T, typename P>
239 constexpr auto partition_split(List<T>& l, P p) {
240     struct res_t {
241         List<T>& notSatisfying;
242         List<T> satisfying;
243     } res = { l, {} };
244     List<T>& sat = res.satisfying;
245     auto insertNode = [&sat](auto* n) {
246         n->next = sat.top;
247         sat.top = n;
248         if (sat.bot == nullptr) {
249             sat.bot = n;
250         }
251     };
252     l.apply_on(p, insertNode);
253     return res;
254 }

```

doubleList.h

```

1  #pragma once
2  #include "utils.h"
3
4  #include <utility>
5
6  template<typename T>
7  struct DoubleList {
8      struct Node {
9          T val;
10         Node* next;
11         Node* prev;
12     };
13     struct It {
14         const Node* n;
15         constexpr It(const Node* n) : n(n) {}
16         constexpr It& operator++() { n = n->next; return *this; }
17         constexpr It& operator--() { n = n->prev; return *this; }
18
19         constexpr It operator++(int) { auto r = n; ++(*this); return r; }
20         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;

```

```

171         if (prev->next == nullptr) {
172             bot = prev;
173         }
174         f(it);
175         it = prev;
176     } else {
177         prev = it;
178         it = it->next;
179     }
180 }
181 }
182 // P is a predicate on T
183 template<typename P>
184 Node* find(P p) {
185     auto n = top;
186     for (; n; n = n->next) {
187         if (p(n->val))
188             return n;
189     }
190     return n;
191 }
192 void insert(Node* n, T val) {
193     if (n == bot) { push_back(val); return; }
194     Node* newN = new Node{ val, n->next, n};
195     n->next->prev = newN;
196     n->next = newN;
197 }
198 size_t size() const {
199     auto it = top;
200     size_t res = 0;
201     while (it) { it = it->next; ++res; }
202     return res;
203 }
204 };

```

graphEdgeList.h

```

1  #pragma once
2  #include "utils.h"
3  #include "list.h"
4
5  #include <iostream>
6  #include <fstream>
7
8  struct Graph_EdgeList {
9      using Vertex = unsigned;
10     struct Edge {
11         Vertex a, b;
12         constexpr Edge(Vertex a, Vertex b) :a(a), b(b) {}
13         friend std::ostream& operator<< (std::ostream& s, Edge e) {

```

```

14         return s << "(" << e.a << ", " << e.b << ")";
15     }
16
17     friend std::istream& operator>> (std::istream& s, Edge& e) {
18         return s >> e.a >> e.b;
19     }
20     constexpr bool operator==(Edge rhs) const {
21         return (rhs.a == a && rhs.b == b) ||
22                (rhs.b == a && rhs.a == b);
23     }
24 };
25 List<Edge> edges;
26 size_t nodeSize;
27
28 template <typename Path>
29 static Graph_EdgeList fromFile(Path path) {
30     std::ifstream f(path);
31     size_t sz = 0;
32     f >> sz;
33     Graph_EdgeList g { {}, sz };
34     Vertex a, b;
35     while (f >> a >> b) { g.addEdge(a, b); }
36     return g;
37 }
38 void addEdge(Vertex a, Vertex b) { edges.emplace_front(a, b); }
39 void addEdge(Edge e) { edges.push_front(e); }
40
41 friend std::ostream& operator<<(std::ostream& s,
42                                const Graph_EdgeList& g) {
43     return s << g.edges;
44 }
45 constexpr bool hasEdge(Edge e) const {
46     return edges.findElem(e) != nullptr;
47 }
48
49 constexpr bool hasEdge(Vertex a, Vertex b) const {
50     return hasEdge({a, b});
51 }
52 };

```

graphAdjacencyList.h

```

1  #pragma once
2
3  #include "utils.h"
4  #include "ptrRange.h"
5  #include "list.h"
6
7  #include <iostream>
8  #include <fstream>

```

```

9  #include <algorithm> //std::count
10 #include <alloca.h>
11
12
13 struct Graph_AdjacencysList {
14     using Vertex = unsigned;
15 private:
16
17     Vertex** adjacencyList;
18     Vertex* adjacencyData;
19
20     size_t vertSize;
21     size_t edgeSize;
22     using Graph = Graph_AdjacencysList;
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 edgeCount() const { return edgeSize; }
51
52 private:
53     Graph_AdjacencysList(Vertex** adjacencyList, Vertex* adjacencyData,
54                          size_t vertSize, size_t edgeSize)
55         : adjacencyList(adjacencyList), adjacencyData(adjacencyData),
56           vertSize(vertSize), edgeSize(edgeSize) {
57         adjacencyData = new Vertex[edgeSize*2];
58     }

```

```

59 public:
60     Graph_AdjacencysList(const Graph&) = delete;
61     Graph_AdjacencysList(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_AdjacencysList();
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_AdjacencysList() {
75         delete[] adjacencyData;
76         delete[] adjacencyList;
77     }
78
79     template <typename Path>
80     static Graph_AdjacencysList 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         }

```



```

109     // std::cout << "cumsum: " << counts << "\n";
110
111     PtrRange<Vertex*> countsPtr((Vertex**) counts.begin(),
112                                (Vertex**) counts.end());
113     Vertex* data = new Vertex[edgeSize*2];
114     for (size_t i = vertSize-1; i > 0; --i) {
115         countsPtr[i] = list[i] = data + counts[i-1];
116     }
117     countsPtr[0] = list[0] = data;
118     list[vertSize] = data+(edgeSize*2);
119
120     //std::cout << "thing: " << countsPtr << "\n";
121
122     f.clear();
123     f.seekg(0);
124     f >> vertSize;
125     Vertex a, b;
126     while (f >> a >> b) {
127         *(countsPtr[a]++) = b;
128         *(countsPtr[b]++) = a;
129     }
130     return Graph_AdjacencyList(list, data, vertSize, edgeSize);
131 }
132
133 friend std::ostream& operator<<(std::ostream& s,
134                                const Graph_AdjacencyList& g) {
135     for (size_t i = 0; i < g.vertSize; ++i) {
136         s << "[" << i << "] = " << g[i] << "\n";
137     }
138     return s;
139 }
140 constexpr bool hasEdge(Vertex a, Vertex b) const {
141     return (*this)[a].contains(b);
142 }
143 };

```

ptrRange.h

```

1  #pragma once
2  #include <iostream>
3  #include <utility>
4  #include <iomanip>
5
6  template<typename T>
7  struct PtrRange {
8      T* first;
9      T* last;
10     constexpr PtrRange(T* first, T* last)
11         : first(first), last(last){}
12

```

