

Alex's Anthology of Algorithms

Common Code for Contests in Concise C++

Draft v0.9, Feb 16, 2020

Alex Li

© 2015—. All rights reserved.

This page is intentionally left blank.

Preface

Visit github.com/alxli/algorithm-anthology for the most up-to-date digital version of this book.

Introduction

Welcome to a comprehensive collection of common algorithms and data structures. The ultimate goal of this book is not to explain concepts from the ground up, but instead to explore the finer details behind their *implementations*. There are many potential ways you can use this, for instance:

- as a reference to help you better understand topics that you have only studied on a high level,
- as a printed codebook, which is a permitted resource for contests such as the ACM ICPC, or
- to cross-check existing code you have written for contest or coding interview questions.

Before diving into any section, it is strongly recommended that you have already studied the algorithms involved. Reading the code first is never an ideal approach to properly understand an algorithm. You should instead try proving (its correctness and time/space complexity) and implementing it from scratch.

Every topic to be explored is easily researchable online. Thus instead of including theoretical discussions, I document just enough to establish the problem being solved, notation being used, and any special trickery involved. I have also included small, non-rigorous examples to demonstrate usage of the code.

We mentioned that the implementation itself is the focus, but what makes an implementation good? The code is written with the following principles in mind:

- *Clarity*: A reader already familiar with an algorithm should have no problem understanding how its implementation works. Consistency in naming conventions should be emphasized, and any tricks or language-specific hacks should be documented.
- *Concision*: To minimize the amount of scrolling and searching during the frenzy of time contests, it is helpful for code to be compact. Shorter code is also generally easier to understand, as long as it is not overly cryptic. Finally, each implementation should fit in a single source file as required by nearly all online judging systems.
- *Efficiency*: The code here is designed to be performant on real contests, and should maintain a low constant overhead. This is often challenging in the face of clarity and tweakability, but we can hope for contest setters to be liberal with time and memory limits. If the code here times out, you can reasonably rule out insufficient constant optimization and assume that you are choosing an algorithm from a suboptimal complexity class.

- *Genericness:* It should be easy to adapt the implementation to achieve slightly different goals. This could be through tweaking some core logic, parameters, data types, etc. If we were tweaking the code to achieve a different purpose during a live contest, we would certainly want the process to be as painless as possible. C++ templates are often used to increase tweakability at the slight cost of simplicity.
- *Portability:* Different contest environments use different versions of C++ (though almost all of them use GCC). In order to maximize compatibility, non-standard features are avoided. The decision to adhere to C++98 standards is due to many contest systems being stuck on an older version of the language. Moreover, minimizing newer C++ specific features will make the implementations more language-agnostic. Refer to the "Portability" section below.

As these points and the title both suggest, there is a slight bias towards contests. Compiling a codebook for my personal reference during contests was indeed how this project got started. This work has become much more multipurpose now; whatever your use case is, I hope you discover something enlightening.

Cheers.
— Alex

Portability Note

All programs were tested with GCC and compiled for a 32-bit target using the switches below:

```
g++ -std=gnu++98 -pedantic -Wall -Wno-long-long -O2
```

This means the following are assumed about data types:

- `bool` and `char` are 8-bit.
- `int` and `float` are 32-bit.
- `double` and `long long` are 64-bit.
- `long double` is 96-bit.

Programs are highly portable (ISO C++ 1998 compliant), **except** in the following regards:

- Usage of `long long` and dependent features, which are compliant in C99/C++0x or later. 64-bit integers are a must for many contest problems.
- Usage of variable sized arrays. While easily replaced by vectors, they are generally simpler and avoid dynamic memory (which some argue is a bad idea for contests).
- Usage of GCC's built-in functions like `__builtin_popcount()` and `__builtin_clz()`. These can be extremely convenient, but are straightforward to implement if unavailable. See here for a reference: <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>
- Usage of compound-literals, e.g. `vec.push_back((mystruct){a, b, c})`. This adds a little more concision by not requiring a constructor definition.
- Hacks that may depend on the platform (e.g. endianness), such as getting the signbit with type-punned pointers. Be weary of portability for all bitwise/lower level code.

Contents

Preface	ii
1 Elementary Algorithms	1
1.1 Array Transformations	1
1.1.1 Sorting Algorithms	1
1.1.2 Array Rotation	8
1.1.3 Counting Inversions	10
1.1.4 Coordinate Compression	12
1.1.5 Selection (Quickselect)	14
1.2 Array Queries	15
1.2.1 Longest Increasing Subsequence	15
1.2.2 Maximal Subarray Sum (Kadane)	16
1.2.3 Majority Element (Boyer-Moore)	19
1.2.4 Subset Sum (Meet-in-the-Middle)	20
1.2.5 Maximal Zero Submatrix	21
1.3 Searching	23
1.3.1 Binary Search	23
1.3.2 Ternary Search	25
1.3.3 Hill Climbing	26
1.3.4 Convex Hull Trick (Semi-Dynamic)	28
1.3.5 Convex Hull Trick (Fully-Dynamic)	29
1.4 Cycle Detection	32
1.4.1 Cycle Detection (Floyd)	32
1.4.2 Cycle Detection (Brent)	34
2 Data Structures	36
2.1 Heaps	36

2.1.1	Binary Heap	36
2.1.2	Randomized Mergeable Heap	38
2.1.3	Skew Heap	41
2.1.4	Pairing Heap	43
2.2	Dictionaries	46
2.2.1	Binary Search Tree	46
2.2.2	Treap	50
2.2.3	AVL Tree	53
2.2.4	Red-Black Tree	58
2.2.5	Splay Tree	64
2.2.6	Size Balanced Tree	68
2.2.7	Interval Treap	73
2.2.8	Hash Map	78
2.2.9	Skip List	81
2.3	Range Queries in One Dimension	85
2.3.1	Sparse Table (Range Minimum Query)	85
2.3.2	Square Root Decomposition	86
2.3.3	Segment Tree (Point Update)	89
2.3.4	Segment Tree (Range Update)	91
2.3.5	Segment Tree (Compressed)	95
2.3.6	Implicit Treap	99
2.4	Range Queries in Two Dimensions	104
2.4.1	Quadtree (Point Update)	104
2.4.2	Quadtree (Range Update)	108
2.4.3	2D Segment Tree	112
2.4.4	2D Range Tree	117
2.4.5	K-d Tree (2D Range Query)	119
2.4.6	K-d Tree (Nearest Neighbor)	122
2.4.7	R-Tree (Nearest Segment)	124
2.5	Fenwick Trees	127
2.5.1	Fenwick Tree (Simple)	127
2.5.2	Fenwick Tree (Range Update, Point Query)	129
2.5.3	Fenwick Tree (Point Update, Range Query)	130
2.5.4	Fenwick Tree (Range Update, Range Query)	132

2.5.5	Fenwick Tree (Compressed)	134
2.5.6	2D Fenwick Tree (Simple)	136
2.5.7	2D Fenwick Tree (Compressed)	137
2.6	Tree Data Structures	140
2.6.1	Disjoint Set Forest (Simple)	140
2.6.2	Disjoint Set Forest (Compressed)	142
2.6.3	Lowest Common Ancestor (Sparse Table)	144
2.6.4	Lowest Common Ancestor (Segment Tree)	146
2.6.5	Heavy Light Decomposition	148
2.6.6	Link-Cut Tree	153
3	Strings	160
3.1	String Utilities	160
3.2	Expression Parsing	165
3.2.1	String Searching (KMP)	165
3.2.2	String Searching (Z Algorithm)	166
3.2.3	String Searching (Aho-Corasick)	168
3.3	String Searching	170
3.3.1	Recursive Descent Parsing (Simple)	170
3.3.2	Recursive Descent Parsing (Generic)	171
3.3.3	Shunting Yard Parsing	176
3.4	Dynamic Programming	181
3.4.1	Longest Common Substring	181
3.4.2	Longest Common Subsequence	182
3.4.3	Sequence Alignment	184
3.5	Suffix Array and LCP	187
3.5.1	Suffix Array and LCP (Manber-Myers)	187
3.5.2	Suffix Array and LCP (Counting Sort)	189
3.5.3	Suffix Array and LCP (Linear DC3)	192
3.6	String Data Structures	195
3.6.1	Trie	195
3.6.2	Radix Tree	199
4	Graphs	205
4.1	Depth-First Search	205

4.1.1	Graph Class and Depth-First Search	205
4.1.2	Topological Sorting (DFS)	208
4.1.3	Eulerian Cycles (DFS)	209
4.1.4	Unweighted Tree Centers (DFS)	212
4.2	Shortest Path	214
4.2.1	Shortest Path (BFS)	214
4.2.2	Shortest Path (Dijkstra)	215
4.2.3	Shortest Path (Bellman-Ford)	217
4.2.4	Shortest Path (Floyd-Warshall)	219
4.3	Connectivity	221
4.3.1	Strongly Connected Components (Kosaraju)	221
4.3.2	Strongly Connected Components (Tarjan)	223
4.3.3	Bridges, Cut-points, and Biconnectivity	225
4.4	Minimum Spanning Tree	228
4.4.1	Minimum Spanning Tree (Prim)	228
4.4.2	Minimum Spanning Tree (Kruskal)	230
4.5	Maximum Flow	231
4.5.1	Maximum Flow (Ford-Fulkerson)	231
4.5.2	Maximum Flow (Edmonds-Karp)	233
4.5.3	Maximum Flow (Dinic)	234
4.5.4	Maximum Flow (Push-Relabel)	236
4.6	Maximum Matching	238
4.6.1	Maximum Bipartite Matching (Kuhn)	238
4.6.2	Maximum Bipartite Matching (Hopcroft-Karp)	240
4.6.3	Maximum Graph Matching (Edmonds)	242
4.7	Hard Problems	245
4.7.1	Maximum Clique (Bron-Kerbosch)	245
4.7.2	Graph Coloring	247
4.7.3	Shortest Hamiltonian Cycle (TSP)	249
4.7.4	Shortest Hamiltonian Path	251
5	Mathematics	253
5.1	Math Utilities	253
5.2	Combinatorics	260
5.2.1	Combinatorial Calculations	260

5.2.2	Enumerating Arrangements	264
5.2.3	Enumerating Permutations	267
5.2.4	Enumerating Combinations	271
5.2.5	Enumerating Partitions	276
5.2.6	Enumerating Generic Combinatorial Sequences	279
5.3	Number Theory	283
5.3.1	GCD, LCM, Mod Inverse, Chinese Remainder	283
5.3.2	Prime Generation	286
5.3.3	Primality Testing	288
5.3.4	Integer Factorization	290
5.3.5	Euler's Totient Function	296
5.3.6	Binary Exponentiation	297
5.4	Arbitrary Precision Arithmetic	298
5.4.1	Big Integers (Simple)	298
5.4.2	Big Integers	302
5.4.3	Rational Numbers	313
5.5	Linear Algebra	318
5.5.1	Matrix Utilities	318
5.5.2	Row Reduction	325
5.5.3	Determinant and Inverse	327
5.5.4	LU Decomposition	330
5.5.5	Linear Programming (Simplex)	334
5.6	Root Finding and Calculus	337
5.6.1	Root Finding (Bracketing)	337
5.6.2	Root Finding (Iteration)	338
5.6.3	Polynomial Root Finding (Differentiation)	340
5.6.4	Polynomial Root Finding (Laguerre)	342
5.6.5	Polynomial Root Finding (RPOLY)	344
5.6.6	Integration (Simpson)	354
6	Geometry	356
6.1	Geometry Library in One File	356
6.1	Geometric Classes	373
6.1.1	Point	373
6.1.2	Line	376

6.1.3	Circle	379
6.1.4	Triangle	381
6.1.5	Rectangle	383
6.2	Elementary Geometric Calculations	386
6.2.1	Angles	386
6.2.2	Distances	388
6.2.3	Line Intersection	391
6.2.4	Circle Intersection	394
6.3	Intermediate Geometric Calculations	399
6.3.1	Polygon Sorting and Area	399
6.3.2	Point-in-Polygon (Ray Casting)	401
6.3.3	Convex Hull and Diametral Pair	403
6.3.4	Minimum Enclosing Circle	405
6.3.5	Closest Pair	407
6.3.6	Segment Intersection Finding	409
6.4	Advanced Geometric Computations	413
6.4.1	Convex Polygon Cut	413
6.4.2	Polygon Intersection and Union	415
6.4.3	Delaunay Triangulation (Simple)	419
6.4.4	Delaunay Triangulation (Fast)	422

Chapter 1

Elementary Algorithms

1.1 Array Transformations

1.1.1 Sorting Algorithms

```
1  /*
2
3 These functions are equivalent to std::sort(), taking random-access iterators
4 as a range [lo, hi) to be sorted. Elements between lo and hi (including the
5 element pointed to by lo but excluding the element pointed to by hi) will be
6 sorted into ascending order after the function call. Optionally, a comparison
7 function object specifying a strict weak ordering may be specified to replace
8 the default operator <.
9
10 These functions are not meant to compete with standard library implementations
11 in terms of speed. Instead, they are meant to demonstrate how common
12 sorting algorithms can be concisely implemented in C++.
13 */
14
15
16 #include <algorithm>
17 #include <functional>
18 #include <iterator>
19 #include <vector>
20
21 /*
22 Quicksort repeatedly selects a pivot and partitions the range so that elements
23 comparing less than the pivot precede the pivot, and elements comparing greater
24 or equal follow it. Divide and conquer is then applied to both sides of the
25 pivot until the original range is sorted. Despite having a worst case of O(n^2),
26 quicksort is often faster in practice than merge sort and heapsort, which both
27 have a worst case time complexity of O(n log n).
28
29 The pivot chosen in this implementation is always a middle element of the range
30 to be sorted. To reduce the likelihood of encountering the worst case, the pivot
31 can be chosen in better ways (e.g. randomly, or using the "median of three"
32 technique).
33
34
```

```

35 Time Complexity (Average): O(n log n).
36 Time Complexity (Worst): O(n^2).
37 Space Complexity: O(log n) auxiliary stack space.
38 Stable?: No.
39
40 */
41
42 template<class It, class Compare>
43 void quicksort(It lo, It hi, Compare comp) {
44     if (hi - lo < 2) {
45         return;
46     }
47     typedef typename std::iterator_traits<It>::value_type T;
48     T pivot = *(lo + (hi - lo)/2);
49     It i, j;
50     for (i = lo, j = hi - 1; ; ++i, --j) {
51         while (comp(*i, pivot)) {
52             ++i;
53         }
54         while (comp(pivot, *j)) {
55             --j;
56         }
57         if (i >= j) {
58             break;
59         }
60         std::swap(*i, *j);
61     }
62     quicksort(lo, i, comp);
63     quicksort(i, hi, comp);
64 }
65
66 template<class It>
67 void quicksort(It lo, It hi) {
68     typedef typename std::iterator_traits<It>::value_type T;
69     quicksort(lo, hi, std::less<T>());
70 }
71
72 /*
73
74 Merge sort first divides a list into n sublists of one element each, then
75 recursively merges the sublists into sorted order until only a single sorted
76 sublist remains. Merge sort is a stable sort, meaning that it preserves the
77 relative order of elements which compare equal by operator < or the custom
78 comparator given.
79
80 An analogous function in the C++ standard library is std::stable_sort(), except
81 that the implementation here requires sufficient memory to be available. When
82 O(n) auxiliary memory is not available, std::stable_sort() falls back to a time
83 complexity of O(n log^2 n) whereas the implementation here will simply fail.
84
85 Time Complexity (Average): O(n log n).
86 Time Complexity (Worst): O(n log n).
87 Space Complexity: O(log n) auxiliary stack space and O(n) auxiliary heap space.
88 Stable?: Yes.
89
90 */
91
92 template<class It, class Compare>
93 void mergesort(It lo, It hi, Compare comp) {

```

```

94     if (hi - lo < 2) {
95         return;
96     }
97     It mid = lo + (hi - lo - 1)/2, a = lo, c = mid + 1;
98     mergesort(lo, mid + 1, comp);
99     mergesort(mid + 1, hi, comp);
100    typedef typename std::iterator_traits<It>::value_type T;
101    std::vector<T> merged;
102    while (a <= mid && c < hi) {
103        merged.push_back(comp(*c, *a) ? *c++ : *a++);
104    }
105    if (a > mid) {
106        for (It it = c; it < hi; ++it) {
107            merged.push_back(*it);
108        }
109    } else {
110        for (It it = a; it <= mid; ++it) {
111            merged.push_back(*it);
112        }
113    }
114    for (int i = 0; i < hi - lo; i++) {
115        *(lo + i) = merged[i];
116    }
117 }
118
119 template<class It>
120 void mergesort(It lo, It hi) {
121     typedef typename std::iterator_traits<It>::value_type T;
122     mergesort(lo, hi, std::less<T>());
123 }
124
125 /*
126
127 Heapsort first rearranges an array to satisfy the max-heap property. Then, it
128 repeatedly pops the max element of the heap (the left, unsorted subrange),
129 moving it to the beginning of the right, sorted subrange until the entire range
130 is sorted. Heapsort has a better worst case time complexity than quicksort and
131 also a better space complexity than merge sort.
132
133 The C++ standard library equivalent is calling std::make_heap(lo, hi), followed
134 by std::sort_heap(lo, hi).
135
136 Time Complexity (Average): O(n log n).
137 Time Complexity (Worst): O(n log n).
138 Space Complexity: O(1) auxiliary.
139 Stable?: No.
140
141 */
142
143 template<class It, class Compare>
144 void heapsort(It lo, It hi, Compare comp) {
145     typename std::iterator_traits<It>::value_type tmp;
146     It i = lo + (hi - lo)/2, j = hi, parent, child;
147     for (;;) {
148         if (i <= lo) {
149             if (--j == lo) {
150                 return;
151             }
152             tmp = *j;