```

13 constexpr PtrRange(T* first, size_t sz)
14     : first(first), last(first+sz){}
15 constexpr T* begin() { return first; }
16 constexpr T* end() { return last; }
17
18 constexpr const T* begin() const { return first; }
19 constexpr const T* end() const { return last; }
20
21 constexpr T& front() { return *first; }
22 constexpr T& back() { return *(last-1); }
23 constexpr const T& front() const { return *first; }
24 constexpr const T& back() const { return *(last-1); }
25
26
27 friend std::ostream& operator<<(std::ostream& s, PtrRange v) {
28     s << "(";
29     T* it = v.first;
30     for (T* end = v.last - 1; it < end; ++it)
31         s << *it << ", ";
32
33     if (it < v.last) s << *it;
34
35     return s << ")";
36 }
37 constexpr const T& operator[](size_t i) const { return first[i]; }
38 constexpr T& operator[](size_t i) { return first[i]; }
39
40 constexpr size_t size() const { return last - first; }
41
42 constexpr T* find(const T& val) {
43     for (auto it = first; it != last; ++it) {
44         if (*it == val) return it;
45     }
46     return nullptr;
47 }
48
49 constexpr T* min() {
50     auto smallest = first;
51     for (auto it = first+1; it != last; ++it) {
52         if (*it < *smallest)
53             smallest = it;
54     }
55     return smallest;
56 }
57
58 // P is a predicate on T
59 template<typename P>
60 constexpr T* find_if(P p) {
61     for (auto it = first; it != last; ++it) {
62         if (p(it->val)) return it;

```

```

63     }
64     return nullptr;
65 }
66
67 template<typename P>
68 constexpr T* find_neighbouring_if(P p) {
69     for (auto it = first; it != (last-1); ++it) {
70         if (p(it->val, (it+1)->val)) return it;
71     }
72     return nullptr;
73 }
74
75 template<typename P>
76 constexpr T* find_neighbouring(const T& a, const T& b) {
77     for (auto it = first; it != (last-1); ++it) {
78         if (*it == a && *(it+1) == b) return it;
79     }
80     return nullptr;
81 }
82 constexpr bool contains(const T& val) const { return find(val) != nullptr; }
83
84 template<typename P>
85 constexpr T* contains_if(P p) const { return find(p) != nullptr; }
86
87 template<typename P>
88 constexpr T* contains_neighbouring_if(P p) const {
89     return find_neighbouring_if(p) != nullptr;
90 }
91 };
92
93 template<typename T>
94 struct ManagedPtrRange : public PtrRange<T> {
95     ManagedPtrRange(size_t sz)
96         : PtrRange<T>(new T[sz], sz) {}
97
98     ManagedPtrRange(const ManagedPtrRange&) = delete;
99     ManagedPtrRange(ManagedPtrRange&& rhs)
100         : PtrRange<T>(std::exchange(rhs.first, nullptr),
101                     std::exchange(rhs.last, nullptr)) {}
102     ManagedPtrRange& operator=(const ManagedPtrRange&) = delete;
103     ManagedPtrRange& operator=(ManagedPtrRange&& rhs) {
104         this->~ManagedPtrRange();
105         this->first = std::exchange(rhs.fist, nullptr);
106         this->last = std::exchange(rhs.last, nullptr);
107         return *this;
108     }
109     ~ManagedPtrRange() { delete[] this->first; }
110 };
111
112 template<typename T>

```

```

113 struct MatrixPtrRange : public PtrRange<T> {
114     size_t rows, cols;
115     MatrixPtrRange(T* first, size_t rows, size_t cols)
116         : PtrRange<T>(first, rows*cols), rows(rows), cols(cols) {}
117 protected:
118     MatrixPtrRange(T* first, T* last, size_t rows, size_t cols)
119         : PtrRange<T>(first, last), rows(rows), cols(cols) {}
120 public:
121     constexpr T& operator()(size_t i, size_t j) {
122         return this->first[cols*i + j];
123     }
124     constexpr const T& operator()(size_t i, size_t j) const {
125         return this->first[cols*i + j];
126     }
127     friend std::ostream& operator<<(std::ostream& s, MatrixPtrRange p) {
128         auto it = p.begin();
129         auto end = p.end();
130         auto endl = it + p.cols;
131         auto w = s.width();
132         for (; endl <= end; endl += p.cols) {
133             for (; it < endl; ++it)
134                 s << std::setw(w) << *it << " ";
135             s << "\n";
136         }
137         return s;
138     }
139 };
140
141 template<typename T>
142 struct ManagedMatrixPtrRange : public MatrixPtrRange<T> {
143     ManagedMatrixPtrRange(size_t rows, size_t cols)
144         : MatrixPtrRange<T>(new T[rows*cols], rows, cols) {}
145
146     ManagedMatrixPtrRange(const ManagedMatrixPtrRange<T>& rhs) :
147         ManagedMatrixPtrRange(static_cast<const MatrixPtrRange<T>&>(rhs)) {}
148     ManagedMatrixPtrRange(const MatrixPtrRange<T>& rhs)
149         : ManagedMatrixPtrRange(rhs.rows, rhs.cols) {
150         auto itL = this->begin();
151         auto itR = rhs.begin();
152         auto endR = rhs.end();
153         for (; itR != endR; ++itR, ++itL)
154             *itL = *itR;
155     }
156     ManagedMatrixPtrRange(ManagedMatrixPtrRange&& rhs)
157         : MatrixPtrRange<T>(std::exchange(rhs.first, nullptr),
158                             std::exchange(rhs.last, nullptr),
159                             rhs.rows, rhs.cols) {}
160
161     ManagedMatrixPtrRange& operator=(const ManagedMatrixPtrRange<T>& rhs) {
162         *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");

```

```

213         for (auto it = this->last-1; it != pos; --it)
214             *it = *(it-1);
215         ++this->last;
216         *pos = val;
217     }
218     void remove(T* pos) {
219         assert(this->last != this->first, "Empty vector");
220         for (auto it = pos; it != this->last; ++it)
221             *it = *(it+1);
222         --this->last;
223     }
224 };

```

dirGraphMat.h

```

1  #pragma once
2  #include "utils.h"
3  #include "ptrRange.h"
4
5  #include <iostream>
6  #include <fstream>
7  #include <utility>
8
9
10 struct DirGraph_Mat {
11     using Graph = DirGraph_Mat;
12     using Vertex = unsigned;
13     bool* mat;
14     size_t size;
15
16     DirGraph_Mat(size_t size) : mat(new bool[size*size]), size(size) {}
17     DirGraph_Mat(const Graph&) = delete;
18     DirGraph_Mat(Graph&& rhs)
19         : mat(std::exchange(rhs.mat, nullptr)), size(rhs.size) {}
20     Graph& operator=(const Graph&) = delete;
21     Graph& operator=(Graph&& rhs) {
22         this->~DirGraph_Mat();
23         mat = std::exchange(rhs.mat, nullptr);
24         size = rhs.size;
25         return *this;
26     }
27     ~DirGraph_Mat() {
28         delete[] mat;
29     }
30
31     constexpr auto operator[] (Vertex v) const {
32         return PtrRange<const bool>(mat+size*v, size);
33     }
34
35     template <typename Path>

```