```

```

153     *j = *lo;
154 } else {
155     tmp = *(--i);
156 }
157 parent = i;
158 child = lo + 2*(i - lo) + 1;
159 while (child < j) {
160     if (child + 1 < j && comp(*child, *(child + 1))) {
161         child++;
162     }
163     if (!comp(tmp, *child)) {
164         break;
165     }
166     *parent = *child;
167     parent = child;
168     child = lo + 2*(parent - lo) + 1;
169 }
170 *(lo + (parent - lo)) = tmp;
171 }
172 }
173
174 template<class It>
175 void heapsort(It lo, It hi) {
176     typedef typename std::iterator_traits<It>::value_type T;
177     heapsort(lo, hi, std::less<T>());
178 }
179
180 /*
181 Comb sort is an improved bubble sort. While bubble sort increments the gap
182 between swapped elements for every inner loop iteration, comb sort fixes the gap
183 size in the inner loop, decreasing it by a particular shrink factor in every
184 iteration of the outer loop. The shrink factor of 1.3 is empirically determined
185 to be the most effective.
186
187 Even though the worst case time complexity is  $O(n^2)$ , a well chosen shrink
188 factor ensures that the gap sizes are co-prime, in turn requiring astronomically
189 large  $n$  to make the algorithm exceed  $O(n \log n)$  steps. On random arrays, comb
190 sort is only 2-3 times slower than merge sort. Its short code length length
191 relative to its good performance makes it a worthwhile algorithm to remember.
192
193 Time Complexity (Worst):  $O(n^2)$ .
194 Space Complexity:  $O(1)$  auxiliary.
195 Stable?: No.
196
197 */
198
199
200 template<class It, class Compare>
201 void combsort(It lo, It hi, Compare comp) {
202     int gap = hi - lo;
203     bool swapped = true;
204     while (gap > 1 || swapped) {
205         if (gap > 1) {
206             gap = (int)((double)gap / 1.3);
207         }
208         swapped = false;
209         for (It it = lo; it + gap < hi; ++it) {
210             if (comp(*(it + gap), *it)) {
211                 std::swap(*it, *(it + gap));

```

```

212         swapped = true;
213     }
214 }
215 }
216 }
217
218 template<class It>
219 void combsort(It lo, It hi) {
220     typedef typename std::iterator_traits<It>::value_type T;
221     combsort(lo, hi, std::less<T>());
222 }
223
224 /*
225
226 Radix sort is used to sort integer elements with a constant number of bits in
227 linear time. This implementation only works on ranges pointing to unsigned
228 integer primitives. The elements in the input range do not strictly have to be
229 unsigned types, as long as their values are nonnegative integers.
230
231 In this implementation, a power of two is chosen to be the base for the sort
232 so that bitwise operations can be easily used to extract digits. This avoids the
233 need to use modulo and exponentiation, which are much more expensive operations.
234 In practice, it's been demonstrated that  $2^8$  is the best choice for sorting
235 32-bit integers (approximately 5 times faster than std::sort(), and typically
236 2-4 faster than radix sort using any other power of two chosen as the base).
237
238 Time Complexity:  $O(n \cdot w)$  for  $n$  integers of  $w$  bits each.
239 Space Complexity:  $O(n + w)$  auxiliary.
240
241 */
242
243 template<class UnsignedIt>
244 void radix_sort(UnsignedIt lo, UnsignedIt hi) {
245     if (hi - lo < 2) {
246         return;
247     }
248     const int radix_bits = 8;
249     const int radix_base = 1 << radix_bits; // e.g.  $2^8 = 256$ 
250     const int radix_mask = radix_base - 1; // e.g.  $2^8 - 1 = 0xFF$ 
251     int num_bits = 8 * sizeof(*lo); // 8 bits per byte
252     typedef typename std::iterator_traits<UnsignedIt>::value_type T;
253     T *buf = new T[hi - lo];
254     for (int pos = 0; pos < num_bits; pos += radix_bits) {
255         int count[radix_base] = {0};
256         for (UnsignedIt it = lo; it != hi; ++it) {
257             count[(*it >> pos) & radix_mask]++;
258         }
259         T *bucket[radix_base], *curr = buf;
260         for (int i = 0; i < radix_base; curr += count[i++]) {
261             bucket[i] = curr;
262         }
263         for (UnsignedIt it = lo; it != hi; ++it) {
264             *bucket[(*it >> pos) & radix_mask]++ = *it;
265         }
266         std::copy(buf, buf + (hi - lo), lo);
267     }
268     delete[] buf;
269 }
270

```

```

271 /** Example Usage and Output:
272
273 mergesort() with default comparisons: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
274 mergesort() with 'compare_as_ints()': 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67
275 -----
276 Sorting five million integers...
277 std::sort(): 0.355s
278 quicksort(): 0.426s
279 mergesort(): 1.263s
280 heapsort(): 1.093s
281 combsort(): 0.827s
282 radix_sort(): 0.076s
283
284 ***/
285
286 #include <cassert>
287 #include <cstdlib>
288 #include <ctime>
289 #include <iomanip>
290 #include <iostream>
291 #include <vector>
292 using namespace std;
293
294 template<class It>
295 void print_range(It lo, It hi) {
296     while (lo != hi) {
297         cout << *lo++ << " ";
298     }
299     cout << endl;
300 }
301
302 template<class It>
303 bool sorted(It lo, It hi) {
304     while (++lo != hi) {
305         if (*lo < *(lo - 1)) {
306             return false;
307         }
308     }
309     return true;
310 }
311
312 bool compare_as_ints(double i, double j) {
313     return (int)i < (int)j;
314 }
315
316 int main () {
317     { // Can be used to sort arrays like std::sort().
318         int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
319         quicksort(a, a + 8);
320         assert(sorted(a, a + 8));
321     }
322     { // STL containers work too.
323         int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
324         vector<int> v(a, a + 8);
325         quicksort(v.begin(), v.end());
326         assert(sorted(v.begin(), v.end()));
327     }
328     { // Reverse iterators work as expected.
329         int a[] = {32, 71, 12, 45, 26, 80, 53, 33};

```

```

330     vector<int> v(a, a + 8);
331     heapsort(v.rbegin(), v.rend());
332     assert(sorted(v.rbegin(), v.rend()));
333 }
334 { // We can sort doubles just as well.
335     double a[] = {1.1, -5.0, 6.23, 4.123, 155.2};
336     vector<double> v(a, a + 5);
337     combsort(v.begin(), v.end());
338     assert(sorted(v.begin(), v.end()));
339 }
340 { // Must use radix_sort with unsigned values, but sorting in reverse works!
341     int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
342     vector<int> v(a, a + 8);
343     radix_sort(v.rbegin(), v.rend());
344     assert(sorted(v.rbegin(), v.rend()));
345 }
346
347 // Example from: http://www.cplusplus.com/reference/algorithm/stable_sort
348 double a[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
349 {
350     vector<double> v(a, a + 8);
351     cout << "mergesort() with default comparisons: ";
352     mergesort(v.begin(), v.end());
353     print_range(v.begin(), v.end());
354 }
355 {
356     vector<double> v(a, a + 8);
357     cout << "mergesort() with 'compare_as_ints()': ";
358     mergesort(v.begin(), v.end(), compare_as_ints);
359     print_range(v.begin(), v.end());
360 }
361 cout << "-----" << endl;
362
363 vector<int> v, v2;
364 for (int i = 0; i < 5000000; i++) {
365     v.push_back((rand() & 0x7fff) | ((rand() & 0x7fff) << 15));
366 }
367 v2 = v;
368 cout << "Sorting five million integers..." << endl;
369 cout.precision(3);
370
371 #define test(sort_function) { \
372     clock_t start = clock(); \
373     sort_function(v.begin(), v.end()); \
374     double t = (double)(clock() - start) / CLOCKS_PER_SEC; \
375     cout << setw(14) << left << #sort_function "() : "; \
376     cout << fixed << t << "s" << endl; \
377     assert(sorted(v.begin(), v.end())); \
378     v = v2; \
379 }
380 test(std::sort);
381 test(quicksort);
382 test(mergesort);
383 test(heapSort);
384 test(combsort);
385 test(radix_sort);
386 return 0;
387 }
```

1.1.2 Array Rotation

```

1  /*
2
3 These functions are equivalent to std::rotate(), taking three iterators lo, mid,
4 and hi ( $lo \leq mid \leq hi$ ) to perform a left rotation on the range  $[lo, hi]$ . After
5 the function call,  $[lo, hi]$  will comprise of the concatenation of the elements
6 originally in  $[mid, hi] + [lo, mid]$ . That is, the range  $[lo, hi]$  will be
7 rearranged in such a way that the element at mid becomes the first element of
8 the new range and the element at mid - 1 becomes the last element, all while
9 preserving the relative ordering of elements within the two rotated subarrays.
10
11 All three versions below achieve the same result using in-place algorithms.
12 Version 1 uses a straightforward swapping algorithm requiring ForwardIterators.
13 Version 2 requires BidirectionalIterators, employing a well-known trick with
14 three simple inversions. Version 3 requires random-access iterators, applying a
15 juggling algorithm which first divides the range into  $\text{gcd}(hi - lo, mid - lo)$ 
16 sets and then rotates the corresponding elements in each set.
17
18 Time Complexity:
19 -  $O(n)$  per call to both functions, where  $n$  is the distance between  $lo$  and  $hi$ .
20
21 Space Complexity:
22 -  $O(1)$  auxiliary for all versions
23
24 */
25
26 #include <algorithm>
27
28 template<class It>
29 void rotate1(It lo, It mid, It hi) {
30     It next = mid;
31     while (lo != next) {
32         std::iter_swap(lo++, next++);
33         if (next == hi) {
34             next = mid;
35         } else if (lo == mid) {
36             mid = next;
37         }
38     }
39 }
40
41 template<class It>
42 void rotate2(It lo, It mid, It hi) {
43     std::reverse(lo, mid);
44     std::reverse(mid, hi);
45     std::reverse(lo, hi);
46 }
47
48 int gcd(int a, int b) {
49     return (b == 0) ? a : gcd(b, a % b);
50 }
51
52 template<class It>
53 void rotate3(It lo, It mid, It hi) {
54     int n = hi - lo, jump = mid - lo;
55     int g = gcd(jump, n), cycle = n / g;

```

```

56     for (int i = 0; i < g; i++) {
57         int curr = i, next;
58         for (int j = 0; j < cycle - 1; j++) {
59             next = curr + jump;
60             if (next >= n) {
61                 next -= n;
62             }
63             std::iter_swap(lo + curr, lo + next);
64             curr = next;
65         }
66     }
67 }
68
69 /** Example Usage and Output:
70
71 before sort:  2 4 2 0 5 10 7 3 7 1
72 after sort:   0 1 2 2 3 4 5 7 7 10
73 rotate left: 1 2 2 3 4 5 7 7 10 0
74 rotate right: 0 1 2 2 3 4 5 7 7 10
75
76 */
77
78 #include <algorithm>
79 #include <cassert>
80 #include <iostream>
81 #include <vector>
82 using namespace std;
83
84 int main() {
85     vector<int> v0, v1, v2, v3;
86     for (int i = 0; i < 10000; i++) {
87         v0.push_back(i);
88     }
89     v1 = v2 = v3 = v0;
90     int mid = 5678;
91     std::rotate(v0.begin(), v0.begin() + mid, v0.end());
92     rotate1(v1.begin(), v1.begin() + mid, v1.end());
93     rotate2(v2.begin(), v2.begin() + mid, v2.end());
94     rotate3(v3.begin(), v3.begin() + mid, v3.end());
95     assert(v0 == v1 && v0 == v2 && v0 == v3);
96
97 // Example from: http://en.cppreference.com/w/cpp/algorithm/rotate
98 int a[] = {2, 4, 2, 0, 5, 10, 7, 3, 7, 1};
99 vector<int> v(a, a + 10);
100 cout << "before sort:  ";
101 for (int i = 0; i < (int)v.size(); i++) {
102     cout << v[i] << " ";
103 }
104 cout << endl;
105
106 // Insertion sort.
107 for (vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
108     rotate1(std::upper_bound(v.begin(), i, *i), i, i + 1);
109 }
110 cout << "after sort:  ";
111 for (int i = 0; i < (int)v.size(); i++) {
112     cout << v[i] << " ";
113 }
114 cout << endl;

```

```

115
116 // Simple rotation to the left.
117 rotate2(v.begin(), v.begin() + 1, v.end());
118 cout << "rotate left: ";
119 for (int i = 0; i < (int)v.size(); i++) {
120     cout << v[i] << " ";
121 }
122 cout << endl;
123
124 // Simple rotation to the right.
125 rotate3(v.rbegin(), v.rbegin() + 1, v.rend());
126 cout << "rotate right: ";
127 for (int i = 0; i < (int)v.size(); i++) {
128     cout << v[i] << " ";
129 }
130 cout << endl;
131
132 return 0;
133 }
```

1.1.3 Counting Inversions

```

1 /*
2
3 The number of inversions for an array a[] is defined as the number of ordered
4 pairs (i, j) such that i < j and a[i] > a[j]. This is roughly how "close" an
5 array is to being sorted, but is *not* the minimum number of swaps required to
6 sort the array. If the array is sorted, then the inversion count is 0. If the
7 array is sorted in decreasing order, then the inversion count is maximal. The
8 following two functions are each techniques to efficiently count inversions.
9
10 - inversions(lo, hi) uses merge sort to return the number of inversions given
11   two random-access iterators as a range [lo, hi). The input range will be
12   sorted after the function call. This requires operator < to be defined on the
13   iterators' value type.
14 - inversions(n, a[]) uses a power-of-two trick to return the number of
15   inversions for an array a[] of n nonnegative integers. After calling the
16   function, every value of a[] will be set to 0. The time and space complexity
17   of this operation are functions of the magnitude of the maximum value in a[].
18   To instead obtain a running time of  $O(n \log n)$  on the number of elements,
19   coordinate compression may be applied to a[] beforehand so that its maximum is
20   strictly less than the length n itself.
21
22 Time Complexity:
23 -  $O(n \log n)$  per call to inversions(lo, hi), where n is the distance between lo
24   and hi.
25 -  $O(n \log m)$  per call to inversions(n, a[]) where n is the distance between lo
26   and hi and m is maximum value in a[].
27
28 Space Complexity:
29 -  $O(n)$  auxiliary space and  $O(\log n)$  stack space for inversions(lo, hi).
30 -  $O(m)$  auxiliary heap space for inversions(n, a[]).
31
32 */
33
```

```

34 #include <algorithm>
35 #include <iterator>
36 #include <vector>
37
38 template<class It>
39 long long inversions(It lo, It hi) {
40     if (hi - lo < 2) {
41         return 0;
42     }
43     It mid = lo + (hi - lo - 1)/2, a = lo, c = mid + 1;
44     long long res = 0;
45     res += inversions(lo, mid + 1);
46     res += inversions(mid + 1, hi);
47     typedef typename std::iterator_traits<It>::value_type T;
48     std::vector<T> merged;
49     while (a <= mid && c < hi) {
50         if (*c < *a) {
51             merged.push_back(*(c++));
52             res += (mid - a) + 1;
53         } else {
54             merged.push_back(*(a++));
55         }
56     }
57     if (a > mid) {
58         for (It it = c; it != hi; ++it) {
59             merged.push_back(*it);
60         }
61     } else {
62         for (It it = a; it <= mid; ++it) {
63             merged.push_back(*it);
64         }
65     }
66     for (It it = lo; it != hi; ++it) {
67         *it = merged[it - lo];
68     }
69     return res;
70 }
71
72 long long inversions(int n, int a[]) {
73     int mx = 0;
74     for (int i = 0; i < n; i++) {
75         mx = std::max(mx, a[i]);
76     }
77     long long res = 0;
78     std::vector<int> count(mx);
79     while (mx > 0) {
80         std::fill(count.begin(), count.end(), 0);
81         for (int i = 0; i < n; i++) {
82             if (a[i] % 2 == 0) {
83                 res += count[a[i] / 2];
84             } else {
85                 count[a[i] / 2]++;
86             }
87         }
88         mx = 0;
89         for (int i = 0; i < n; i++) {
90             mx = std::max(mx, a[i] /= 2);
91         }
92     }

```

```

93     return res;
94 }
95
96 /** Example Usage ***/
97
98 #include <cassert>
99
100 int main() {
101 {
102     int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
103     assert(inversions(a, a + 8) == 16);
104 }
105 {
106     int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
107     assert(inversions(8, a) == 16);
108 }
109     return 0;
110 }
```