```

36     static Graph fromFile(Path path) {
37         std::ifstream f(path);
38         size_t sz = 0;
39         f >> sz;
40         Graph g(sz);
41         Vertex a, b;
42         while (f >> a >> b) { g.addArc(a, b); }
43         return g;
44     }
45     constexpr void addArc(Vertex a, Vertex b) {
46         mat[a * size + b] = true;
47     }
48     constexpr bool hasArc(Vertex a, Vertex b) const {
49         return mat[a * size + b];
50     }
51     void printMatrix() const {
52         bool* it = mat;
53         bool* end = mat + size*size;
54         bool* endl = mat + size;
55         for (;endl < end; endl += size) {
56             for (;it < endl; ++it)
57                 std::cout << *it << " ";
58             std::cout << "\n";
59         }
60     }
61 };

```

dirGraphAdjacencyList.h

```

1  #pragma once
2
3  #include "utils.h"
4  #include "ptrRange.h"
5  #include "list.h"
6
7  #include <iostream>
8  #include <fstream>
9  #include <algorithm> //std::count
10 #include <alloca.h>
11
12
13 struct DirGraph_AdjacencyList {
14     using Vertex = unsigned;
15 private:
16
17     Vertex** adjacencyList;
18     Vertex* adjacencyData;
19
20     size_t vertSize;
21     size_t arcSize;

```

```

22     using Graph = DirGraph_AdjacencysList;
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 *p; }
40         constexpr bool operator==(const Iterator& o) const { return p == o.p; }
41         constexpr bool operator!=(const 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_AdjacencysList(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_AdjacencysList(const Graph&) = delete;
61     DirGraph_AdjacencysList(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_AdjacencysList();
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);

```

```

122         f >> vertSize;
123         Vertex a, b;
124         while (f >> a >> b) {
125             *(countsPtr[a]++) = b;
126         }
127         return Graph(list, data, vertSize, arcSize);
128     }
129
130     friend std::ostream& operator<<(std::ostream& s,
131                                     const Graph& g) {
132         for (size_t i = 0; i < g.vertSize; ++i) {
133             s << "[" << i << "] = " << g[i] << "\n";
134         }
135         return s;
136     }
137     /*
138     constexpr bool hasArc(Vertex a, Vertex b) const {
139         return (*this)[a].contains(b);
140     }*/
141 };

```

weightedDirGraphMat.h

```

1  #include "utils.h"
2  #include "ptrRange.h"
3
4  #include <iostream>
5  #include <limits>
6  #include <utility>
7
8  struct WeightedDirGraph_Mat {
9      using Graph = WeightedDirGraph_Mat;
10     using Vertex = unsigned;
11     using Weight = double;
12
13     using Matrix = ManagedMatrixPtrRange<Weight>;
14     static_assert(std::numeric_limits<Weight>::has_infinity);
15     static constexpr Weight inf = std::numeric_limits<Weight>::infinity();
16     static_assert(inf + 1 == inf);
17     static_assert(inf - 1 == inf);
18     static_assert(1 < inf);
19
20     Matrix mat;
21     constexpr size_t size() const { return mat.cols; }
22     WeightedDirGraph_Mat(size_t size): mat(size, size) {
23         for (size_t i = 0; i < size*size; ++i) {
24             mat[i] = inf;
25         }
26         for (size_t i = 0; i < size; ++i)
27             addArc(i, i, 0);

```

```

28     }
29
30     static WeightedDirGraph_Mat fromFile(const char* path) {
31         std::ifstream f(path);
32         size_t sz = 0;
33         f >> sz;
34         Graph g(sz);
35         Vertex a, b;
36         Weight w;
37
38         while (f >> a >> b >> w) { g.addArc(a, b, w); }
39         return g;
40     }
41
42     friend std::ostream& operator<< (std::ostream& s, const Graph& g) {
43         return s << g.mat;
44     }
45
46     constexpr void addArc(Vertex a, Vertex b, Weight w) {
47         mat(a, b) = w;
48     }
49     constexpr Weight arcWeight(Vertex a, Vertex b) const {
50         return mat(a, b);
51     }
52 };

```

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 specifice¹, 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 linii² sau pe coloane³. Se vor folosi tablouri bidimensionale alocate static.

```

1  #include "utils.h"
2
3  #include <iostream>
4  #include <cmath>
5
6  struct Mat {
7      double data[MAX_SZ][MAX_SZ] {};
8      int m, n;
9
10     Mat() : m(0), n(0) {}
11     Mat(int m, int n) : m(m), n(n) {}
12
13     static Mat read() {
14         Mat res(readSize("m"), readSize("n"));
15
16         for (int i = 0; i < res.m; ++i)
17             for (int j = 0; j < res.n; ++j)
18                 std::cin >> res.data[i][j];
19         return res;
20     }
21     void setSize(int m, int n) {
22         this->m = m;
23         this->n = n;
24     }
25     double& at(int i, int j) { return data[i][j]; }
26     double at(int i, int j) const { return data[i][j]; }
27

```

¹Dacă $A \in \mathcal{M}_{m \times n}(\mathbb{R})$, atunci $\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$, $\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$, $\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}$.

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

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

```

28     void print(const char* name) const {
29         std::cout << name << " = Mat " << m << "x" << n << "{\n";
30         for (int i = 0; i < m; ++i) {
31             for (int j = 0; j < n; ++j)
32                 std::cout << at(i, j) << " ";
33             std::cout << "\n";
34         }
35         std::cout << "}\n";
36     }
37 private:
38     enum class Type { Row, Col };
39     template<Type type>
40     bool isStrictlyDiagonallyDominantImpl() const {
41         assert(m == n, "Matrix must be square");
42         for (int i = 0; i < m; ++i) {
43             double val = std::abs(at(i, i));
44             double sum = -val;
45             for (int j = 0; j < m; ++j)
46                 sum += std::abs(type == Type::Col ? at(j, i) : at(i, j));
47             if (sum >= val) return false;
48         }
49         return true;
50     }
51
52     template<Type type>
53     double normImpl(int sz1, int sz2) const {
54         double max = -1;
55         for (int j = 0; j < sz1; ++j) {
56             double x = 0;
57             for (int i = 0; i < sz2; ++i)
58                 x += std::abs(type == Type::Col ? at(j, i) : at(i, j));
59             if (x > max) max = x;
60         }
61         return max;
62     }
63 public:
64     double norm1() const { return normImpl<Type::Row>(n, m); }
65     double normInf() const { return normImpl<Type::Col>(m, n); }
66     double normF() const {
67         double res = 0;
68         for (int i = 0; i < m; ++i)
69             for (int j = 0; j < n; ++j)
70                 res += at(i, j) * at(i, j);
71
72         return std::sqrt(res);
73     }
74     bool isStrictlyRowDiagonallyDominant() const {
75         return isStrictlyDiagonallyDominantImpl<Type::Row>();
76     }
77     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 tastatură a celor n componente ale unui vector, afișarea componentelor vectorului sub forma unui n -uplu de elemente. Se vor folosi tablouri

unidimensionale allocate dinamic.

```
1  #include "utils.h"
2
3  #include <iostream>
4  #include <utility>
5  #include <cmath>
6
7  struct Vec {
8      double *_begin, *_end;
9
10     constexpr double* begin() { return _begin; }
11     constexpr const double* begin() const { return _begin; }
12
13
14     constexpr double* end() { return _end; }
15     constexpr const double* end() const { return _end; }
16
17     constexpr Vec() : _begin(nullptr), _end(nullptr) {}
18     explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
19     Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
20         auto it = _begin;
21         for (const auto& v : list) *(it++) = v;
22     }
23     Vec(const Vec&) = delete;
24     Vec(Vec&& rhs) noexcept
25         : _begin(std::exchange(rhs._begin, nullptr)),
26         _end(std::exchange(rhs._end, nullptr)) {}
27
28     Vec& operator=(const Vec&) = delete;
29     Vec& operator=(Vec&& rhs) noexcept {
30         this->~Vec();
31         _begin = std::exchange(rhs._begin, nullptr);
32         _end = std::exchange(rhs._end, nullptr);
33         return *this;
34     }
35
36     ~Vec() { delete[] _begin; }
37
38     constexpr size_t size() const { return _end - _begin; }
39
40     constexpr double& operator[](size_t i) { return _begin[i]; }
41     constexpr double operator[](size_t i) const { return _begin[i]; }
42
43     void setSize(size_t n) {
44         if (size() == n) return;
45         *this = Vec(n);
46     }
47
48     friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
49         s << "(";
```

```

50     double* it = v._begin;
51     for (double* end = v._end - 1; it < end; ++it)
52         s << *it << ", ";
53
54     if (it < v._end) s << *it;
55
56     return s << " ";
57 }
58 static Vec read() {
59     Vec res(readSize("n"));
60     for (auto& v: res) std::cin >> v;
61     return res;
62 }
63 double norm() const;
64 };
65 void assertSizes(const Vec& a, const Vec& b) {
66     assert(a.size() == b.size(), "Sizes don't match");
67 }
68
69 Vec& add(const Vec& a, const Vec& b, Vec& res) {
70     assertSizes(a, b);
71     res.setSize(a.size());
72     auto aIt = a.begin();
73     auto bIt = b.begin();
74     for (auto& v : res) v = *(aIt++) + *(bIt++);
75     return res;
76 }
77
78 Vec& mul(double a, const Vec& b, Vec& res) {
79     res.setSize(b.size());
80     auto it = b.begin();
81     for (auto& v : res) v = a * (*(it++));
82     return res;
83 }
84
85 Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }
86
87 Vec& sub(const Vec& a, const Vec& b, Vec& res) {
88     return add(a, neg(b, res), res);
89 }
90 double dot(const Vec& a, const Vec& b) {
91     double res = 0;
92     assertSizes(a, b);
93     auto bIt = b.begin();
94     for (auto& v : a) res += v * (*(bIt++));
95     return res;
96 }
97 double norm(const Vec& a) {
98     return std::sqrt(dot(a, a));
99 }

```



```
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.

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

```
1  #include "utils.h"
2
3  #include "vector.h"
4  #include "matrix.h"
5
6  #include <iostream>
7  #include <cmath>
8
9  constexpr double eps = 1e-7;
10
11 Vec& mul(const Mat& m, const Vec& v, Vec& res) {
12     assert(v.begin() != res.begin(), "Can't multiply inplace");
13     assert(v.size() == size_t(m.n), "Sizes don't match");
14     res.setSize(m.m);
15
16
17     for (size_t i = 0; i < res.size(); ++i) {
18         res[i] = 0;
19         for (size_t k = 0; k < v.size(); ++k)
20             res[i] += m.at(i, k) * v[k];
21     }
22     return res;
23 }
24
25 // A * X = b
26 struct System {
27     Mat A;
28     Vec b;
29 }
```

```

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 {

```

```

80     Vec r;
81     mul(A, x, r);
82     sub(b, r, r);
83     for (auto& v : r) {
84         if (std::abs(v) > eps) return false;
85     }
86     return true;
87 }
88
89 static Vec solveCustom(int n) {
90     System s(n, n);
91     for (auto& v : s.A) v = 0;
92     s.A.at(0,0) = 2;
93     s.A.at(0,1) = -1;
94     s.b[0] = 1;
95     for (int i = 1; i < n - 1; ++i) {
96         s.b[i] = 1;
97
98         s.A.at(i,i-1) = -1;
99         s.A.at(i,i) = 2;
100        s.A.at(i,i+1) = -1;
101    }
102    s.b[n-1] = 1;
103    s.A.at(n-1, n-2) = -1;
104    s.A.at(n-1, n-1) = 2;
105    s.customTriangulate();
106    return s.solveTriangulated();
107 }
108 void customTriangulate() {
109     multiplyLine(0, 1 / A.at(0, 0));
110     for (int i = 1; i < A.m; ++i) {
111         addLines(i, 1, i-1);
112         multiplyLine(i, 1 / A.at(i, i));
113     }
114 }
115 };
116
117 int main() {
118     try {
119         int n = readSize("n", 2, 51);
120         std::cout << "x = " << System::solveCustom(n) << "\n";
121     } catch (std::exception& e) {
122         std::cerr << "Error" << e.what() << "\n";
123         return 1;
124     }
125     return 0;
126 }

```

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 corespunzătoare până la întâlnirea vocalei citite.

```
1  #include "utils.h"
2  #include "stack.h"
3
4  #include <iostream>
5  #include <cstdlib>
6
7  bool isVowel(char c) {
8      c = tolower(c);
9      return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
10 }
11
12 int main() {
13     std::string str;
14     std::cout << "str: ";
15     std::getline(std::cin, str);
16
17     char v;
18     do {
19         v = read<char>("vowel");
20     } while (!isVowel(v));
21
22     Stack<char> lower;
23     Stack<char> upper;
24     auto printStacks = [&] (const char* s) {
25         std::cout << s;
26         std::cout << "lower:" << lower << "\n";
27         std::cout << "upper:" << upper << "\n";
28     };
29
30     printStacks("Before:\n");
31     for (auto& c : str) {
32         if (islower(c)) lower.push(c);
33         else if (isupper(c)) upper.push(c);
34     }
35
```

```

36     printStacks("After Adding:\n");
37     Stack<char>& stack = isupper(v)? upper: lower;
38     while (!stack.empty()) {
39         if (char c = stack.pop(); c == v) {
40             break;
41         }
42     }
43     printStacks("Result:\n");
44     return 0;
45 }

```

11. Creați o listă liniară simplu înlănțuită în nodurile căreia sunt memorate numere naturale. Separați numerele naturale memorate în listă, în două liste, una corespunzătoare 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.