1.1.4 Coordinate Compression

```

1 /*
2
3 Given two ForwardIterators as a range [lo, hi) of n numerical elements, reassign
4 each element in the range to an integer in [0, k), where k is the number of
5 distinct elements in the original range, while preserving the initial relative
6 ordering of elements. That is, if a[] is an array of the original values and b[]
7 is the compressed values, then every pair of indices i, j (0 <= i, j < n) shall
8 satisfy a[i] < a[j] if and only if b[i] < b[j].
9
10 Both implementations below require operator < to be defined on the iterator's
11 value type. Version 1 performs the compression by sorting the array, removing
12 duplicates, and binary searching for the position of each original value.
13 Version 2 achieves the same result by inserting all values in a balanced binary
14 search tree (std::map) which automatically removes duplicate values and supports
15 efficient lookups of the compressed values.
16
17 Time Complexity:
18 - O(n log n) per call to either function, where n is the distance between lo and
19   hi.
20
21 Space Complexity:
22 - O(n) auxiliary heap space.
23
24 */
25
26 #include <algorithm>
27 #include <iterator>
28 #include <map>
29 #include <vector>
30
31 template<class It> void compress1(It lo, It hi) {
32     typedef typename std::iterator_traits<It>::value_type T;
33     std::vector<T> v(lo, hi);
34     std::sort(v.begin(), v.end());
```

```

35     v.resize(std::unique(v.begin(), v.end()) - v.begin());
36     for (It it = lo; it != hi; ++it) {
37         *it = (int)(std::lower_bound(v.begin(), v.end(), *it) - v.begin());
38     }
39 }
40
41 template<class It> void compress2(It lo, It hi) {
42     typedef typename std::iterator_traits<It>::value_type T;
43     std::map<T, int> m;
44     for (It it = lo; it != hi; ++it) {
45         m[*it] = 0;
46     }
47     typename std::map<T, int>::iterator it = m.begin();
48     for (int id = 0; it != m.end(); it++) {
49         it->second = id++;
50     }
51     for (It it = lo; it != hi; ++it) {
52         *it = m[*it];
53     }
54 }
55
56 /** Example Usage and Output:
57
58 0 4 4 1 3 2 5 5
59 0 4 4 1 3 2 5 5
60 1 0 2 0 3 1
61
62 */
63
64 #include <iostream>
65 using namespace std;
66
67 template<class It> void print_range(It lo, It hi) {
68     while (lo != hi) {
69         cout << *lo++ << " ";
70     }
71     cout << endl;
72 }
73
74 int main() {
75 {
76     int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
77     compress1(a, a + 8);
78     print_range(a, a + 8);
79 }
80 {
81     int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
82     compress2(a, a + 8);
83     print_range(a, a + 8);
84 }
85 // Non-integral types work too, as long as ints can be assigned to them.
86 double a[] = {0.5, -1.0, 3, -1.0, 20, 0.5};
87 compress1(a, a + 6);
88 print_range(a, a + 6);
89 }
90 return 0;
91 }
```

1.1.5 Selection (Quickselect)

```

1  /*
2
3 nth_element2() is equivalent to std::nth_element(), taking random-access
4 iterators lo, nth, and hi as the range [lo, hi) to be partially sorted. The
5 values in [lo, hi) are rearranged such that the value pointed to by nth is the
6 element that would be there if the range were sorted. Furthermore, the range is
7 partitioned such that no value in [lo, nth) compares greater than the value
8 pointed to by nth and no value in (nth, hi) compares less. This implementation
9 requires operator < to be defined on the iterator's value type.
10
11 Time Complexity:
12 - O(n) on average per call to nth_element2(), where n is the distance between lo
13 and hi.
14
15 Space Complexity:
16 - O(1) auxiliary.
17
18 */
19
20 #include <algorithm>
21 #include <cstdlib>
22 #include <iterator>
23
24 int rand32() {
25     return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
26 }
27
28 template<class It>
29 void nth_element2(It lo, It nth, It hi) {
30     for (;;) {
31         std::iter_swap(lo + rand32() % (hi - lo), hi - 1);
32         typename std::iterator_traits<It>::value_type mid = *(hi - 1);
33         It k = lo - 1;
34         for (It it = lo; it != hi; ++it) {
35             if (!(mid < *it)) {
36                 std::iter_swap(++k, it);
37             }
38         }
39         if (nth < k) {
40             hi = k;
41         } else if (k < nth) {
42             lo = k + 1;
43         } else {
44             return;
45         }
46     }
47 }
48
49 /** Example Usage and Output:
50
51 2 3 3 4 5 6 6 7 9
52
53 ***/
54 #include <cassert>
```

```

56 #include <iostream>
57 using namespace std;
58
59 template<class It>
60 void print_range(It lo, It hi) {
61     while (lo != hi) {
62         cout << *lo++ << " ";
63     }
64     cout << endl;
65 }
66
67 int main () {
68     int n = 9;
69     int a[] = {5, 6, 4, 3, 2, 6, 7, 9, 3};
70     nth_element2(a, a + n/2, a + n);
71     assert(a[n/2] == 5);
72     print_range(a, a + n);
73     return 0;
74 }
```

1.2 Array Queries

1.2.1 Longest Increasing Subsequence

```

1 /*
2
3 Given two random-access iterators lo and hi specifying a range [lo, hi),
4 determine a longest subsequence of the range such that all of its elements are
5 in strictly ascending order. This implementation requires operator < to be
6 defined on the iterator's value type. The subsequence is not necessarily
7 contiguous or unique, so only one such answer will be found. The answer is
8 computed using binary search and dynamic programming.
9
10 Time Complexity:
11 - O(n log n) per call to longest_increasing_subsequence(), where n is the
12   distance between lo and hi.
13
14 Space Complexity:
15 - O(n) auxiliary heap space for longest_increasing_subsequence().
16
17 */
18
19 #include <iterator>
20 #include <vector>
21
22 template<class It>
23 std::vector<typename std::iterator_traits<It>::value_type>
24 longest_increasing_subsequence(It lo, It hi) {
25     int len = 0, n = hi - lo;
26     std::vector<int> prev(n), tail(n);
27     for (int i = 0; i < n; i++) {
28         int l = -1, h = len;
29         while (h - l > 1) {
```

```

30     int mid = (l + h)/2;
31     if (*(lo + tail[mid]) < *(lo + i)) {
32         l = mid;
33     } else {
34         h = mid;
35     }
36 }
37 if (len < h + 1) {
38     len = h + 1;
39 }
40 prev[i] = h > 0 ? tail[h - 1] : -1;
41 tail[h] = i;
42 }
43 std::vector<typename std::iterator_traits<It>::value_type> res(len);
44 for (int i = tail[len - 1]; i != -1; i = prev[i]) {
45     res[--len] = *(lo + i);
46 }
47 return res;
48 }
49
50 /** Example Usage and Output:
51 -5 1 9 10 11 13
52 */
53
54 /**
55
56 #include <iostream>
57 using namespace std;
58
59 template<class It> void print_range(It lo, It hi) {
60     while (lo != hi) {
61         cout << *lo++ << " ";
62     }
63     cout << endl;
64 }
65
66 int main () {
67     int a[] = {-2, -5, 1, 9, 10, 8, 11, 10, 13, 11};
68     vector<int> res = longest_increasing_subsequence(a, a + 10);
69     print_range(res.begin(), res.end());
70     return 0;
71 }
```

1.2.2 Maximal Subarray Sum (Kadane)

```

1 /*
2
3 Given an array of numbers (at least one of which must be positive), determine
4 the maximum possible sum of any contiguous subarray. Kadane's algorithm scans
5 through the array, at each index computing the maximum positive sum subarray
6 ending there. Either this subarray is empty (in which case its sum is zero) or
7 it consists of one more element than the maximum sequence ending at the previous
8 position. This can be adapted to compute the maximal submatrix sum as well.
9
10 - max_subarray_sum(lo, hi, &res_lo, &res_hi) returns the maximal subarray sum
```

```

11  for the range [lo, hi), where lo and hi are random-access iterators to
12  numeric types. This implementation requires operators + and < to be defined on
13  the iterators' value type. Optionally, two int pointers may be passed to store
14  the inclusive boundary indices [res_lo, res_hi] of the resulting subarray. By
15  convention, an input range consisting of only negative values will yield a
16  size 1 subarray consisting of the maximum value.
17 - max_submatrix_sum(matrix, &r1, &c1, &r2, &c2) returns the largest sum of any
18  rectangular submatrix for a matrix of n rows by m columns. The matrix should
19  be given as a 2-dimensional vector, where the outer vector must contain n
20  vectors each of size m. This implementation requires operators + and < to be
21  defined on the iterators' value type. Optionally, four int pointers may be
22  passed to store the boundary indices of the resulting subarray, with (r1, c1)
23  specifying the top-left index and (r2, c2) specifying the bottom-right index.
24  By convention, an input matrix consisting of only negative values will yield a
25  size 1 submatrix consisting of the maximum value.
26
27 Time Complexity:
28 - O(n) per call to max_subarray_sum(), where n is the distance between lo and
29  hi.
30 - O(n*m^2) per call to max_submatrix_sum(), where n is the number of rows and m
31  is the number of columns in the matrix.
32
33 Space Complexity:
34 - O(1) auxiliary for max_subarray_sum().
35 - O(n) auxiliary heap space for max_submatrix_sum(), where n is the number of
36  rows in the matrix.
37
38 */
39
40 #include <algorithm>
41 #include <cstddef>
42 #include <iterator>
43 #include <limits>
44 #include <vector>
45
46 template<class It>
47 typename std::iterator_traits<It>::value_type
48 max_subarray_sum(It lo, It hi, int *res_lo = NULL, int *res_hi = NULL) {
49     typedef typename std::iterator_traits<It>::value_type T;
50     int curr_begin = 0, begin = 0, end = -1;
51     T sum = 0, max_sum = std::numeric_limits<T>::min();
52     for (It it = lo; it != hi; ++it) {
53         sum += *it;
54         if (sum < 0) {
55             sum = 0;
56             curr_begin = (it - lo) + 1;
57         } else if (max_sum < sum) {
58             max_sum = sum;
59             begin = curr_begin;
60             end = it - lo;
61         }
62     }
63     if (end == -1) { // All negative. By convention, return the maximum value.
64         for (It it = lo; it != hi; ++it) {
65             if (max_sum < *it) {
66                 max_sum = *it;
67                 begin = it - lo;
68                 end = begin;
69             }

```

```

70     }
71 }
72 if (res_lo != NULL && res_hi != NULL) {
73     *res_lo = begin;
74     *res_hi = end;
75 }
76 return max_sum;
77 }

78
79 template<class T>
80 T max_submatrix_sum(const std::vector<std::vector<T> > &matrix,
81     int *r1 = NULL, int *c1 = NULL, int *r2 = NULL, int *c2 = NULL) {
82     int n = matrix.size(), m = matrix[0].size();
83     std::vector<T> sums(n);
84     T sum, max_sum = std::numeric_limits<T>::min();
85     for (int clo = 0; clo < m; clo++) {
86         std::fill(sums.begin(), sums.end(), 0);
87         for (int chi = clo; chi < m; chi++) {
88             for (int i = 0; i < n; i++) {
89                 sums[i] += matrix[i][chi];
90             }
91             int rlo, rhi;
92             sum = max_subarray_sum(sums.begin(), sums.end(), &rlo, &rhi);
93             if (max_sum < sum) {
94                 max_sum = sum;
95                 if (r1 != NULL && c1 != NULL && r2 != NULL && c2 != NULL) {
96                     *r1 = rlo;
97                     *c1 = clo;
98                     *r2 = rhi;
99                     *c2 = chi;
100                }
101            }
102        }
103    }
104    return max_sum;
105 }

106
107 /** Example Usage and Output:
108
109 Maximal sum subarray:
110 4 -1 2 1
111
112 Maximal sum submatrix:
113 9 2
114 -4 1
115 -1 8
116
117 */
118
119 #include <cassert>
120 #include <iostream>
121 using namespace std;
122
123 int main() {
124 {
125     int a[] = {-2, -1, -3, 4, -1, 2, 1, -5, 4};
126     int lo = 0, hi = 0;
127     assert(max_subarray_sum(a, a + 3) == -1);
128     assert(max_subarray_sum(a, a + 9, &lo, &hi) == 6);

```

```

129     cout << "Maximal sum subarray:" << endl;
130     for (int i = lo; i <= hi; i++) {
131         cout << a[i] << " ";
132     }
133     cout << endl;
134 }
135 {
136     const int n = 4, m = 5;
137     int a[n][m] = {{0, -2, -7, 0, 5},
138                     {9, 2, -6, 2, -4},
139                     {-4, 1, -4, 1, 0},
140                     {-1, 8, 0, -2, 3}};
141     vector<vector<int>> matrix(n);
142     for (int i = 0; i < n; i++) {
143         matrix[i] = vector<int>(a[i], a[i] + m);
144     }
145     int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
146     assert(max_submatrix_sum(matrix, &r1, &c1, &r2, &c2) == 15);
147     cout << "\nMaximal sum submatrix:" << endl;
148     for (int i = r1; i <= r2; i++) {
149         for (int j = c1; j <= c2; j++) {
150             cout << matrix[i][j] << " ";
151         }
152         cout << endl;
153     }
154 }
155 return 0;
156 }
```

1.2.3 Majority Element (Boyer-Moore)

```

1 /*
2
3 Given two ForwardIterators lo and hi specifying a range [lo, hi) of n elements,
4 return an iterator to the first occurrence of the majority element, or the
5 iterator hi if there is no majority element. The majority element is defined as
6 an element which occurs strictly more than floor(n/2) times in the range. This
7 implementation requires operator == to be defined on the iterator's value type.
8
9 Time Complexity:
10 - O(n) per call to majority(), where n is the size of the array.
11
12 Space Complexity:
13 - O(1) auxiliary.
14
15 */
16
17 template<class It>
18 It majority(It lo, It hi) {
19     int count = 0;
20     It candidate = lo;
21     for (It it = lo; it != hi; ++it) {
22         if (count == 0) {
23             candidate = it;
24             count = 1;
```

```

25     } else if (*it == *candidate) {
26         count++;
27     } else {
28         count--;
29     }
30 }
31 count = 0;
32 for (It it = lo; it != hi; ++it) {
33     if (*it == *candidate) {
34         count++;
35     }
36 }
37 if (count <= (hi - lo)/2) {
38     return hi;
39 }
40 return candidate;
41 }
42
43 /** Example Usage ***/
44
45 #include <cassert>
46
47 int main() {
48     int a[] = {3, 2, 3, 1, 3};
49     assert(*majority(a, a + 5) == 3);
50     int b[] = {2, 3, 3, 3, 2, 1};
51     assert(majority(b, b + 6) == b + 6);
52     return 0;
53 }
```

1.2.4 Subset Sum (Meet-in-the-Middle)

```

1 /*
2
3 Given random-access iterators lo and hi specifying a range [lo, hi) of integers,
4 return the minimum sum of any subset of the range that is greater than or equal
5 to a given integer v. This is a generalization of the NP-complete subset sum
6 problem, which asks whether a subset summing to 0 exists (equivalent in this
7 case to checking if v = 0 yields an answer of 0). This implementation uses a
8 meet-in-the-middle algorithm to precompute and search for a lower bound. Note
9 that 64-bit integers are used in intermediate calculations to avoid overflow.
10
11 Time Complexity:
12 - O(n*2^(n/2)) per call to sum_lower_bound(), where n is the distance between lo
13 and hi.
14
15 Space Complexity:
16 - O(n) auxiliary heap space, where n is the number of array elements.
17
18 */
19
20 #include <algorithm>
21 #include <limits>
22 #include <vector>
23
```

```

24 template<class It>
25 long long sum_lower_bound(It lo, It hi, long long v) {
26     int n = hi - lo, llen = 1 << (n/2), hlen = 1 << (n - n/2);
27     std::vector<long long> lsum(llen), hsum(hlen);
28     for (int mask = 0; mask < llen; mask++) {
29         for (int i = 0; i < n/2; i++) {
30             if ((mask >> i) & 1) {
31                 lsum[mask] += *(lo + i);
32             }
33         }
34     }
35     for (int mask = 0; mask < hlen; mask++) {
36         for (int i = 0; i < (n - n/2); i++) {
37             if ((mask >> i) & 1) {
38                 hsum[mask] += *(lo + i + n/2);
39             }
40         }
41     }
42     std::sort(lsum.begin(), lsum.end());
43     std::sort(hsum.begin(), hsum.end());
44     int l = 0, h = hlen - 1;
45     long long curr = std::numeric_limits<long long>::min();
46     while (l < llen && h >= 0) {
47         if (lsum[l] + hsum[h] <= v) {
48             curr = std::max(curr, lsum[l] + hsum[h]);
49             l++;
50         } else {
51             h--;
52         }
53     }
54     return curr;
55 }
56
57 /** Example Usage ***/
58
59 #include <cassert>
60
61 int main() {
62     int a[] = {9, 1, 5, 0, 1, 11, 5};
63     assert(sum_lower_bound(a, a + 7, 8) == 7);
64     int b[] = {-7, -3, -2, 5, 8};
65     assert(sum_lower_bound(b, b + 5, 0) == 0);
66     return 0;
67 }
```

1.2.5 Maximal Zero Submatrix

```

1 /*
2
3 Given a rectangular matrix with n rows and m columns consisting of only 0's and
4 1's as a two-dimensional vector of bool, return the area of the largest
5 rectangular submatrix consisting of only 0's. This solution uses a reduction to
6 the problem of finding the maximum rectangular area under a histogram, which is
7 efficiently solved using a stack algorithm.
8
```

```

9 Time Complexity:
10 - O(n*m) per call to max_zero_submatrix(), where n is the number of rows and m
11 is the number of columns in the matrix.
12
13 Space Complexity:
14 - O(m) auxiliary heap space, where m is the number of columns in the matrix.
15
16 */
17
18 #include <algorithm>
19 #include <stack>
20 #include <vector>
21
22 int max_zero_submatrix(const std::vector<std::vector<bool> > &matrix) {
23     int n = matrix.size(), m = matrix[0].size(), res = 0;
24     std::vector<int> d(m, -1), d1(m), d2(m);
25     for (int r = 0; r < n; r++) {
26         for (int c = 0; c < m; c++) {
27             if (matrix[r][c]) {
28                 d[c] = r;
29             }
30         }
31         std::stack<int> s;
32         for (int c = 0; c < m; c++) {
33             while (!s.empty() && d[s.top()] <= d[c]) {
34                 s.pop();
35             }
36             d1[c] = s.empty() ? -1 : s.top();
37             s.push(c);
38         }
39         while (!s.empty()) {
40             s.pop();
41         }
42         for (int c = m - 1; c >= 0; c--) {
43             while (!s.empty() && d[s.top()] <= d[c]) {
44                 s.pop();
45             }
46             d2[c] = s.empty() ? m : s.top();
47             s.push(c);
48         }
49         for (int j = 0; j < m; j++) {
50             res = std::max(res, (r - d[j]) * (d2[j] - d1[j] - 1));
51         }
52     }
53     return res;
54 }
55
56 /** Example Usage ***/
57
58 #include <cassert>
59 using namespace std;
60
61 int main() {
62     const int n = 5, m = 6;
63     bool a[n][m] = {{1, 0, 1, 1, 0, 0},
64                     {1, 0, 0, 1, 0, 0},
65                     {0, 0, 0, 0, 0, 1},
66                     {1, 0, 0, 1, 0, 0},
67                     {1, 0, 1, 0, 0, 1}};

```

```

68     vector<vector<bool> > matrix(n);
69     for (int i = 0; i < n; i++) {
70         matrix[i] = vector<bool>(a[i], a[i] + m);
71     }
72     assert(max_zero_submatrix(matrix) == 6);
73     return 0;
74 }
```

1.3 Searching

1.3.1 Binary Search

```

1  /*
2
3 Binary search can be generally used to find the input value corresponding to any
4 output value of a monotonic (strictly non-increasing or strictly non-decreasing)
5 function in  $O(\log n)$  time with respect to the domain size. This is a special
6 case of finding the exact point at which any given monotonic Boolean function
7 changes from true to false or vice versa. Unlike searching through an array,
8 discrete binary search is not restricted by available memory, making it useful
9 for handling infinitely large search spaces such as real number intervals.
10
11 - binary_search_first_true() takes two integers lo and hi as boundaries for the
12   search space [lo, hi) (i.e. including lo, but excluding hi) and returns the
13   smallest integer k in [lo, hi) for which the predicate pred(k) tests true. If
14   pred(k) tests false for every k in [lo, hi), then hi is returned. This
15   function must be used on a range in which there exists a constant k such that
16   pred(x) tests false for every x in [lo, k) and true for every x in [k, hi).
17 - binary_search_last_true() takes two integers lo and hi as boundaries for the
18   search space [lo, hi) (i.e. including lo, but excluding hi) and returns the
19   largest integer k in [lo, hi) for which the predicate pred(k) tests true. If
20   pred(k) tests false for every k in [lo, hi), then hi is returned. This
21   function must be used on a range in which there exists a constant k such that
22   pred(x) tests true for every x in [lo, k] and false for every x in (k, hi).
23 - fbinary_search() is the equivalent of binary_search_first_true() on floating
24   point predicates. Since any interval of real numbers is dense, the exact
25   target cannot be found due to floating point error. Instead, a value that is
26   very close to the border between false and true is returned. The precision of
27   the answer depends on the number of repetitions the function performs. Since
28   each repetition bisects the search space, the absolute error of the answer is
29    $1/(2^n)$  times the distance between lo and hi after n repetitions. Although it
30   is possible to control the error by looping while hi - lo is greater than an
31   arbitrary epsilon, it is simpler to let the loop run for a desired number of
32   iterations until floating point arithmetic break down. 100 iterations is
33   usually sufficient, since the search space will be reduced to  $2^{-100}$  (roughly
34    $10^{-30}$ ) times its original size. This implementation can be modified to find
35   the "last true" point in the range by simply interchanging the assignments of
36   lo and hi in the if-else statements.
37
38 Time Complexity:
39 -  $O(\log n)$  calls will be made to pred() in binary_search_first_true() and
40   binary_search_last_true(), where n is the distance between lo and hi.
41 -  $O(n)$  calls will be made to pred() in fbinary_search(), where n is the number
```

```

42     of iterations performed.
43
44 Space Complexity:
45 - O(1) auxiliary for all operations.
46
47 */
48
49 template<class Int, class IntPredicate>
50 Int binary_search_first_true(Int lo, Int hi, IntPredicate pred) { // 000[1]11
51     Int mid, _hi = hi;
52     while (lo < hi) {
53         mid = lo + (hi - lo)/2;
54         if (pred(mid)) {
55             hi = mid;
56         } else {
57             lo = mid + 1;
58         }
59     }
60     if (!pred(lo)) {
61         return _hi; // All false.
62     }
63     return lo;
64 }
65
66 template<class Int, class IntPredicate>
67 Int binary_search_last_true(Int lo, Int hi, IntPredicate pred) { // 11[1]000
68     Int mid, _hi = hi--;
69     while (lo < hi) {
70         mid = lo + (hi - lo + 1)/2;
71         if (pred(mid)) {
72             lo = mid;
73         } else {
74             hi = mid - 1;
75         }
76     }
77     if (!pred(lo)) {
78         return _hi; // All false.
79     }
80     return lo;
81 }
82
83 template<class DoublePredicate>
84 double fbinary_search(double lo, double hi, DoublePredicate pred) { // 000[1]11
85     double mid;
86     for (int i = 0; i < 100; i++) {
87         mid = (lo + hi)/2.0;
88         if (pred(mid)) {
89             hi = mid;
90         } else {
91             lo = mid;
92         }
93     }
94     return lo;
95 }
96
97 /** Example Usage ***/
98
99 #include <cassert>
100 #include <cmath>

```

```

101 // Simple predicate examples.
102 bool pred1(int x) { return x >= 3; }
103 bool pred2(int x) { return false; }
104 bool pred3(int x) { return x <= 5; }
105 bool pred4(int x) { return true; }
106 bool pred5(double x) { return x >= 1.2345; }
107
108 int main() {
109     assert(binary_search_first_true(0, 7, pred1) == 3);
110     assert(binary_search_first_true(0, 7, pred2) == 7);
111     assert(binary_search_last_true(0, 7, pred3) == 5);
112     assert(binary_search_last_true(0, 7, pred4) == 6);
113     assert(fabs(fbinary_search(-10.0, 10.0, pred5) - 1.2345) < 1e-15);
114
115     return 0;
116 }
```

1.3.2 Ternary Search

```

1 /*
2
3 Given a unimodal function f(x) taking a single double argument, find its global
4 maximum or minimum point to a specified absolute error.
5
6 ternary_search_min() takes the domain [lo, hi] of a continuous function f(x) and
7 returns a number x such that f is strictly decreasing on the interval [lo, x]
8 and strictly increasing on the interval [x, hi]. For the function to be correct
9 and deterministic, such an x must exist and be unique.
10
11 ternary_search_max() takes the domain [lo, hi] of a continuous function f(x) and
12 returns a number x such that f is strictly increasing on the interval [lo, x]
13 and strictly decreasing on the interval [x, hi]. For the function to be correct
14 and deterministic, such an x must exist and be unique.
15
16 Time Complexity:
17 - O(log n) calls will be made to f(), where n is the distance between lo and hi
18 divided by the specified absolute error (epsilon).
19
20 Space Complexity:
21 - O(1) auxiliary for both operations.
22
23 */
24
25 template<class UnimodalFunction>
26 double ternary_search_min(double lo, double hi, UnimodalFunction f,
27                           const double EPS = 1e-12) {
28     double lthird, hthird;
29     while (hi - lo > EPS) {
30         lthird = lo + (hi - lo)/3;
31         hthird = hi - (hi - lo)/3;
32         if (f(lthird) < f(hthird)) {
33             hi = hthird;
34         } else {
35             lo = lthird;
36         }
```

```

37     }
38     return lo;
39 }
40
41 template<class UnimodalFunction>
42 double ternary_search_max(double lo, double hi, UnimodalFunction f,
43                         const double EPS = 1e-12) {
44     double lthird, hthird;
45     while (hi - lo > EPS) {
46         lthird = lo + (hi - lo)/3;
47         hthird = hi - (hi - lo)/3;
48         if (f(lthird) < f(hthird)) {
49             lo = lthird;
50         } else {
51             hi = hthird;
52         }
53     }
54     return hi;
55 }
56
57 /** Example Usage **/
58
59 #include <cassert>
60 #include <cmath>
61
62 bool equal(double a, double b) {
63     return fabs(a - b) < 1e-7;
64 }
65
66 // Parabola opening up with vertex at (-2, -24).
67 double f1(double x) {
68     return 3*x*x + 12*x - 12;
69 }
70
71 // Parabola opening down with vertex at (2/19, 8366/95) ~ (0.105, 88.063).
72 double f2(double x) {
73     return -5.7*x*x + 1.2*x + 88;
74 }
75
76 // Absolute value function shifted to the right by 30 units.
77 double f3(double x) {
78     return fabs(x - 30);
79 }
80
81 int main() {
82     assert(equal(ternary_search_min(-1000, 1000, f1), -2));
83     assert(equal(ternary_search_max(-1000, 1000, f2), 2.0/19));
84     assert(equal(ternary_search_min(-1000, 1000, f3), 30));
85     return 0;
86 }
```

1.3.3 Hill Climbing

```

1 /*
2
```

```

3 Given a continuous function f(x, y) to double and a (possibly arbitrary) initial
4 guess (x0, y0), return a potential global minimum found through hill-climbing.
5 Optionally, two double pointers critical_x and critical_y may be passed to store
6 the input points to f at which the returned minimum value is attained.
7
8 Hill-climbing is a heuristic which starts at the guess, then considers taking
9 a single step in each of a fixed number of directions. The direction with the
10 best (in this case, minimum) value is chosen, and further steps are taken in it
11 until the answer no longer improves. When this happens, the step size is reduced
12 and the same process repeats until a desired absolute error is reached. The
13 technique's success heavily depends on the behavior of f and the initial guess.
14 Therefore, the result is not guaranteed to be the global minimum.
15
16 Time Complexity:
17 - O(d log n) call will be made to f, where d is the number of directions
18   considered at each position and n is the search space that is approximately
19   proportional to the maximum possible step size divided by the minimum possible
20   step size.
21
22 Space Complexity:
23 - O(1) auxiliary.
24 */
25
26
27 #include <cstddef>
28 #include <cmath>
29
30 template<class ContinuousFunction>
31 double find_min(ContinuousFunction f, double x0, double y0,
32                  double *critical_x = NULL, double *critical_y = NULL,
33                  const double STEP_MIN = 1e-9, const double STEP_MAX = 1e6,
34                  const int NUM_DIRECTIONS = 6) {
35     static const double PI = acos(-1.0);
36     double x = x0, y = y0, res = f(x0, y0);
37     for (double step = STEP_MAX; step > STEP_MIN; ) {
38         double best = res, best_x = x, best_y = y;
39         bool found = false;
40         for (int i = 0; i < NUM_DIRECTIONS; i++) {
41             double a = 2.0*PI*i / NUM_DIRECTIONS;
42             double x2 = x + step*cos(a), y2 = y + step*sin(a);
43             double value = f(x2, y2);
44             if (best > value) {
45                 best_x = x2;
46                 best_y = y2;
47                 best = value;
48                 found = true;
49             }
50         }
51         if (!found) {
52             step /= 2.0;
53         } else {
54             x = best_x;
55             y = best_y;
56             res = best;
57         }
58     }
59     if (critical_x != NULL && critical_y != NULL) {
60         *critical_x = x;
61         *critical_y = y;

```

```

62     }
63     return res;
64 }
65
66 /** Example Usage **/
67
68 #include <cassert>
69 #include <cmath>
70
71 bool eq(double a, double b) {
72     return fabs(a - b) < 1e-8;
73 }
74
75 // Paraboloid with global minimum at f(2, 3) = 0.
76 double f(double x, double y) {
77     return (x - 2)*(x - 2) + (y - 3)*(y - 3);
78 }
79
80 int main() {
81     double x, y;
82     assert(eq(find_min(f, 0, 0, &x, &y), 0));
83     assert(eq(x, 2) && eq(y, 3));
84     return 0;
85 }
```

1.3.4 Convex Hull Trick (Semi-Dynamic)

```

1 /*
2
3 Given a set of pairs (m, b) specifying lines of the form y = mx + b, process a
4 set of x-coordinate queries each asking to find the minimum y-value when any of
5 the given lines are evaluated at the specified x. For each add_line(m, b) call,
6 m must be the minimum m of all lines added so far. For each query(x) call, x
7 must be the maximum x of all queries made so far.
8
9 The following implementation is a concise, semi-dynamic version of the convex
10 hull optimization technique. It supports an interlaced sequence of add_line()
11 and query() calls, as long as the preconditions of descending m and ascending x
12 are satisfied. As a result, it may be necessary to sort the lines and queries
13 before calling the functions. In that case, the overall time complexity will be
14 dominated by the sorting step.
15
16 Time Complexity:
17 - O(n) for any interlaced sequence of add_line() and query() calls, where n is
18 the number of lines added. This is because the overall number of steps taken
19 by add_line() and query() are respectively bounded by the number of lines.
20 Thus a single call to either add_line() or query() will have an amortized O(1)
21 running time.
22
23 Space Complexity:
24 - O(n) for storage of the lines.
25 - O(1) auxiliary for add_line() and query().
26
27 */
28
```

```

29 #include <vector>
30
31 std::vector<long long> M, B;
32 int ptr = 0;
33
34 void add_line(long long m, long long b) {
35     int len = M.size();
36     while (len > 1 && (B[len - 2] - B[len - 1])*(m - M[len - 1]) >=
37             (B[len - 1] - b)*(M[len - 1] - M[len - 2])) {
38         len--;
39     }
40     M.resize(len);
41     B.resize(len);
42     M.push_back(m);
43     B.push_back(b);
44 }
45
46 long long query(long long x) {
47     if (ptr >= (int)M.size()) {
48         ptr = (int)M.size() - 1;
49     }
50     while (ptr + 1 < (int)M.size() &&
51             M[ptr + 1]*x + B[ptr + 1] <= M[ptr]*x + B[ptr]) {
52         ptr++;
53     }
54     return M[ptr]*x + B[ptr];
55 }
56
57 /** Example Usage ***/
58
59 #include <cassert>
60
61 int main() {
62     add_line(3, 0);
63     add_line(2, 1);
64     add_line(1, 2);
65     add_line(0, 6);
66     assert(query(0) == 0);
67     assert(query(1) == 3);
68     assert(query(2) == 4);
69     assert(query(3) == 5);
70     return 0;
71 }
```

1.3.5 Convex Hull Trick (Fully-Dynamic)

```

1 /*
2
3 Given a set of pairs (m, b) specifying lines of the form  $y = mx + b$ , process a
4 set of x-coordinate queries each asking to find the minimum y-value when any of
5 the given lines are evaluated at the specified x. To instead have the queries
6 optimize for maximum y-value, call the constructor with query_max=true.
7
8 The following implementation is a fully dynamic variant of the convex hull
9 optimization technique, using a self-balancing binary search tree (std::set) to
```

```

10 support the ability to call add_line() and query() in any desired order.
11
12 Time Complexity:
13 - O(n) for any interlaced sequence of add_line() and query() calls, where n
14   is the number of lines added. This is because the overall number of steps
15   taken by add_line() and query() are respectively bounded by the number of
16   lines. Thus a single call to either add_line() or query() will have an O(1)
17   amortized running time.
18
19 Space Complexity:
20 - O(n) for storage of the lines.
21 - O(1) auxiliary for add_line() and query().
22
23 */
24
25 #include <limits>
26 #include <set>
27
28 class hull_optimizer {
29     struct line {
30         long long m, b, value;
31         double xlo;
32         bool is_query, query_max;
33
34         line(long long m, long long b, long long v, bool is_query, bool query_max)
35             : m(m), b(b), value(v), xlo(-std::numeric_limits<double>::max()),
36               is_query(is_query), query_max(query_max) {}
37
38         double intersect(const line &l) const {
39             if (m == l.m) {
40                 return std::numeric_limits<double>::max();
41             }
42             return (double)(l.b - b)/(m - l.m);
43         }
44
45         bool operator<(const line &l) const {
46             if (l.is_query) {
47                 return query_max ? (xlo < l.value) : (l.value < xlo);
48             }
49             return m < l.m;
50         }
51     };
52
53     std::set<line> hull;
54     bool query_max;
55
56     typedef std::set<line>::iterator hulliter;
57
58     bool has_prev(hulliter it) const {
59         return it != hull.begin();
60     }
61
62     bool has_next(hulliter it) const {
63         return (it != hull.end()) && (++it != hull.end());
64     }
65
66     bool irrelevant(hulliter it) const {
67         if (!has_prev(it) || !has_next(it)) {
68             return false;