```

1  #include "utils.h"
2  #include "list.h"
3
4  #include <iostream>
5
6  int main() {
7      auto nums = List<int>::read("numbers");
8
9      auto [evens, odds] = partition_split(nums, [](int n) {return n % 2;});
10     std::cout << "odd: " << odds << "\n";
11     std::cout << "even: " << evens << "\n";
12
13     int x;
14     do {
15         std::cout << "x (must be even): ";
16         std::cin >> x;
17     } while (x % 2);
18
19     std::cout << "removing...\n";
20     evens.remove(x);
21
22     std::cout << "even: " << evens << "\n";
23     return 0;
24 }

```

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 în 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

```

1  #pragma once
2  #include "fixedPoint.h"

```

```

3  #include "list.h"
4
5  #include <iostream>
6  #include <iomanip>
7
8  constexpr size_t precision = 2;
9  using FP = FixedPoint<precision>;
10
11 template<class T>
12 constexpr void setMax(T& a, T b) {
13     if (b > a) a = b;
14 }
15 struct Product {
16     std::string name;
17     FP quantity;
18     std::string unit;
19     FP unitPrice;
20     constexpr auto totalPrice() const {
21         return quantity * unitPrice;
22     }
23 };
24
25 struct Stock {
26     List<Product> products;
27
28     //return value: was the operation was successful
29     bool add(Product p) {
30         auto* res = products.find(
31             [&] (Product& other) { return other.name == p.name; });
32         if (res != nullptr) {
33             auto& v = res->val;
34             std::cout << "Product '" << p.name << "' already exists.\n"
35                 << "Resupplying.\n";
36             if (v.unit != p.unit) {
37                 std::cout << "Units did not match ('"
38                     << v.unit << "' != '" << p.unit << "')\n";
39                 return false;
40             }
41             v.unitPrice += (v.quantity*v.unitPrice + p.quantity*p.unitPrice)
42                 / (v.quantity + p.quantity);
43             v.quantity += p.quantity;
44         } else {
45             products.push_front(p);
46         }
47         return true;
48     }
49     //return value: was the operation was successful
50     bool resupply(const std::string& name, FP quantity) {
51         auto* res = products.find(
52             [&] (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 //return 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     public:
101         static constexpr std::string_view fields[] = {
102             "Name", "Quantity", "Unit Price", "Total Price"

```



```

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 << "| ";

```

```

153         printPadded(longest.name, "");
154         std::cout << " | ";
155         printPadded(longest.quantity + 1 + longest.unit, "");
156         std::cout << " | ";
157         printPadded(longest.unitPrice, "");
158         std::cout << " | "<<std::right;
159         printPadded(longest.totalPrice, totalPrice);
160         std::cout << " |\n";
161         printLine();
162     }
163
164     private:
165         template<typename T>
166         static void printPadded(size_t len, T v) {
167             std::cout << std::setw(len) << v;
168         }
169         static void printCentered(size_t len, std::string_view s) {
170             //we can safely assume that len >= s.len();
171             int total = len - s.size();
172             int left = total / 2;
173             int right = total - left;
174             hline(left, ' ');
175             std::cout << std::setw(0) << s;
176             hline(right, ' ');
177         }
178
179         static void hline(int len, char c) {
180             while (--len >= 0)
181                 std::cout << c;
182         }
183
184         void printLine() const {
185             std::cout << "+";
186             hline(longest.name+2, '-');
187             std::cout << "+";
188             hline(longest.quantity + 3 + longest.unit, '-');
189             std::cout << "+";
190             hline(longest.unitPrice+2, '-');
191             std::cout << "+";
192             hline(longest.totalPrice+2, '-');
193             std::cout << "+\n";
194         }
195     };
196 };

```

stock.cpp

```

1  #include "stock.h"
2
3  #include "inputHelper.h"

```

```

4  #include "utils.h"
5  #include "list.h"
6
7  #include <iostream>
8  #include <iomanip>
9
10 struct Command {
11     char shortName;
12     std::string_view name;
13     std::string_view args;
14     bool (*f)(Stock& s, MultiInputHelper& ih); //returns true if should exit
15     std::string_view description;
16     void print() const {
17         std::cout << " " << std::setw(10) << std::left << name
18             << " - " << shortName << " "
19             << std::setw(25) << args << " - " << description << "\n";
20     }
21 };
22 using IH = MultiInputHelper;
23
24 void printHelp();
25 bool printHelp(Stock&, IH&) {
26     printHelp();
27     return true;
28 }
29 bool quit(Stock&, IH&) { return false; }
30 bool print(Stock& s, IH&) {
31     s.print();
32     return true;
33 }
34
35 bool add(Stock& s, IH& ih) {
36     Product p;
37     ih.readName(p.name, "name");
38     ih.readFP(p.quantity, "quantity");
39     ih.readString(p.unit, "unit");
40     ih.readFP(p.unitPrice, "unit price");
41     s.add(p);
42     return true;
43 }
44
45 bool sell(Stock& s, IH& ih) {
46     std::string name;
47     FP quantity;
48     ih.readName(name, "name");
49     ih.readFP(quantity, "quantity");
50
51     s.sell(name, quantity);
52     return true;
53 }

```

```

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& 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         }

```

```

104     }
105     for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i) {
106         if (cmds[i].name == s) return cmds[i].f(stock, ih);
107     }
108     std::cout << "Invalid command!\n";
109     printHelp();
110     std::cout << "!";
111     return true;
112 }
113
114 int main() {
115     Stock stock;
116     printHelp();
117     MultiInputHelper ih;
118     for (;;) {
119         if (!eval(stock, ih)) break;
120     }
121     return 0;
122 }

```

fixedPoint.h

```

1  #pragma once
2  #include "utils.h"
3  #include <iostream>
4
5  constexpr char decimalSeparator = '.';
6
7  constexpr long pow(long b, long e) {
8      long res = 1;
9      while (--e >= 0) res *= b;
10     return res;
11 }
12 template<size_t decimals = 2>
13 class FixedPoint {
14     static_assert(decimals < 18);
15     static constexpr long factor = pow(10, decimals);
16     int64_t val;
17     using FP = FixedPoint<decimals>;
18
19 public:
20     static constexpr FP make(int64_t v) {
21         FP r;
22         r.val = v;
23         return r;
24     }
25     constexpr FixedPoint(long double v) : val(v * factor) {}
26     constexpr FixedPoint(double v) : val(v * factor) {}
27     constexpr FixedPoint(float v) : val(v * factor) {}
28     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)

```