```

```

69      }
70      hulliter prev = it, next = it;
71      --prev;
72      ++next;
73      return query_max ? (prev->intersect(*next) <= prev->intersect(*it))
74                      : (next->intersect(*prev) <= next->intersect(*it));
75  }
76
77  hulliter update_left_border(hulliter it) {
78  if ((query_max && !has_prev(it)) || (!query_max && !has_next(it))) {
79      return it;
80  }
81  hulliter it2 = it;
82  double value = it->intersect(query_max ? --it2 : *++it2);
83  line l(*it);
84  l.xlo = value;
85  hull.erase(it++);
86  return hull.insert(it, l);
87 }
88
89 public:
90 hull_optimizer(bool query_max = false) : query_max(query_max) {}
91
92 void add_line(long long m, long long b) {
93  line l(m, b, 0, false, query_max);
94  hulliter it = hull.lower_bound(l);
95  if (it != hull.end() && it->m == l.m) {
96      if ((query_max && it->b < b) || (!query_max && b < it->b)) {
97          hull.erase(it++);
98      } else {
99          return;
100     }
101 }
102 it = hull.insert(it, l);
103 if (irrelevant(it)) {
104     hull.erase(it);
105     return;
106 }
107 while (has_prev(it) && irrelevant(--it)) {
108     hull.erase(it++);
109 }
110 while (has_next(it) && irrelevant(++it)) {
111     hull.erase(it--);
112 }
113 it = update_left_border(it);
114 if (has_prev(it)) {
115     update_left_border(--it);
116 }
117 if (has_next(++it)) {
118     update_left_border(++it);
119 }
120 }
121
122 long long query(long long x) const {
123  line q(0, 0, x, true, query_max);
124  hulliter it = hull.lower_bound(q);
125  if (query_max) {
126      --it;
127  }

```

```

128     return it->m*x + it->b;
129 }
130 };
131
132 /** Example Usage ***/
133
134 #include <cassert>
135
136 int main() {
137     hull_optimizer h;
138     h.add_line(3, 0);
139     h.add_line(0, 6);
140     h.add_line(1, 2);
141     h.add_line(2, 1);
142     assert(h.query(0) == 0);
143     assert(h.query(2) == 4);
144     assert(h.query(1) == 3);
145     assert(h.query(3) == 5);
146     return 0;
147 }
```

1.4 Cycle Detection

1.4.1 Cycle Detection (Floyd)

```

1 /*
2
3 Given a function f mapping a set of integers to itself and an x-coordinate in
4 the set, return a pair containing the (position, length) of a cycle in the
5 sequence of numbers obtained from repeatedly composing f with itself starting
6 with the initial x. Formally, since f maps a finite set S to itself, some value
7 is guaranteed to eventually repeat in the sequence:
8     x[0], x[1]=f(x[0]), x[2]=f(x[1]), ..., x[n]=f(x[n - 1]), ...
9
10 There must exist a pair of indices i and j (i < j) such that x[i] = x[j]. When
11 this happens, the rest of the sequence will consist of the subsequence from x[i]
12 to x[j - 1] repeating indefinitely. The cycle detection problem asks to find
13 such an i, along with the length of the repeating subsequence. A well-known
14 special case is the problem of cycle-detection in a degenerate linked list.
15
16 Floyd's cycle-finding algorithm, a.k.a. the "tortoise and the hare algorithm",
17 is a space-efficient algorithm that moves two pointers through the sequence at
18 different speeds. Each step in the algorithm moves the "tortoise" one step
19 forward and the "hare" two steps forward in the sequence, comparing the sequence
20 values at each step. The first value which is simultaneously pointed to by both
21 pointers is the start of the sequence.
22
23 Time Complexity:
24 - O(m + n) per call to find_cycle_floyd(), where m is the smallest index of the
25   sequence which is the beginning of a cycle, and n is the cycle's length.
26
27 Space Complexity:
28 - O(1) auxiliary.
```

```
29
30 */
31
32 #include <utility>
33
34 template<class IntFunction>
35 std::pair<int, int> find_cycle_floyd(IntFunction f, int x0) {
36     int tortoise = f(x0), hare = f(f(x0));
37     while (tortoise != hare) {
38         tortoise = f(tortoise);
39         hare = f(f(hare));
40     }
41     int start = 0;
42     tortoise = x0;
43     while (tortoise != hare) {
44         tortoise = f(tortoise);
45         hare = f(hare);
46         start++;
47     }
48     int length = 1;
49     hare = f(tortoise);
50     while (tortoise != hare) {
51         hare = f(hare);
52         length++;
53     }
54     return std::make_pair(start, length);
55 }
56
57 /** Example Usage ***/
58
59 #include <cassert>
60 #include <set>
61 using namespace std;
62
63 int f(int x) {
64     return (123*x*x + 4567890) % 1337;
65 }
66
67 void verify(int x0, int start, int length) {
68     set<int> s;
69     int x = x0;
70     for (int i = 0; i < start; i++) {
71         assert(!s.count(x));
72         s.insert(x);
73         x = f(x);
74     }
75     int startx = x;
76     s.clear();
77     for (int i = 0; i < length; i++) {
78         assert(!s.count(x));
79         s.insert(x);
80         x = f(x);
81     }
82     assert(startx == x);
83 }
84
85 int main () {
86     int x0 = 0;
87     pair<int, int> res = find_cycle_floyd(f, x0);
```

```

88     assert(res == make_pair(4, 2));
89     verify(x0, res.first, res.second);
90     return 0;
91 }
```

1.4.2 Cycle Detection (Brent)

```

1  /*
2
3 Given a function f mapping a set of integers to itself and an x-coordinate in
4 the set, return a pair containing the (position, length) of a cycle in the
5 sequence of numbers obtained from repeatedly composing f with itself starting
6 with the initial x. Formally, since f maps a finite set S to itself, some value
7 is guaranteed to eventually repeat in the sequence:
8   x[0], x[1]=f(x[0]), x[2]=f(x[1]), ..., x[n]=f(x[n - 1]), ...
9
10 There must exist a pair of indices i and j (i < j) such that x[i] = x[j]. When
11 this happens, the rest of the sequence will consist of the subsequence from x[i]
12 to x[j - 1] repeating indefinitely. The cycle detection problem asks to find
13 such an i, along with the length of the repeating subsequence. A well-known
14 special case is the problem of cycle-detection in a degenerate linked list.
15
16 While Floyd's cycle-finding algorithm finds cycles by simultaneously moving two
17 pointers at different speeds, Brent's algorithm keeps the tortoise pointer
18 stationary in every iteration, only teleporting it to the hare pointer at every
19 power of two. The smallest power of two at which they meet is the start of the
20 first cycle. This improves upon the constant factor of Floyd's algorithm by
21 reducing the number of calls made to f.
22
23 Time Complexity:
24 - O(m + n) per call to find_cycle_brent(), where m is the smallest index of the
25   sequence which is the beginning of a cycle, and n is the cycle's length.
26
27 Space Complexity:
28 - O(1) auxiliary.
29
30 */
31
32 #include <utility>
33
34 template<class IntFunction>
35 std::pair<int, int> find_cycle_brent(IntFunction f, int x0) {
36     int power = 1, length = 1, tortoise = x0, hare = f(x0);
37     while (tortoise != hare) {
38         if (power == length) {
39             tortoise = hare;
40             power *= 2;
41             length = 0;
42         }
43         hare = f(hare);
44         length++;
45     }
46     hare = x0;
47     for (int i = 0; i < length; i++) {
48         hare = f(hare);
```

```
49     }
50     int start = 0;
51     tortoise = x0;
52     while (tortoise != hare) {
53         tortoise = f(tortoise);
54         hare = f(hare);
55         start++;
56     }
57     return std::make_pair(start, length);
58 }
59
60 /*** Example Usage ***/
61
62 #include <cassert>
63 #include <set>
64 using namespace std;
65
66 int f(int x) {
67     return (123*x*x + 4567890) % 1337;
68 }
69
70 void verify(int x0, int start, int length) {
71     set<int> s;
72     int x = x0;
73     for (int i = 0; i < start; i++) {
74         assert(!s.count(x));
75         s.insert(x);
76         x = f(x);
77     }
78     int startx = x;
79     s.clear();
80     for (int i = 0; i < length; i++) {
81         assert(!s.count(x));
82         s.insert(x);
83         x = f(x);
84     }
85     assert(startx == x);
86 }
87
88 int main () {
89     int x0 = 0;
90     pair<int, int> res = find_cycle_brent(f, x0);
91     assert(res == make_pair(4, 2));
92     verify(x0, res.first, res.second);
93     return 0;
94 }
```

Chapter 2

Data Structures

2.1 Heaps

2.1.1 Binary Heap

```
1  /*
2
3  Maintain a min-priority queue, that is, a collection of elements with support
4  for querying and extraction of the minimum. This implementation requires an
5  ordering on the set of possible elements defined by the < operator. A binary
6  min-heap implements a priority queue by inserting and deleting nodes into a
7  binary tree such that the parent of any node is always less than its children.
8
9  - binary_heap() constructs an empty priority queue.
10 - binary_heap(lo, hi) constructs a priority queue from two ForwardIterators,
11   consisting of elements in the range [lo, hi].
12 - size() returns the size of the priority queue.
13 - empty() returns whether the priority queue is empty.
14 - push(v) inserts the value v into the priority queue.
15 - pop() removes the minimum element from the priority queue.
16 - top() returns the minimum element in the priority queue.
17
18 Time Complexity:
19 - O(1) per call to the first constructor, size(), empty(), and top().
20 - O(log n) per call to push() and pop(), where n is the number of elements
21   in the priority queue.
22 - O(n) per call to the second constructor on the distance between lo and hi.
23
24 Space Complexity:
25 - O(n) for storage of the priority queue elements.
26 - O(1) auxiliary for all operations.
27
28 */
29
30 #include <algorithm>
31 #include <stdexcept>
32 #include <vector>
33
34 template<class T>
```

```
35 class binary_heap {
36     std::vector<T> heap;
37
38 public:
39     binary_heap() {}
40
41     template<class It>
42     binary_heap(It lo, It hi) {
43         while (lo != hi) {
44             push(*(lo++));
45         }
46     }
47
48     int size() const {
49         return heap.size();
50     }
51
52     bool empty() const {
53         return heap.empty();
54     }
55
56     void push(const T &v) {
57         heap.push_back(v);
58         int i = heap.size() - 1;
59         while (i > 0) {
60             int parent = (i - 1)/2;
61             if (!(heap[i] < heap[parent])) {
62                 break;
63             }
64             std::swap(heap[i], heap[parent]);
65             i = parent;
66         }
67     }
68
69     void pop() {
70         if (heap.empty()) {
71             throw std::runtime_error("Cannot pop from empty heap.");
72         }
73         heap[0] = heap.back();
74         heap.pop_back();
75         int i = 0;
76         for (;;) {
77             int child = 2*i + 1;
78             if (child >= (int)heap.size()) {
79                 break;
80             }
81             if (child + 1 < (int)heap.size() && heap[child + 1] < heap[child]) {
82                 child++;
83             }
84             if (heap[child] < heap[i]) {
85                 std::swap(heap[i], heap[child]);
86                 i = child;
87             } else {
88                 break;
89             }
90         }
91     }
92
93     T top() const {
```

```

94     if (heap.empty()) {
95         throw std::runtime_error("Cannot get top of empty heap.");
96     }
97     return heap[0];
98 }
99 };
100
101 /**
102  * Example Usage and Output:
103  * -1
104  * 0
105  * 5
106  * 10
107  * 12
108
109 */
110
111 #include <iostream>
112 using namespace std;
113
114 int main() {
115     int a[] = {0, 5, -1, 12};
116     binary_heap<int> h(a, a + 4);
117     h.push(10);
118     while (!h.empty()) {
119         cout << h.top() << endl;
120         h.pop();
121     }
122     return 0;
123 }
```

2.1.2 Randomized Mergeable Heap

```

1 /*
2
3 Maintain a mergeable min-priority queue, that is, a collection of elements with
4 support for querying and extraction of the minimum as well as efficient merging
5 with other instances. This implementation requires an ordering on the set of
6 possible elements defined by the < operator. A randomized mergeable heap uses a
7 coin-toss to recursively combine nodes of two binary trees.
8
9 - randomized_heap() constructs an empty priority queue.
10 - randomized_heap(lo, hi) constructs a priority queue from two ForwardIterators,
11   consisting of elements in the range [lo, hi).
12 - size() returns the size of the priority queue.
13 - empty() returns whether the priority queue is empty.
14 - push(v) inserts the value v into the priority queue.
15 - pop() removes the minimum element from the priority queue.
16 - top() returns the minimum element in the priority queue.
17 - absorb(h) inserts every value from h and sets h to the empty priority queue.
18
19 Time Complexity:
20 - O(1) per call to the first constructor, size(), empty(), and top().
21 - O(log n) expected worst case per call to push(), pop(), and absorb(), where n
22   is the number of elements in the priority queue.
```

```
23 - O(n) per call to the second constructor on the distance between lo and hi.
24
25 Space Complexity:
26 - O(n) for storage of the priority queue elements.
27 - O(log n) auxiliary stack space for push(), pop(), and absorb().
28 - O(1) auxiliary for all other operations.
29
30 */
31
32 #include <algorithm>
33 #include <cstddef>
34 #include <stdexcept>
35
36 template<class T>
37 class randomized_heap {
38     struct node_t {
39         T value;
40         node_t *left, *right;
41
42         node_t(const T &v) : value(v), left(NULL), right(NULL) {}
43     } *root;
44
45     int num_nodes;
46
47     static node_t* merge(node_t *a, node_t *b) {
48         if (a == NULL) {
49             return b;
50         }
51         if (b == NULL) {
52             return a;
53         }
54         if (b->value < a->value) {
55             std::swap(a, b);
56         }
57         if (rand() % 2 == 0) {
58             std::swap(a->left, a->right);
59         }
60         a->left = merge(a->left, b);
61         return a;
62     }
63
64     static void clean_up(node_t *n) {
65         if (n != NULL) {
66             clean_up(n->left);
67             clean_up(n->right);
68             delete n;
69         }
70     }
71
72 public:
73     randomized_heap() : root(NULL), num_nodes(0) {}
74
75     template<class It>
76     randomized_heap(It lo, It hi) : root(NULL), num_nodes(0) {
77         while (lo != hi) {
78             push(*(lo++));
79         }
80     }
81 }
```

```

82     ~randomized_heap() {
83         clean_up(root);
84     }
85
86     int size() const {
87         return num_nodes;
88     }
89
90     bool empty() const {
91         return root == NULL;
92     }
93
94     void push(const T &v) {
95         root = merge(root, new node_t(v));
96         num_nodes++;
97     }
98
99     void pop() {
100        if (empty()) {
101            throw std::runtime_error("Cannot pop from empty heap.");
102        }
103        node_t *tmp = root;
104        root = merge(root->left, root->right);
105        delete tmp;
106        num_nodes--;
107    }
108
109    T top() const {
110        if (empty()) {
111            throw std::runtime_error("Cannot get top of empty heap.");
112        }
113        return root->value;
114    }
115
116    void absorb(randomized_heap &h) {
117        root = merge(root, h.root);
118        h.root = NULL;
119    }
120 };
121
122 /**
123  -1
124  0
125  5
126  10
127  12
128 */
129
130 /**
131
132 #include <iostream>
133 using namespace std;
134
135 int main() {
136     randomized_heap<int> h, h2;
137     h.push(12);
138     h.push(10);
139     h2.push(5);
140     h2.push(-1);

```

```

141     h2.push(0);
142     h.absorb(h2);
143     while (!h.empty()) {
144         cout << h.top() << endl;
145         h.pop();
146     }
147     return 0;
148 }
```

2.1.3 Skew Heap

```

1  /*
2
3  Maintain a mergeable min-priority queue, that is, a collection of elements with
4  support for querying and extraction of the minimum as well as efficient merging
5  with other instances. This implementation requires an ordering on the set of
6  possible elements defined by the < operator. A skew heap attempts to maintain
7  balance by unconditionally swapping all nodes in the merge path when merging.
8
9  - skew_heap() constructs an empty priority queue.
10 - skew_heap(lo, hi) constructs a priority queue from two ForwardIterators,
11   consisting of elements in the range [lo, hi].
12 - size() returns the size of the priority queue.
13 - empty() returns whether the priority queue is empty.
14 - push(v) inserts the value v into the priority queue.
15 - pop() removes the minimum element from the priority queue.
16 - top() returns the minimum element in the priority queue.
17 - absorb(h) inserts every value from h and sets h to the empty priority queue.
18
19 Time Complexity:
20 - O(1) per call to the first constructor, size(), empty(), and top().
21 - O(log n) amortized auxiliary per call to push(), pop(), and absorb(), where n
22   is the number of elements in the priority queue.
23 - O(n) per call to the second constructor, where n is the distance between lo
24   and hi.
25
26 Space Complexity:
27 - O(n) for storage of the priority queue elements.
28 - O(log n) amortized auxiliary stack space for push(), pop(), and absorb().
29 - O(1) auxiliary for all other operations.
30
31 */
32
33 #include <algorithm>
34 #include <cstddef>
35 #include <stdexcept>
36
37 template<class T>
38 class skew_heap {
39     struct node_t {
40         T value;
41         node_t *left, *right;
42
43         node_t(const T &v) : value(v), left(NULL), right(NULL) {}
44     } *root;
```

```

45
46     int num_nodes;
47
48     static node_t* merge(node_t *a, node_t *b) {
49         if (a == NULL) {
50             return b;
51         }
52         if (b == NULL) {
53             return a;
54         }
55         if (b->value < a->value) {
56             std::swap(a, b);
57         }
58         std::swap(a->left, a->right);
59         a->left = merge(b, a->left);
60         return a;
61     }
62
63     static void clean_up(node_t *n) {
64         if (n != NULL) {
65             clean_up(n->left);
66             clean_up(n->right);
67             delete n;
68         }
69     }
70
71 public:
72     skew_heap() : root(NULL), num_nodes(0) {}
73
74     template<class It>
75     skew_heap(It lo, It hi) : root(NULL), num_nodes(0) {
76         while (lo != hi) {
77             push(*(lo++));
78         }
79     }
80
81     ~skew_heap() {
82         clean_up(root);
83     }
84
85     int size() const {
86         return num_nodes;
87     }
88
89     bool empty() const {
90         return root == NULL;
91     }
92
93     void push(const T &v) {
94         root = merge(root, new node_t(v));
95         num_nodes++;
96     }
97
98     void pop() {
99         if (empty()) {
100             throw std::runtime_error("Cannot pop from empty heap.");
101         }
102         node_t *tmp = root;
103         root = merge(root->left, root->right);

```

```

104     delete tmp;
105     num_nodes--;
106 }
107
108 T top() const {
109     if (empty()) {
110         throw std::runtime_error("Cannot get top of empty heap.");
111     }
112     return root->value;
113 }
114
115 void absorb(skew_heap &h) {
116     root = merge(root, h.root);
117     h.root = NULL;
118 }
119 };
120
121 /** Example Usage and Output:
122
123 -1
124 0
125 5
126 10
127 12
128
129 */
130
131 #include <iostream>
132 using namespace std;
133
134 int main() {
135     skew_heap<int> h, h2;
136     h.push(12);
137     h.push(10);
138     h2.push(5);
139     h2.push(-1);
140     h2.push(0);
141     h.absorb(h2);
142     while (!h.empty()) {
143         cout << h.top() << endl;
144         h.pop();
145     }
146     return 0;
147 }
```

2.1.4 Pairing Heap

```

1 /*
2
3 Maintain a mergeable min-priority queue, that is, a collection of elements with
4 support for querying and extraction of the minimum as well as efficient merging
5 with other instances. This implementation requires an ordering on the set of
6 possible elements defined by the < operator. A pairing heap is a heap-ordered
7 multi-way tree, using a two-pass merge to self-adjust during each deletion.
8
```

```

9 - pairing_heap() constructs an empty priority queue.
10 - pairing_heap(lo, hi) constructs a priority queue from two ForwardIterators,
11   consisting of elements in the range [lo, hi).
12 - size() returns the size of the priority queue.
13 - empty() returns whether the priority queue is empty.
14 - push(v) inserts the value v into the priority queue.
15 - pop() removes the minimum element from the priority queue.
16 - top() returns the minimum element in the priority queue.
17 - absorb(h) inserts every value from h and sets h to the empty priority queue.
18
19 Time Complexity:
20 - O(1) per call to the first constructor, size(), empty(), top(), push(), and
21   absorb().
22 - O(log n) amortized per call to pop().
23 - O(n) per call to the second constructor on the distance between lo and hi.
24
25 Space Complexity:
26 - O(n) for storage of the priority queue elements.
27 - O(log n) amortized auxiliary stack space for pop().
28 - O(1) auxiliary for all other operations.
29 */
30
31
32 #include <cstddef>
33 #include <stdexcept>
34
35 template<class T>
36 class pairing_heap {
37     struct node_t {
38         T value;
39         node_t *left, *next;
40
41         node_t(const T &v) : value(v), left(NULL), next(NULL) {}
42
43         void add_child(node_t *n) {
44             if (left == NULL) {
45                 left = n;
46             } else {
47                 n->next = left;
48                 left = n;
49             }
50         }
51     } *root;
52
53     int num_nodes;
54
55     static node_t* merge(node_t *a, node_t *b) {
56         if (a == NULL) {
57             return b;
58         }
59         if (b == NULL) {
60             return a;
61         }
62         if (a->value < b->value) {
63             a->add_child(b);
64             return a;
65         }
66         b->add_child(a);
67         return b;

```

```
68     }
69
70     static node_t* merge_pairs(node_t *n) {
71         if (n == NULL || n->next == NULL) {
72             return n;
73         }
74         node_t *a = n, *b = n->next, *c = n->next->next;
75         a->next = b->next = NULL;
76         return merge(merge(a, b), merge_pairs(c));
77     }
78
79     static void clean_up(node_t *n) {
80         if (n != NULL) {
81             clean_up(n->left);
82             clean_up(n->right);
83             delete n;
84         }
85     }
86
87     public:
88     pairing_heap() : root(NULL), num_nodes(0) {}
89
90     template<class It>
91     pairing_heap(It lo, It hi) : root(NULL), num_nodes(0) {
92         while (lo != hi) {
93             push(*(lo++));
94         }
95     }
96
97     ~pairing_heap() {
98         clean_up(root);
99     }
100
101    int size() const {
102        return num_nodes;
103    }
104
105    bool empty() const {
106        return root == NULL;
107    }
108
109    void push(const T &v) {
110        root = merge(root, new node_t(v));
111        num_nodes++;
112    }
113
114    void pop() {
115        if (empty()) {
116            throw std::runtime_error("Cannot pop from empty heap.");
117        }
118        node_t *tmp = root;
119        root = merge_pairs(root->left);
120        delete tmp;
121        num_nodes--;
122    }
123
124    T top() const {
125        if (empty()) {
126            throw std::runtime_error("Cannot get top of empty heap.");
127        }
128    }
```

```

127     }
128     return root->value;
129 }
130
131 void absorb(pairing_heap &h) {
132     root = merge(root, h.root);
133     h.root = NULL;
134 }
135 };
136
137 /** Example Usage and Output:
138
139 -1
140 0
141 5
142 10
143 12
144
145 */
146
147 #include <iostream>
148 using namespace std;
149
150 int main() {
151     pairing_heap<int> h, h2;
152     h.push(12);
153     h.push(10);
154     h2.push(5);
155     h2.push(-1);
156     h2.push(0);
157     h.absorb(h2);
158     while (!h.empty()) {
159         cout << h.top() << endl;
160         h.pop();
161     }
162     return 0;
163 }
```

2.2 Dictionaries

2.2.1 Binary Search Tree

```

1 /*
2
3 Maintain a map, that is, a collection of key-value pairs such that each possible
4 key appears at most once in the collection. This implementation requires an
5 ordering on the set of possible keys defined by the < operator on the key type.
6 A binary search tree implements this map by inserting and deleting keys into a
7 binary tree such that every node's left child has a lesser key and every node's
8 right child has a greater key.
9
10 - binary_search_tree() constructs an empty map.
11 - size() returns the size of the map.
```



```

71      }
72      if (n->left != NULL && n->right != NULL) {
73          node_t *tmp = n->right, *parent = NULL;
74          while (tmp->left != NULL) {
75              parent = tmp;
76              tmp = tmp->left;
77          }
78          n->key = tmp->key;
79          n->value = tmp->value;
80          if (parent != NULL) {
81              return erase(parent->left, parent->left->key);
82          }
83          return erase(n->right, n->right->key);
84      }
85      node_t *tmp = (n->left != NULL) ? n->left : n->right;
86      delete n;
87      n = tmp;
88      return true;
89  }
90
91  template<class KVFunction>
92  static void walk(node_t *n, KVFunction f) {
93      if (n != NULL) {
94          walk(n->left, f);
95          f(n->key, n->value);
96          walk(n->right, f);
97      }
98  }
99
100 static void clean_up(node_t *n) {
101     if (n != NULL) {
102         clean_up(n->left);
103         clean_up(n->right);
104         delete n;
105     }
106 }
107
108 public:
109     binary_search_tree() : root(NULL), num_nodes(0) {}
110
111     ~binary_search_tree() {
112         clean_up(root);
113     }
114
115     int size() const {
116         return num_nodes;
117     }
118
119     bool empty() const {
120         return root == NULL;
121     }
122
123     bool insert(const K &k, const V &v) {
124         if (insert(root, k, v)) {
125             num_nodes++;
126             return true;
127         }
128         return false;
129     }

```

```

130
131     bool erase(const K &k) {
132         if (erase(root, k)) {
133             num_nodes--;
134             return true;
135         }
136         return false;
137     }
138
139     const V* find(const K &k) const {
140         node_t *n = root;
141         while (n != NULL) {
142             if (k < n->key) {
143                 n = n->left;
144             } else if (n->key < k) {
145                 n = n->right;
146             } else {
147                 return &(n->value);
148             }
149         }
150         return NULL;
151     }
152
153     template<class KVFunction>
154     void walk(KVFunction f) const {
155         walk(root, f);
156     }
157 };
158
159 /**
160  * Example Usage and Output:
161  * abcde
162  * bcde
163  */
164
165
166 #include <cassert>
167 #include <iostream>
168 using namespace std;
169
170 void printch(int k, char v) {
171     cout << v;
172 }
173
174 int main() {
175     binary_search_tree<int, char> t;
176     t.insert(2, 'b');
177     t.insert(1, 'a');
178     t.insert(3, 'c');
179     t.insert(5, 'e');
180     assert(t.insert(4, 'd'));
181     assert(*t.find(4) == 'd');
182     assert(!t.insert(4, 'd'));
183     t.walk(printch);
184     cout << endl;
185     assert(t.erase(1));
186     assert(!t.erase(1));
187     assert(t.find(1) == NULL);
188     t.walk(printch);

```

```

189     cout << endl;
190     return 0;
191 }
```

2.2.2 Treap

```

1  /*
2
3  Maintain a map, that is, a collection of key-value pairs such that each possible
4  key appears at most once in the collection. This implementation requires an
5  ordering on the set of possible keys defined by the < operator on the key type.
6  A treap is a binary search tree that is balanced by preserving a heap property
7  on the randomly generated priority value assigned to every node, thereby making
8  insertions and deletions run in O(log n) with high probability.
9
10 - treap() constructs an empty map.
11 - size() returns the size of the map.
12 - empty() returns whether the map is empty.
13 - insert(k, v) adds an entry with key k and value v to the map, returning true
14   if a new entry was added or false if the key already exists (in which case
15   the map is unchanged and the old value associated with the key is preserved).
16 - erase(k) removes the entry with key k from the map, returning true if the
17   removal was successful or false if the key to be removed was not found.
18 - find(k) returns a pointer to a const value associated with key k, or NULL if
19   the key was not found.
20 - walk(f) calls the function f(k, v) on each entry of the map, in ascending
21   order of keys.
22
23 Time Complexity:
24 - O(1) per call to the constructor, size(), and empty().
25 - O(log n) on average per call to insert(), erase(), and find(), where n is the
26   number of entries currently in the map.
27 - O(n) per call to walk().
28
29 Space Complexity:
30 - O(n) for storage of the map elements.
31 - O(log n) auxiliary stack space on average for insert(), erase(), and walk().
32 - O(1) auxiliary for all other operations.
33 */
34
35
36 #include <cstdlib>
37
38 template<class K, class V>
39 class treap {
40     struct node_t {
41         static inline int rand32() {
42             return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
43         }
44
45         K key;
46         V value;
47         int priority;
48         node_t *left, *right;
49     };
50
51     node_t *root;
52
53     void rotate_left(node_t*& root);
54     void rotate_right(node_t*& root);
55
56     void insert(node_t*& root, const K& key, const V& value);
57     void erase(node_t*& root, const K& key);
58
59     V* find(const K& key);
60
61     void walk(node_t*& root, void (*f)(const K& key, const V& value));
62
63     int size();
64     bool empty();
65
66     node_t* clone(node_t* root);
67
68     void print(node_t* root);
69
70     void print_inorder(node_t* root);
71
72     void print_preorder(node_t* root);
73
74     void print_postorder(node_t* root);
75
76     void print_levelorder(node_t* root);
77
78     void print_inorder(node_t* root);
79
80     void print_preorder(node_t* root);
81
82     void print_postorder(node_t* root);
83
84     void print_levelorder(node_t* root);
85
86     void print_inorder(node_t* root);
87
88     void print_preorder(node_t* root);
89
90     void print_postorder(node_t* root);
91
92     void print_levelorder(node_t* root);
93
94     void print_inorder(node_t* root);
95
96     void print_preorder(node_t* root);
97
98     void print_postorder(node_t* root);
99
100    void print_levelorder(node_t* root);
101
102    void print_inorder(node_t* root);
103
104    void print_preorder(node_t* root);
105
106    void print_postorder(node_t* root);
107
108    void print_levelorder(node_t* root);
109
110    void print_inorder(node_t* root);
111
112    void print_preorder(node_t* root);
113
114    void print_postorder(node_t* root);
115
116    void print_levelorder(node_t* root);
117
118    void print_inorder(node_t* root);
119
120    void print_preorder(node_t* root);
121
122    void print_postorder(node_t* root);
123
124    void print_levelorder(node_t* root);
125
126    void print_inorder(node_t* root);
127
128    void print_preorder(node_t* root);
129
130    void print_postorder(node_t* root);
131
132    void print_levelorder(node_t* root);
133
134    void print_inorder(node_t* root);
135
136    void print_preorder(node_t* root);
137
138    void print_postorder(node_t* root);
139
140    void print_levelorder(node_t* root);
141
142    void print_inorder(node_t* root);
143
144    void print_preorder(node_t* root);
145
146    void print_postorder(node_t* root);
147
148    void print_levelorder(node_t* root);
149
150    void print_inorder(node_t* root);
151
152    void print_preorder(node_t* root);
153
154    void print_postorder(node_t* root);
155
156    void print_levelorder(node_t* root);
157
158    void print_inorder(node_t* root);
159
160    void print_preorder(node_t* root);
161
162    void print_postorder(node_t* root);
163
164    void print_levelorder(node_t* root);
165
166    void print_inorder(node_t* root);
167
168    void print_preorder(node_t* root);
169
170    void print_postorder(node_t* root);
171
172    void print_levelorder(node_t* root);
173
174    void print_inorder(node_t* root);
175
176    void print_preorder(node_t* root);
177
178    void print_postorder(node_t* root);
179
180    void print_levelorder(node_t* root);
181
182    void print_inorder(node_t* root);
183
184    void print_preorder(node_t* root);
185
186    void print_postorder(node_t* root);
187
188    void print_levelorder(node_t* root);
189
190    void print_inorder(node_t* root);
191
192    void print_preorder(node_t* root);
193
194    void print_postorder(node_t* root);
195
196    void print_levelorder(node_t* root);
197
198    void print_inorder(node_t* root);
199
200    void print_preorder(node_t* root);
201
202    void print_postorder(node_t* root);
203
204    void print_levelorder(node_t* root);
205
206    void print_inorder(node_t* root);
207
208    void print_preorder(node_t* root);
209
210    void print_postorder(node_t* root);
211
212    void print_levelorder(node_t* root);
213
214    void print_inorder(node_t* root);
215
216    void print_preorder(node_t* root);
217
218    void print_postorder(node_t* root);
219
220    void print_levelorder(node_t* root);
221
222    void print_inorder(node_t* root);
223
224    void print_preorder(node_t* root);
225
226    void print_postorder(node_t* root);
227
228    void print_levelorder(node_t* root);
229
230    void print_inorder(node_t* root);
231
232    void print_preorder(node_t* root);
233
234    void print_postorder(node_t* root);
235
236    void print_levelorder(node_t* root);
237
238    void print_inorder(node_t* root);
239
240    void print_preorder(node_t* root);
241
242    void print_postorder(node_t* root);
243
244    void print_levelorder(node_t* root);
245
246    void print_inorder(node_t* root);
247
248    void print_preorder(node_t* root);
249
250    void print_postorder(node_t* root);
251
252    void print_levelorder(node_t* root);
253
254    void print_inorder(node_t* root);
255
256    void print_preorder(node_t* root);
257
258    void print_postorder(node_t* root);
259
260    void print_levelorder(node_t* root);
261
262    void print_inorder(node_t* root);
263
264    void print_preorder(node_t* root);
265
266    void print_postorder(node_t* root);
267
268    void print_levelorder(node_t* root);
269
270    void print_inorder(node_t* root);
271
272    void print_preorder(node_t* root);
273
274    void print_postorder(node_t* root);
275
276    void print_levelorder(node_t* root);
277
278    void print_inorder(node_t* root);
279
280    void print_preorder(node_t* root);
281
282    void print_postorder(node_t* root);
283
284    void print_levelorder(node_t* root);
285
286    void print_inorder(node_t* root);
287
288    void print_preorder(node_t* root);
289
290    void print_postorder(node_t* root);
291
292    void print_levelorder(node_t* root);
293
294    void print_inorder(node_t* root);
295
296    void print_preorder(node_t* root);
297
298    void print_postorder(node_t* root);
299
299 }
```

```

50     node_t(const K &k, const V &v)
51         : key(k), value(v), priority(rand32()), left(NULL), right(NULL) {}
52     } *root;
53
54     int num_nodes;
55
56     static void rotate_left(node_t *&n) {
57         node_t *tmp = n;
58         n = n->right;
59         tmp->right = n->left;
60         n->left = tmp;
61     }
62
63     static void rotate_right(node_t *&n) {
64         node_t *tmp = n;
65         n = n->left;
66         tmp->left = n->right;
67         n->right = tmp;
68     }
69
70     static bool insert(node_t *&n, const K &k, const V &v) {
71         if (n == NULL) {
72             n = new node_t(k, v);
73             return true;
74         }
75         if (k < n->key && insert(n->left, k, v)) {
76             if (n->left->priority < n->priority) {
77                 rotate_right(n);
78             }
79             return true;
80         }
81         if (n->key < k && insert(n->right, k, v)) {
82             if (n->right->priority < n->priority) {
83                 rotate_left(n);
84             }
85             return true;
86         }
87         return false;
88     }
89
90     static bool erase(node_t *&n, const K &k) {
91         if (n == NULL) {
92             return false;
93         }
94         if (k < n->key) {
95             return erase(n->left, k);
96         } else if (n->key < k) {
97             return erase(n->right, k);
98         }
99         if (n->left != NULL && n->right != NULL) {
100             if (n->left->priority < n->right->priority) {
101                 rotate_right(n);
102                 return erase(n->right, k);
103             }
104             rotate_left(n);
105             return erase(n->left, k);
106         }
107         node_t *tmp = (n->left != NULL) ? n->left : n->right;
108         delete n;