```

79     if (val < factor * 10000000000000) return baseline + 12;
80     if constexpr (decimals <= 5)
81     if (val < factor * 100000000000000) return baseline + 13;
82     if constexpr (decimals <= 4)
83     if (val < factor * 1000000000000000) return baseline + 14;
84     if constexpr (decimals <= 3)
85     if (val < factor * 10000000000000000) return baseline + 15;
86     if constexpr (decimals <= 2)
87     if (val < factor * 100000000000000000) return baseline + 16;
88     if constexpr (decimals <= 1)
89     if (val < factor * 1000000000000000000) return baseline + 17;
90     if constexpr (decimals <= 0)
91     if (val < factor * 10000000000000000000) return baseline + 18;
92     return baseline + (19 - decimals);
93 }
94
95 // a buffer of size at most 22 is needed
96  //(19 for digits, 1 for the dot, 1 for sign, and 1 for \0)
97 static constexpr size_t MaxBuffSize = 22;
98
99 // returns the string length (no \0)
100 constexpr size_t toString(char* buf, bool showSign) const {
101     auto len = textLen(showSign);
102     char* p = buf + len;
103     auto i = val;
104     if (val < 0) {
105         *buf = '-';
106         i = -val;
107     }
108     else if (showSign) *buf = '+';
109     *p-- = '\0';
110     char* decimalPoint = p - decimals;
111     for (; p != decimalPoint; --p) {
112         *p = (i % 10) + '0';
113         i /= 10;
114     }
115     *p = decimalSeparator;
116     while (i) {
117         *(--p) = (i % 10) + '0';
118         i /= 10;
119     }
120     return len;
121 }
122 static constexpr auto isDigit(char c) {
123     return c >= '0' && c <= '9';
124 }
125 // if the return value is nullptr it means we didn't read anything good
126 // else we return a pointer to the end of the read FixedPoint
127 static constexpr const char* fromString(const char* str, FP& res) {
128     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);

```



```

179         return s << buf;
180     }
181
182     friend std::istream& operator>>(std::istream& s, FP& fp) {
183         long double ld;
184         s >> ld;
185         fp = ld;
186         return s;
187     }
188 };

```

inputHelper.h

```

1  #pragma once
2  #include "fixedPoint.h"
3  #include <iostream>
4
5  class MultiInputHelper {
6  public:
7      std::string line;
8      const char* p;
9
10 public:
11     MultiInputHelper() {}
12     void getLine(std::string_view msg = "") {
13         if (msg!=""){
14             std::cout << msg;
15             if (msg.back() != ' ') std::cout << ": ";
16         }
17         std::getline(std::cin, line);
18         p = line.c_str();
19     }
20
21     void getLineAfterInvalid(std::string_view msg) {
22         std::cout << "!";
23         getLine(msg);
24     }
25     void readName(std::string& res, std::string_view msg) {
26         eatWhiteSpace(msg);
27         if (*p == '"') {
28             const char* beg = ++p;
29             res = "";
30             for (;;) {
31                 if (!*p) {
32                     getLine/*AfterInvalid*/(msg);
33                     return readName(res, msg);
34                 } else if (*p++ == '"') {
35                     if (*(p-2) == '\\') {
36                         res += std::string(beg, p-beg-2) + '"';
37                         beg = p;

```

```

38         } else {
39             res += std::string(beg, p-beg-1);
40             return;
41         }
42     }
43 }
44 } else {
45     readString(res, msg);
46 }
47 }
48
49 void readString(std::string& res, std::string_view msg) {
50     eatWhiteSpace(msg);
51     const char* beg = p;
52     while (*p && !isspace(*p)) {
53         ++p;
54     }
55     auto sz = beg-p;
56     if (sz == 0) {
57         getLine/*AfterInvalid*/(msg);
58         return readString(res, msg);
59     } else {
60         res = std::string(beg, p - beg);
61     }
62 }
63 char readChar(std::string_view msg = "") {
64     eatWhiteSpace(msg);
65     return *p++;
66 }
67
68 std::string_view readStringView(std::string_view msg) {
69     eatWhiteSpace(msg);
70     const char* beg = p;
71     while (*p && !isspace(*p)) ++p;
72
73     auto sz = beg-p;
74     if (sz == 0) {
75         getLineAfterInvalid(msg);
76         return readStringView(msg);
77     } else {
78         return std::string_view(beg, p - beg);
79     }
80 }
81 template<size_t precision>
82 void readFP(FixedPoint<precision>& res, std::string_view msg) {
83     eatWhiteSpace(msg);
84     auto str = FixedPoint<precision>::fromString(p, res);
85     if (str == nullptr) {
86         getLineAfterInvalid(msg);
87         readFP(res, msg);

```

```

88         return;
89     } else {
90         p = str;
91     }
92 }
93
94 private:
95     void eatWhiteSpace(std::string_view message) {
96         for (;;) {
97             if (*p == 0) getLine(message);
98             if (isspace(*p)) ++p;
99             else return;
100         }
101     }
102 };

```

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.

```
1  #include "doubleList.h"
2  #include "utils.h"
3
4  #include <iostream>
5
6  struct Student {
7      std::string firstName;
8      std::string lastName;
9      double grades[3];
10     double average() const {
11         return ::average(grades, grades+3);
12     }
13     friend std::ostream& operator <<(std::ostream& os, const Student& s) {
14         return os << s.firstName << " " << s.lastName
15             << " - average: " << s.average();
16     }
17
18     friend std::istream& operator >>(std::istream& stream, Student& s) {
19         return stream >> s.firstName >> s.lastName
20             >> s.grades[0] >> s.grades[1] >> s.grades[2];
21     }
22 };
23
24 void printStudents(const DoubleList<Student>& students) {
25     std::cout << "students:\n";
26     for (auto& s : students) std::cout << s << "\n";
27 }
28 double getAverage(const DoubleList<Student>& students) {
29     double res = 0;
30     int count = 0;
31     for (auto& s : students) {
32         res += s.average();
33         ++count;
34     }
35     return res/count;
36 }
37
```

```

38 int main() {
39     auto students = DoubleList<Student>::read("Students");
40     printStudents(students);
41     std::cout << "Average: " << getAverage(students) << "\n";
42
43     return 0;
44 }

```

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

- Scrieți o funcție care primește ca parametru adresa primului nod al listei și o afișează în ambele sensuri.
- 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*.
- Scrieți o funcție care primește ca parametru adresa *p* a unui nod și șterge nodul indicat de *p*.