```

```

109     n = tmp;
110     return true;
111 }
112
113 template<class KVFunction>
114 static void walk(node_t *n, KVFunction f) {
115     if (n != NULL) {
116         walk(n->left, f);
117         f(n->key, n->value);
118         walk(n->right, f);
119     }
120 }
121
122 static void clean_up(node_t *n) {
123     if (n != NULL) {
124         clean_up(n->left);
125         clean_up(n->right);
126         delete n;
127     }
128 }
129
130 public:
131     treap() : root(NULL), num_nodes(0) {}
132
133     ~treap() {
134         clean_up(root);
135     }
136
137     int size() const {
138         return num_nodes;
139     }
140
141     bool empty() const {
142         return root == NULL;
143     }
144
145     bool insert(const K &k, const V &v) {
146         if (insert(root, k, v)) {
147             num_nodes++;
148             return true;
149         }
150         return false;
151     }
152
153     bool erase(const K &k) {
154         if (erase(root, k)) {
155             num_nodes--;
156             return true;
157         }
158         return false;
159     }
160
161     const V* find(const K &k) const {
162         node_t *n = root;
163         while (n != NULL) {
164             if (k < n->key) {
165                 n = n->left;
166             } else if (n->key < k) {
167                 n = n->right;
168             }
169         }
170         return n->value;
171     }

```

```

168     } else {
169         return &(n->value);
170     }
171 }
172 return NULL;
173 }
174
175 template<class KVFunction>
176 void walk(KVFunction f) const {
177     walk(root, f);
178 }
179 };
180
181 /** Example Usage and Output:
182
183 abcde
184 bcde
185
186 */
187
188 #include <cassert>
189 #include <iostream>
190 using namespace std;
191
192 void printch(int k, char v) {
193     cout << v;
194 }
195
196 int main() {
197     treap<int, char> t;
198     t.insert(2, 'b');
199     t.insert(1, 'a');
200     t.insert(3, 'c');
201     t.insert(5, 'e');
202     assert(t.insert(4, 'd'));
203     assert(*t.find(4) == 'd');
204     assert(!t.insert(4, 'd'));
205     t.walk(printch);
206     cout << endl;
207     assert(t.erase(1));
208     assert(!t.erase(1));
209     assert(t.find(1) == NULL);
210     t.walk(printch);
211     cout << endl;
212     return 0;
213 }
```

2.2.3 AVL Tree

```

1 /*
2
3 Maintain a map, that is, a collection of key-value pairs such that each possible
4 key appears at most once in the collection. This implementation requires an
5 ordering on the set of possible keys defined by the < operator on the key type.
6 An AVL tree is a binary search tree balanced by height, guaranteeing O(log n)
```

```

7 worst-case running time in insertions and deletions by making sure that the
8 heights of the left and right subtrees at every node differ by at most 1.
9
10 - avl_tree() constructs an empty map.
11 - size() returns the size of the map.
12 - empty() returns whether the map is empty.
13 - insert(k, v) adds an entry with key k and value v to the map, returning true
14   if a new entry was added or false if the key already exists (in which case
15   the map is unchanged and the old value associated with the key is preserved).
16 - erase(k) removes the entry with key k from the map, returning true if the
17   removal was successful or false if the key to be removed was not found.
18 - find(k) returns a pointer to a const value associated with key k, or NULL if
19   the key was not found.
20 - walk(f) calls the function f(k, v) on each entry of the map, in ascending
21   order of keys.
22
23 Time Complexity:
24 - O(1) per call to the constructor, size(), and empty().
25 - O(log n) per call to insert(), erase(), and find(), where n is the number of
26   entries currently in the map.
27 - O(n) per call to walk().
28
29 Space Complexity:
30 - O(n) for storage of the map elements.
31 - O(log n) auxiliary stack space for insert(), erase(), and walk().
32 - O(1) auxiliary for all other operations.
33
34 */
35
36 #include <algorithm>
37 #include <cstddef>
38
39 template<class K, class V>
40 class avl_tree {
41     struct node_t {
42         K key;
43         V value;
44         int height;
45         node_t *left, *right;
46
47         node_t(const K &k, const V &v)
48             : key(k), value(v), height(1), left(NULL), right(NULL) {}
49     } *root;
50
51     int num_nodes;
52
53     static int height(node_t *n) {
54         return (n != NULL) ? n->height : 0;
55     }
56
57     static void update_height(node_t *n) {
58         if (n != NULL) {
59             n->height = 1 + std::max(height(n->left), height(n->right));
60         }
61     }
62
63     static void rotate_left(node_t *&n) {
64         node_t *tmp = n;
65         n = n->right;

```

```

66     tmp->right = n->left;
67     n->left = tmp;
68     update_height(tmp);
69     update_height(n);
70 }
71
72 static void rotate_right(node_t *&n) {
73     node_t *tmp = n;
74     n = n->left;
75     tmp->left = n->right;
76     n->right = tmp;
77     update_height(tmp);
78     update_height(n);
79 }
80
81 static int balance_factor(node_t *n) {
82     return (n != NULL) ? (height(n->left) - height(n->right)) : 0;
83 }
84
85 static void rebalance(node_t *&n) {
86     if (n == NULL) {
87         return;
88     }
89     update_height(n);
90     int bf = balance_factor(n);
91     if (bf > 1 && balance_factor(n->left) >= 0) {
92         rotate_right(n);
93     } else if (bf > 1 && balance_factor(n->left) < 0) {
94         rotate_left(n->left);
95         rotate_right(n);
96     } else if (bf < -1 && balance_factor(n->right) <= 0) {
97         rotate_left(n);
98     } else if (bf < -1 && balance_factor(n->right) > 0) {
99         rotate_right(n->right);
100        rotate_left(n);
101    }
102 }
103
104 static bool insert(node_t *&n, const K &k, const V &v) {
105     if (n == NULL) {
106         n = new node_t(k, v);
107         return true;
108     }
109     if ((k < n->key && insert(n->left, k, v)) ||
110         (n->key < k && insert(n->right, k, v))) {
111         rebalance(n);
112         return true;
113     }
114     return false;
115 }
116
117 static bool erase(node_t *&n, const K &k) {
118     if (n == NULL) {
119         return false;
120     }
121     if (!(k < n->key || n->key < k)) {
122         if (n->left != NULL && n->right != NULL) {
123             node_t *tmp = n->right, *parent = NULL;
124             while (tmp->left != NULL) {

```

```

125         parent = tmp;
126         tmp = tmp->left;
127     }
128     n->key = tmp->key;
129     n->value = tmp->value;
130     if (parent != NULL) {
131         if (!erase(parent->left, parent->left->key)) {
132             return false;
133         }
134     } else if (!erase(n->right, n->right->key)) {
135         return false;
136     }
137 } else {
138     node_t *tmp = (n->left != NULL) ? n->left : n->right;
139     delete n;
140     n = tmp;
141 }
142 rebalance(n);
143 return true;
144 }
145 if ((k < n->key && erase(n->left, k)) ||
146     (n->key < k && erase(n->right, k))) {
147     rebalance(n);
148     return true;
149 }
150 return false;
151 }
152
153 template<class KVFunction>
154 static void walk(node_t *n, KVFunction f) {
155     if (n != NULL) {
156         walk(n->left, f);
157         f(n->key, n->value);
158         walk(n->right, f);
159     }
160 }
161
162 static void clean_up(node_t *n) {
163     if (n != NULL) {
164         clean_up(n->left);
165         clean_up(n->right);
166         delete n;
167     }
168 }
169
170 public:
171     avl_tree() : root(NULL), num_nodes(0) {}
172
173     ~avl_tree() {
174         clean_up(root);
175     }
176
177     int size() const {
178         return num_nodes;
179     }
180
181     bool empty() const {
182         return root == NULL;
183     }

```

```

184
185     bool insert(const K &k, const V &v) {
186         if (insert(root, k, v)) {
187             num_nodes++;
188             return true;
189         }
190         return false;
191     }
192
193     bool erase(const K &k) {
194         if (erase(root, k)) {
195             num_nodes--;
196             return true;
197         }
198         return false;
199     }
200
201     const V* find(const K &k) const {
202         node_t *n = root;
203         while (n != NULL) {
204             if (k < n->key) {
205                 n = n->left;
206             } else if (n->key < k) {
207                 n = n->right;
208             } else {
209                 return &(n->value);
210             }
211         }
212         return NULL;
213     }
214
215     template<class KVFunction>
216     void walk(KVFunction f) const {
217         walk(root, f);
218     }
219 };
220
221 /**
222  abcde
223  bcde
224 */
225
226 /**
227
228 #include <cassert>
229 #include <iostream>
230 using namespace std;
231
232 void printch(int k, char v) {
233     cout << v;
234 }
235
236 int main() {
237     avl_tree<int, char> t;
238     t.insert(2, 'b');
239     t.insert(1, 'a');
240     t.insert(3, 'c');
241     t.insert(5, 'e');
242     assert(t.insert(4, 'd'));

```

```

243     assert(*t.find(4) == 'd');
244     assert(!t.insert(4, 'd'));
245     t.walk(printch);
246     cout << endl;
247     assert(t.erase(1));
248     assert(!t.erase(1));
249     assert(t.find(1) == NULL);
250     t.walk(printch);
251     cout << endl;
252     return 0;
253 }
```

2.2.4 Red-Black Tree

```

1  /*
2
3  Maintain a map, that is, a collection of key-value pairs such that each possible
4  key appears at most once in the collection. This implementation requires an
5  ordering on the set of possible keys defined by the < operator on the key type.
6  A red black tree is a binary search tree balanced by coloring its nodes red or
7  black, then constraining node colors on any simple path from the root to a leaf.
8
9  - red_black_tree() constructs an empty map.
10 - size() returns the size of the map.
11 - empty() returns whether the map is empty.
12 - insert(k, v) adds an entry with key k and value v to the map, returning true
13   if a new entry was added or false if the key already exists (in which case
14   the map is unchanged and the old value associated with the key is preserved).
15 - erase(k) removes the entry with key k from the map, returning true if the
16   removal was successful or false if the key to be removed was not found.
17 - find(k) returns a pointer to a const value associated with key k, or NULL if
18   the key was not found.
19 - walk(f) calls the function f(k, v) on each entry of the map, in ascending
20   order of keys.
21
22 Time Complexity:
23 - O(1) per call to the constructor, size(), and empty().
24 - O(log n) per call to insert(), erase(), and find(), where n is the number of
25   entries currently in the map.
26 - O(n) per call to walk().
27
28 Space Complexity:
29 - O(n) for storage of the map elements.
30 - O(log n) auxiliary stack space for walk().
31 - O(1) auxiliary for all other operations.
32
33 */
34
35 #include <algorithm>
36 #include <cstddef>
37
38 template<class K, class V>
39 class red_black_tree {
40     enum color_t { RED, BLACK };
41     struct node_t {
```

```

42 K key;
43 V value;
44 color_t color;
45 node_t *left, *right, *parent;
46
47 node_t(const K &k, const V &v, color_t c)
48     : key(k), value(v), color(c), left(NULL), right(NULL), parent(NULL) {}
49 } *root, *LEAF_NIL;
50
51 int num_nodes;
52
53 void rotate_left(node_t *n) {
54     node_t *tmp = n->right;
55     if ((n->right = tmp->left) != LEAF_NIL) {
56         n->right->parent = n;
57     }
58     if ((tmp->parent = n->parent) == LEAF_NIL) {
59         root = tmp;
60     } else if (n->parent->left == n) {
61         n->parent->left = tmp;
62     } else {
63         n->parent->right = tmp;
64     }
65     tmp->left = n;
66     n->parent = tmp;
67 }
68
69 void rotate_right(node_t *n) {
70     node_t *tmp = n->left;
71     if ((n->left = tmp->right) != LEAF_NIL) {
72         n->left->parent = n;
73     }
74     if ((tmp->parent = n->parent) == LEAF_NIL) {
75         root = tmp;
76     } else if (n->parent->right == n) {
77         n->parent->right = tmp;
78     } else {
79         n->parent->left = tmp;
80     }
81     tmp->right = n;
82     n->parent = tmp;
83 }
84
85 void insert_fix(node_t *n) {
86     while (n->parent->color == RED) {
87         node_t *parent = n->parent;
88         node_t *grandparent = n->parent->parent;
89         if (parent == grandparent->left) {
90             node_t *uncle = grandparent->right;
91             if (uncle->color == RED) {
92                 grandparent->color = RED;
93                 parent->color = BLACK;
94                 uncle->color = BLACK;
95                 n = grandparent;
96             } else {
97                 if (n == parent->right) {
98                     rotate_left(parent);
99                     n = parent;
100                    parent = n->parent;
101                }
102            }
103        }
104    }
105 }
```

```

101     }
102     rotate_right(grandparent);
103     std::swap(parent->color, grandparent->color);
104     n = parent;
105 }
106 } else if (parent == grandparent->right) {
107     node_t *uncle = grandparent->left;
108     if (uncle->color == RED) {
109         grandparent->color = RED;
110         parent->color = BLACK;
111         uncle->color = BLACK;
112         n = grandparent;
113     } else {
114         if (n == parent->left) {
115             rotate_right(parent);
116             n = parent;
117             parent = n->parent;
118         }
119         rotate_left(grandparent);
120         std::swap(parent->color, grandparent->color);
121         n = parent;
122     }
123 }
124 root->color = BLACK;
125 }
126 }
127
128 void replace(node_t *n, node_t *replacement) {
129     if (n->parent == LEAF_NIL) {
130         root = replacement;
131     } else if (n == n->parent->left) {
132         n->parent->left = replacement;
133     } else {
134         n->parent->right = replacement;
135     }
136     replacement->parent = n->parent;
137 }
138
139 void erase_fix(node_t *n) {
140     while (n != root && n->color == BLACK) {
141         node_t *parent = n->parent;
142         if (n == parent->left) {
143             node_t *sibling = parent->right;
144             if (sibling->color == RED) {
145                 sibling->color = BLACK;
146                 parent->color = RED;
147                 rotate_left(parent);
148                 sibling = parent->right;
149             }
150             if (sibling->left->color == BLACK && sibling->right->color == BLACK) {
151                 sibling->color = RED;
152                 n = parent;
153             } else {
154                 if (sibling->right->color == BLACK) {
155                     sibling->left->color = BLACK;
156                     sibling->color = RED;
157                     rotate_right(sibling);
158                     sibling = parent->right;
159                 }

```

```

160         sibling->color = parent->color;
161         parent->color = BLACK;
162         sibling->right->color = BLACK;
163         rotate_left(parent);
164         n = root;
165     }
166 } else {
167     node_t *sibling = parent->left;
168     if (sibling->color == RED) {
169         sibling->color = BLACK;
170         parent->color = RED;
171         rotate_right(parent);
172         sibling = parent->left;
173     }
174     if (sibling->left->color == BLACK && sibling->right->color == BLACK) {
175         sibling->color = RED;
176         n = parent;
177     } else {
178         if (sibling->left->color == BLACK) {
179             sibling->right->color = BLACK;
180             sibling->color = RED;
181             rotate_left(sibling);
182             sibling = parent->left;
183         }
184         sibling->color = parent->color;
185         parent->color = BLACK;
186         sibling->left->color = BLACK;
187         rotate_right(parent);
188         n = root;
189     }
190 }
191 }
192 n->color = BLACK;
193 }

194 template<class KVFunction>
195 void walk(node_t *n, KVFunction f) const {
196     if (n != LEAF_NIL) {
197         walk(n->left, f);
198         f(n->key, n->value);
199         walk(n->right, f);
200     }
201 }
202 }

203 void clean_up(node_t *n) {
204     if (n != LEAF_NIL) {
205         clean_up(n->left);
206         clean_up(n->right);
207         delete n;
208     }
209 }
210 }

211 public:
212     red_black_tree() : num_nodes(0) {
213         root = LEAF_NIL = new node_t(K(), V(), BLACK);
214     }
215
216     ~red_black_tree() {
217         clean_up(root);
218     }