```

1  #include "doubleList.h"
2  #include "utils.h"
3
4  #include <iostream>
5
6  template<typename T>
7  void printBothWays(typename DoubleList<T>::Node *n) {
8      if (n == nullptr) return;
9      typename DoubleList<T>::It it(n);
10     std::cout << "Forward:\n";
11
12     while (it.hasNext()) {
13         std::cout << *it++ << ", ";
14     }
15     std::cout << *it << "\n";
16     std::cout << "Reverse:\n";
17     while (it.hasPrev()) {
18         std::cout << *it << ", ";
19         --it;
20     }
21     std::cout << *it << "\n";
22 }
23
24 template<typename T>
25 void insert(typename DoubleList<T>::Node *n, T val) {
26     using Node = typename DoubleList<T>::Node;
27     Node* newN = new Node{ val, n->next, n};
28     n->next->prev = newN;
29     n->next = newN;
30 }
31
32 template<typename T>
33 void remove(typename DoubleList<T>::Node *n) {
34     assert(n->prev && n->next, "Please call list.remove(n).");

```

```

35     n->prev->next = n->next;
36     n->next->prev = n->prev;
37 }
38
39 int main() {
40     DoubleList<int> list = {1, 2, 5, 4 };
41     insert(list.top->next, 3);
42     std::cout << list << "\n";
43     remove<int>(list.bot->prev);
44
45     printBothWays<int>(list.top);
46
47     return 0;
48 }

```

Grafuri neorientate

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

```
1  #include "utils.h"
2  #include "list.h"
3
4  #include <iostream>
5  #include <fstream>
6
7  struct Graph_EdgeList {
8      using Vertex = unsigned;
9      struct Edge {
10         Vertex a, b;
11         constexpr Edge(Vertex a, Vertex b) :a(a), b(b) {}
12         friend std::ostream& operator<< (std::ostream& s, Edge e) {
13             return s << "(" << e.a << ", " << e.b << ")";
14         }
15
16         friend std::istream& operator>> (std::istream& s, Edge& e) {
17             return s >> e.a >> e.b;
18         }
19         constexpr bool operator==(Edge rhs) const {
20             return (rhs.a == a && rhs.b == b) ||
21                    (rhs.b == a && rhs.a == b);
22         }
23     };
24     List<Edge> edges;
25     size_t nodeSize;
26
27     template <typename Path>
28     static Graph_EdgeList fromFile(Path path) {
29         std::ifstream f(path);
30         size_t sz = 0;
31         f >> sz;
32         Graph_EdgeList g { {}, sz };
33         Vertex a, b;
34         while (f >> a >> b) { g.addEdge(a, b); }
35         return g;
36     }
37     void addEdge(Vertex a, Vertex b) { edges.emplace_front(a, b); }
38     void addEdge(Edge e) { edges.push_front(e); }
39 }
```

```

40     friend std::ostream& operator<<(std::ostream& s,
41                                     const Graph_EdgeList& g) {
42         return s << g.edges;
43     }
44     constexpr bool hasEdge(Edge e) const {
45         return edges.findElem(e) != nullptr;
46     }
47
48     constexpr bool hasEdge(Vertex a, Vertex b) const {
49         return hasEdge({a, b});
50     }
51 };
52
53 int main() {
54     auto g = Graph_EdgeList::fromFile("nodes.txt");
55     std::cout << "g:" << g << "\n";
56     std::cout << "has: " << g.hasEdge(3, 2);
57     return 0;
58 }

```

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

```

1  #include "utils.h"
2  #include "graphAdjacencyList.h"
3
4  #include <iostream>
5
6  using Graph = Graph_AdjacencyList;
7
8  // A simple path or a simple cycle is a path or cycle that has no
9  // repeated vertices and consequently no repeated edges.
10 void printSimplePaths(const Graph& g, PtrRange<Graph::Vertex> verts) {
11     for (auto v : g[verts.back()]) {
12         //if (verts.front() == v) {
13         //*verts.last++ = v;
14         // continue;
15         // } else
16         if (verts.contains(v)) { continue; }
17         auto res = PtrRange<Graph::Vertex>(verts.first, verts.last+1);
18         *verts.last = v;
19         std::cout << res << "\n";
20         printSimplePaths(g, res);
21     }
22 }
23
24 void printSimplePaths(const Graph& g) {
25     // I assume that a node is not a simple path.
26     std::cout << "Simple Paths: \n";
27     if (g.edgeCount() == 0) {
28         std::cout << "No simple paths found\n";
29         return;
30     }

```



```

29     }
30     using Vertex = Graph::Vertex;
31     Vertex* verts = (Vertex*) alloca(sizeof(Vertex) * g.vertCount());
32     for (Vertex v = 0; v < g.vertCount(); ++v) {
33         verts[v] = v;
34         printSimplePaths(g, PtrRange<Vertex>(verts, v));
35     }
36 }
37
38 int main() {
39     auto g = Graph::fromFile("nodes.txt");
40     std::cout << "g:\n" << g << "\n";
41     printSimplePaths(g);
42     return 0;
43 }

```

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

```

1  #include "utils.h"
2  #include "graphEdgeList.h"
3
4  #include <iostream>
5
6  using Graph = Graph_EdgeList;
7
8  using Vertex = Graph::Vertex;
9  using Edge = Graph::Edge;
10 // has a cycle that visits every edge once
11 bool isEulerian(Edge* begin,
12                 Edge* current,
13                 Edge* end) {
14     /*
15      edges right of current are unvisited,
16      edges on the left form a chain
17     */
18     if (end - current == 0) {
19         if (begin->a == (end-1)->b) {
20             std::cout << "Is Eulerian With solution: ";
21             for (auto* it = begin; it != end; ++it) {
22                 std::cout << it->a << " ";
23             }
24             std::cout << (end-1)->b;
25             std::cout << "\n";
26             return true;
27         }
28     }
29     auto last = (current-1)->b;
30     for (auto* it = current; it != end; ++it) {
31         if (it->a == last) ;

```

```

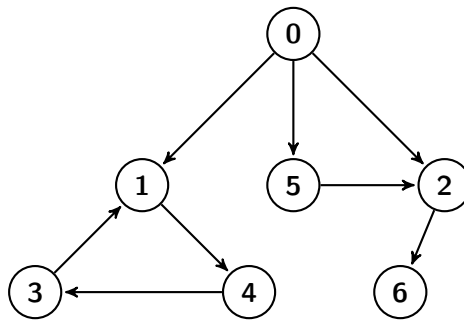
32         else if (it->b == last) { std::swap(it->b, it->a); }
33         else continue;
34         std::swap(*current, *it);
35         if (isEulerian(begin, current+1, end)) {
36             return true;
37         }
38         std::swap(*current, *it);
39     }
40     return false;
41 }
42 void printIsEulerian(const 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.



```
1  #include "utils.h"
2  #include "dirGraphMat.h"
3  #include "dirGraphAdjacencyList.h"
4
5  #include <iostream>
6
7  void visitDepthFirst(const DirGraph_Mat& g,
8                      DirGraph_Mat::Vertex current,
9                      DirGraph_Mat::Vertex* visited) {
10     std::cout << current << " ";
11     visited[current] = true;
12     auto line = g[current];
13     for (size_t i = 0; i < g.size; ++i) {
14         if (line[i]) {
15             if (visited[i]) continue;
16             visitDepthFirst(g, i, visited);
17         }
18     }
19 }
20
21 void visitDepthFirst(const DirGraph_Mat& g) {
22     using Graph = DirGraph_Mat;
23     using Vertex = Graph::Vertex;
24
25     Vertex* visited = (Vertex*)alloca(sizeof(Vertex) * g.size);
26     for (Vertex i = 0; i < g.size; ++i) {
27         std::cout << "Nodes from " << i << ":\n";
```

```

28         for (Vertex j = 0; j <= g.size; ++j) visited[j] = 0;
29         visitDepthFirst(g, i, visited);
30         std::cout << "\n";
31     }
32     std::cout << "\n";
33 }
34
35 template<typename T>
36 struct SimpleQueue {
37     T* first;
38     T* last;
39     constexpr void push(const T& val) {
40         *(last++) = val;
41     }
42
43     T pop() {
44         assert(first < last, "Queue empty");
45         return *(first++);
46     }
47     constexpr bool empty() const {
48         return first == last;
49     }
50 };
51
52 void visitBreadthFirst(const DirGraph_AdjacencysList& g,
53                       DirGraph_AdjacencysList::Vertex current,
54                       SimpleQueue<DirGraph_AdjacencysList::Vertex>& queue,
55                       DirGraph_AdjacencysList::Vertex* visited) {
56     using Graph = DirGraph_AdjacencysList;
57     using Vertex = Graph::Vertex;
58     queue.push(current);
59     visited[current] = true;
60     while (!queue.empty()) {
61         Vertex i = queue.pop();
62
63         std::cout << i << " ";
64         for (auto& v : g[i]) {
65             if (visited[v]) continue;
66             visited[v] = true;
67             queue.push(v);
68         }
69     }
70 }
71 void visitBreadthFirst(const DirGraph_AdjacencysList& g) {
72     using Graph = DirGraph_Mat;
73     using Vertex = Graph::Vertex;
74
75     Vertex* visited = (Vertex*)alloca(sizeof(Vertex) * g.vertCount());
76     Vertex* queueData = (Vertex*)alloca(sizeof(Vertex) * g.arcCount());
77     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.

```
1  #include "weightedDirGraphMat.h"
2  #include <iostream>
3  #include <iomanip>
4
5  using Graph = WeightedDirGraph_Mat;
6  using Vertex = Graph::Vertex;
7  using Weight = Graph::Weight;
8
9  constexpr Vertex invalidVert = -1;
10
11 template<typename T>
12 using Matrix = ManagedMatrixPtrRange<T>;
13
14 struct DijkstraRes {
15     ManagedPtrRange<Weight> dist;
16     ManagedPtrRange<Vertex> prev;
17     Vertex from;
18
19     bool printPath(Vertex u) {
20         StaticVector<Vertex> s(prev.size());
21         while (u != from) {
22             if (prev[u] == invalidVert) return false;
23             s.push_back(u);
24             u = prev[u];
25         }
26         std::cout << from << " ";
27         for (auto it = s.last-1; it > s.first -1; --it) {
28             std::cout << *it << " ";
29         }
30         std::cout << "\n";
31         return true;
32     }
33 }
```

```

32     }
33 };
34 auto dijkstra(const Graph& g, Vertex from) {
35     DijkstraRes res = { {g.size()}, { g.size()} , from};
36     StaticVector<Vertex> q(g.size(), g.size());
37     for (Vertex v = 0; v < g.size(); ++v) {
38         res.dist[v] = Graph::inf;
39         res.prev[v] = invalidVert;
40         q[v] = v;
41     }
42     res.dist[from] = 0;
43
44     while (q.size() != 0) {
45         auto it = q.min();
46         auto u = *it;
47         q.remove(it);
48         for (Vertex v = 0; v < g.size(); ++v) {
49             if (v == u) continue;
50             auto val = g.mat(u, v);
51             if (val != Graph::inf) {
52                 auto alt = res.dist[u] + val;
53                 if (alt < res.dist[v]) {
54                     res.dist[v] = alt;
55                     res.prev[v] = u;
56                 }
57             }
58         }
59     }
60     return res;
61 }
62
63 struct RoyFloydRes {
64     Matrix<Weight> dist;
65     Matrix<Vertex> next;
66
67     bool printPath(Vertex u, Vertex v) {
68         if (next(u, v) == invalidVert) {
69             std::cout << "No path\n";
70             return false;
71         }
72         while (u != v) {
73             std::cout << u << " ";
74             u = next(u, v);
75         }
76         std::cout << v << "\n";
77         return true;
78     }
79 };
80
81 RoyFloydRes royFloyd(const Graph& g) {

```

```

82     auto n = g.size();
83     RoyFloydRes res = {g.mat, {n,n} };
84
85     for (Vertex u = 0; u < n; ++u) {
86         for (Vertex v = 0; v < n; ++v) {
87             res.next(u, v) = res.dist(u, v) != Graph::inf? v : invalidVert;
88         }
89     }
90     for (Vertex k = 0; k < n; ++k)
91         for (Vertex i = 0; i < n; ++i)
92             for (Vertex j = 0; j < n; ++j) {
93                 if (res.dist(i, j) > res.dist(i, k) + res.dist(k, j)) {
94                     res.dist(i, j) = res.dist(i, k) + res.dist(k, j);
95                     res.next(i, j) = res.next(i, k);
96                 }
97             }
98     return res;
99 }
100
101 int main() {
102     Graph g = Graph::fromFile("nodes.txt");
103     std::cout << std::setw(3) << g;
104
105     auto s = read<Vertex>("Start");
106     auto e = read<Vertex>("End");
107     auto dij = dijkstra(g, s);
108     dij.printPath(e);
109
110     std::cout << "\nR-F:\n";
111
112     auto res = royFloyd(g);
113     std::cout << std::setw(3) << res.dist;
114     std::cout << "Next:\n" << res.next << "\n";
115     res.printPath(1, 2);
116
117     auto res2 = dijkstra(g, 1);
118
119     res2.printPath(2);
120     return 0;
121 }

```