```

```

219     delete LEAF_NIL;
220 }
221
222 int size() const {
223     return num_nodes;
224 }
225
226 bool empty() const {
227     return num_nodes == 0;
228 }
229
230 bool insert(const K &k, const V &v) {
231     node_t *curr = root, *prev = LEAF_NIL;
232     while (curr != LEAF_NIL) {
233         prev = curr;
234         if (k < curr->key) {
235             curr = curr->left;
236         } else if (curr->key < k) {
237             curr = curr->right;
238         } else {
239             return false;
240         }
241     }
242     node_t *n = new node_t(k, v, RED);
243     n->parent = prev;
244     if (prev == LEAF_NIL) {
245         root = n;
246     } else if (k < prev->key) {
247         prev->left = n;
248     } else {
249         prev->right = n;
250     }
251     n->left = n->right = LEAF_NIL;
252     insert_fix(n);
253     num_nodes++;
254     return true;
255 }
256
257 bool erase(const K &k) {
258     node_t *n = root;
259     while (n != LEAF_NIL) {
260         if (k < n->key) {
261             n = n->left;
262         } else if (n->key < k) {
263             n = n->right;
264         } else {
265             break;
266         }
267     }
268     if (n == LEAF_NIL) {
269         return false;
270     }
271     color_t color = n->color;
272     node_t *replacement;
273     if (n->left == LEAF_NIL) {
274         replacement = n->right;
275         replace(n, n->right);
276     } else if (n->right == LEAF_NIL) {
277         replacement = n->left;

```

```

278     replace(n, n->left);
279 } else {
280     node_t *tmp = n->right;
281     while (tmp->left != LEAF_NIL) {
282         tmp = tmp->left;
283     }
284     color = tmp->color;
285     replacement = tmp->right;
286     if (tmp->parent == n) {
287         replacement->parent = tmp;
288     } else {
289         replace(tmp, tmp->right);
290         tmp->right = n->right;
291         tmp->right->parent = tmp;
292     }
293     replace(n, tmp);
294     tmp->left = n->left;
295     tmp->left->parent = tmp;
296     tmp->color = n->color;
297 }
298 delete n;
299 if (color == BLACK) {
300     erase_fix(replacement);
301 }
302 return true;
303 }

304 const V* find(const K &k) const {
305     node_t *n = root;
306     while (n != LEAF_NIL) {
307         if (k < n->key) {
308             n = n->left;
309         } else if (n->key < k) {
310             n = n->right;
311         } else {
312             return &(n->value);
313         }
314     }
315     return NULL;
316 }
317 }

318 template<class KVFunction>
319 void walk(KVFunction f) const {
320     walk(root, f);
321 }
322 };
323 };

324 /**
325 *** Example Usage and Output:
326
327 abcde
328 bcde
329 ***
330 */
331
332 #include <cassert>
333 #include <iostream>
334 using namespace std;
335
336 void printch(int k, char v) {

```

```

337     cout << v;
338 }
339
340 int main() {
341     red_black_tree<int, char> t;
342     t.insert(2, 'b');
343     t.insert(1, 'a');
344     t.insert(3, 'c');
345     t.insert(5, 'e');
346     assert(t.insert(4, 'd'));
347     assert(*t.find(4) == 'd');
348     assert(!t.insert(4, 'd'));
349     t.walk(printch);
350     cout << endl;
351     assert(t.erase(1));
352     assert(!t.erase(1));
353     assert(t.find(1) == NULL);
354     t.walk(printch);
355     cout << endl;
356     return 0;
357 }
```

2.2.5 Splay Tree

```

1 /*
2
3 Maintain a map, that is, a collection of key-value pairs such that each possible
4 key appears at most once in the collection. This implementation requires an
5 ordering on the set of possible keys defined by the < operator on the key type.
6 A splay tree is a balanced binary search tree with the additional property that
7 recently accessed elements are quick to access again.
8
9 - splay_tree() constructs an empty map.
10 - size() returns the size of the map.
11 - empty() returns whether the map is empty.
12 - insert(k, v) adds an entry with key k and value v to the map, returning true
13   if a new entry was added or false if the key already exists (in which case
14   the map is unchanged and the old value associated with the key is preserved).
15 - erase(k) removes the entry with key k from the map, returning true if the
16   removal was successful or false if the key to be removed was not found.
17 - find(k) returns a pointer to a const value associated with key k, or NULL if
18   the key was not found.
19 - walk(f) calls the function f(k, v) on each entry of the map, in ascending
20   order of keys.
21
22 Time Complexity:
23 - O(1) per call to the constructor, size(), and empty().
24 - O(log n) per call to insert(), erase(), and find(), where n is the number of
25   entries currently in the map.
26 - O(n) per call to walk().
27
28 Space Complexity:
29 - O(n) for storage of the map elements.
30 - O(log n) auxiliary stack space for insert(), erase(), and walk().
31 - O(1) auxiliary for all other operations.
```

```
32
33 */
34
35 #include <cstddef>
36
37 template<class K, class V>
38 class splay_tree {
39     struct node_t {
40         K key;
41         V value;
42         node_t *left, *right;
43
44         node_t(const K &k, const V &v)
45             : key(k), value(v), left(NULL), right(NULL) {}
46     } *root;
47
48     int num_nodes;
49
50     static void rotate_left(node_t *&n) {
51         node_t *tmp = n;
52         n = n->right;
53         tmp->right = n->left;
54         n->left = tmp;
55     }
56
57     static void rotate_right(node_t *&n) {
58         node_t *tmp = n;
59         n = n->left;
60         tmp->left = n->right;
61         n->right = tmp;
62     }
63
64     static void splay(node_t *&n, const K &k) {
65         if (n == NULL) {
66             return;
67         }
68         if (k < n->key && n->left != NULL) {
69             if (k < n->left->key) {
70                 splay(n->left->left, k);
71                 rotate_right(n);
72             } else if (n->left->key < k) {
73                 splay(n->left->right, k);
74                 if (n->left->right != NULL) {
75                     rotate_left(n->left);
76                 }
77             }
78             if (n->left != NULL) {
79                 rotate_right(n);
80             }
81         } else if (n->key < k && n->right != NULL) {
82             if (k < n->right->key) {
83                 splay(n->right->left, k);
84                 if (n->right->left != NULL) {
85                     rotate_right(n->right);
86                 }
87             } else if (n->right->key < k) {
88                 splay(n->right->right, k);
89                 rotate_left(n);
90             }
91     }
92 }
```

```

91     if (n->right != NULL) {
92         rotate_left(n);
93     }
94 }
95 }

96 static bool insert(node_t *&n, const K &k, const V &v) {
97     if (n == NULL) {
98         n = new node_t(k, v);
99         return true;
100    }
101    splay(n, k);
102    if (k < n->key) {
103        node_t *tmp = new node_t(k, v);
104        tmp->left = n->left;
105        tmp->right = n;
106        n->left = NULL;
107        n = tmp;
108    } else if (n->key < k) {
109        node_t *tmp = new node_t(k, v);
110        tmp->left = n;
111        tmp->right = n->right;
112        n->right = NULL;
113        n = tmp;
114    } else {
115        return false;
116    }
117    return true;
118 }
119 }

120 static bool erase(node_t *&n, const K &k) {
121     if (n == NULL) {
122         return false;
123     }
124     splay(n, k);
125     if (k < n->key || n->key < k) {
126         return false;
127     }
128     node_t *tmp = n;
129     if (n->left == NULL) {
130         n = n->right;
131     } else {
132         splay(n->left, k);
133         n = n->left;
134         n->right = tmp->right;
135     }
136     delete tmp;
137     return true;
138 }
139 }

140 template<class KVFunction>
141 static void walk(node_t *n, KVFunction f) {
142     if (n != NULL) {
143         walk(n->left, f);
144         f(n->key, n->value);
145         walk(n->right, f);
146     }
147 }
148 }

149

```

```

150     static void clean_up(node_t *n) {
151         if (n != NULL) {
152             clean_up(n->left);
153             clean_up(n->right);
154             delete n;
155         }
156     }
157
158     public:
159     splay_tree() : root(NULL), num_nodes(0) {}
160
161     ~splay_tree() {
162         clean_up(root);
163     }
164
165     int size() const {
166         return num_nodes;
167     }
168
169     bool empty() const {
170         return root == NULL;
171     }
172
173     bool insert(const K &k, const V &v) {
174         if (insert(root, k, v)) {
175             num_nodes++;
176             return true;
177         }
178         return false;
179     }
180
181     bool erase(const K &k) {
182         if (erase(root, k)) {
183             num_nodes--;
184             return true;
185         }
186         return false;
187     }
188
189     const V* find(const K &k) {
190         splay(root, k);
191         return (k < root->key || root->key < k) ? NULL : &(root->value);
192     }
193
194     template<class KVFunction>
195     void walk(KVFunction f) const {
196         walk(root, f);
197     }
198 };
199
200 /**
201  abcde
202  bcde
203
204  */
205
206
207 #include <cassert>
208 #include <iostream>

```

```

209 using namespace std;
210
211 void printch(int k, char v) {
212     cout << v;
213 }
214
215 int main() {
216     splay_tree<int, char> t;
217     t.insert(2, 'b');
218     t.insert(1, 'a');
219     t.insert(3, 'c');
220     t.insert(5, 'e');
221     assert(t.insert(4, 'd'));
222     assert(*t.find(4) == 'd');
223     assert(!t.insert(4, 'd'));
224     t.walk(printch);
225     cout << endl;
226     assert(t.erase(1));
227     assert(!t.erase(1));
228     assert(t.find(1) == NULL);
229     t.walk(printch);
230     cout << endl;
231     return 0;
232 }
```

2.2.6 Size Balanced Tree

```

1 /*
2
3 Maintain a map, that is, a collection of key-value pairs such that each possible
4 key appears at most once in the collection. In addition, support queries for
5 keys given their ranks as well as queries for the ranks of given keys. This
6 implementation requires an ordering on the set of possible keys defined by the
7 < operator on the key type. A size balanced tree augments each nodes with the
8 size of its subtree, using it to maintain balance and compute order statistics.
9
10 - size_balanced_tree() constructs an empty map.
11 - size() returns the size of the map.
12 - empty() returns whether the map is empty.
13 - insert(k, v) adds an entry with key k and value v to the map, returning true
14   if a new entry was added or false if the key already exists (in which case
15   the map is unchanged and the old value associated with the key is preserved).
16 - erase(k) removes the entry with key k from the map, returning true if the
17   removal was successful or false if the key to be removed was not found.
18 - find(k) returns a pointer to a const value associated with key k, or NULL if
19   the key was not found.
20 - select(r) returns a key-value pair of the node with a key of zero-based rank r
21   in the map, throwing an exception if the rank is not between 0 and size() - 1.
22 - rank(k) returns the zero-based rank of key k in the map, throwing an
23   exception if the key was not found in the map.
24 - walk(f) calls the function f(k, v) on each entry of the map, in ascending
25   order of keys.
26
27 Time Complexity:
28 - O(1) per call to the constructor, size(), and empty().
```

```

29 -  $O(\log n)$  per call to insert(), erase(), find(), select(), and rank(), where  $n$ 
30 is the number of entries currently in the map.
31 -  $O(n)$  per call to walk().
32
33 Space Complexity:
34 -  $O(n)$  for storage of the map elements.
35 -  $O(\log n)$  auxiliary stack space for insert(), erase(), and walk().
36 -  $O(1)$  auxiliary for all other operations.
37
38 */
39
40 #include <cstddef>
41 #include <stdexcept>
42 #include <utility>
43
44 template<class K, class V>
45 class size_balanced_tree {
46     struct node_t {
47         K key;
48         V value;
49         int size;
50         node_t *left, *right;
51
52         node_t(const K &k, const V &v)
53             : key(k), value(v), size(1), left(NULL), right(NULL) {}
54
55         inline node_t*& child(int c) {
56             return (c == 0) ? left : right;
57         }
58
59         void update() {
60             size = 1;
61             if (left != NULL) {
62                 size += left->size;
63             }
64             if (right != NULL) {
65                 size += right->size;
66             }
67         }
68     } *root;
69
70     static inline int size(node_t *n) {
71         return (n == NULL) ? 0 : n->size;
72     }
73
74     static void rotate(node_t *&n, int c) {
75         node_t *tmp = n->child(c);
76         n->child(c) = tmp->child(!c);
77         tmp->child(!c) = n;
78         n->update();
79         tmp->update();
80         n = tmp;
81     }
82
83     static void maintain(node_t *&n, int c) {
84         if (n == NULL || n->child(c) == NULL) {
85             return;
86         }
87         node_t *&tmp = n->child(c);

```

```

88     if (size(tmp->child(c)) > size(n->child(!c))) {
89         rotate(n, c);
90     } else if (size(tmp->child(!c)) > size(n->child(c))) {
91         rotate(tmp, !c);
92         rotate(n, c);
93     } else {
94         return;
95     }
96     maintain(n->left, 0);
97     maintain(n->right, 1);
98     maintain(n, 0);
99     maintain(n, 1);
100 }
101
102 static bool insert(node_t *&n, const K &k, const V &v) {
103     if (n == NULL) {
104         n = new node_t(k, v);
105         return true;
106     }
107     bool result;
108     if (k < n->key) {
109         result = insert(n->left, k, v);
110         maintain(n, 0);
111     } else if (n->key < k) {
112         result = insert(n->right, k, v);
113         maintain(n, 1);
114     } else {
115         return false;
116     }
117     n->update();
118     return result;
119 }
120
121 static bool erase(node_t *&n, const K &k) {
122     if (n == NULL) {
123         return false;
124     }
125     bool result;
126     int c = (k < n->key);
127     if (k < n->key) {
128         result = erase(n->left, k);
129     } else if (n->key < k) {
130         result = erase(n->right, k);
131     } else {
132         if (n->right == NULL || n->left == NULL) {
133             node_t *tmp = n;
134             n = (n->right == NULL) ? n->left : n->right;
135             delete tmp;
136             return true;
137         }
138         node_t *p = n->right;
139         while (p->left != NULL) {
140             p = p->left;
141         }
142         n->key = p->key;
143         result = erase(n->right, p->key);
144     }
145     maintain(n, c);
146     n->update();

```

```

147     return result;
148 }
149
150 static std::pair<K, V> select(node_t *n, int r) {
151     int rank = size(n->left);
152     if (r < rank) {
153         return select(n->left, r);
154     } else if (r > rank) {
155         return select(n->right, r - rank - 1);
156     }
157     return std::make_pair(n->key, n->value);
158 }
159
160 static int rank(node_t *n, const K &k) {
161     if (n == NULL) {
162         throw std::runtime_error("Cannot rank key that's not in tree.");
163     }
164     int r = size(n->left);
165     if (k < n->key) {
166         return rank(n->left, k);
167     } else if (n->key < k) {
168         return rank(n->right, k) + r + 1;
169     }
170     return r;
171 }
172
173 template<class KVFunction>
174 static void walk(node_t *n, KVFunction f) {
175     if (n != NULL) {
176         walk(n->left, f);
177         f(n->key, n->value);
178         walk(n->right, f);
179     }
180 }
181
182 static void clean_up(node_t *n) {
183     if (n != NULL) {
184         clean_up(n->left);
185         clean_up(n->right);
186         delete n;
187     }
188 }
189
190 public:
191     size_balanced_tree() : root(NULL) {}
192
193     ~size_balanced_tree() {
194         clean_up(root);
195     }
196
197     int size() const {
198         return size(root);
199     }
200
201     bool empty() const {
202         return root == NULL;
203     }
204
205     bool insert(const K &k, const V &v) {

```

```

206     return insert(root, k, v);
207 }
208
209 bool erase(const K &k) {
210     return erase(root, k);
211 }
212
213 const V* find(const K &k) const {
214     node_t *n = root;
215     while (n != NULL) {
216         if (k < n->key) {
217             n = n->left;
218         } else if (n->key < k) {
219             n = n->right;
220         } else {
221             return &(n->value);
222         }
223     }
224     return NULL;
225 }
226
227 std::pair<K, V> select(int r) const {
228     if (r < 0 || r >= size(root)) {
229         throw std::runtime_error("Select rank must be between 0 and size() - 1.");
230     }
231     return select(root, r);
232 }
233
234 int rank(const K &k) const {
235     return rank(root, k);
236 }
237
238 template<class KVFunction>
239 void walk(KVFunction f) const {
240     walk(root, f);
241 }
242 };
243
244 /**
245  abcde
246  bcde
247
248 */
249
250
251 #include <cassert>
252 #include <iostream>
253 using namespace std;
254
255 void printch(int k, char v) {
256     cout << v;
257 }
258
259 int main() {
260     size_balanced_tree<int, char> t;
261     t.insert(2, 'b');
262     t.insert(1, 'a');
263     t.insert(3, 'c');
264     t.insert(5, 'e');

```

```

265 assert(t.insert(4, 'd'));
266 assert(*t.find(4) == 'd');
267 assert(!t.insert(4, 'd'));
268 t.walk(printch);
269 cout << endl;
270 assert(t.erase(1));
271 assert(!t.erase(1));
272 assert(t.find(1) == NULL);
273 t.walk(printch);
274 cout << endl;
275 assert(t.rank(2) == 0);
276 assert(t.rank(3) == 1);
277 assert(t.rank(5) == 3);
278 assert(t.select(0).first == 2);
279 assert(t.select(1).first == 3);
280 assert(t.select(2).first == 4);
281 return 0;
282 }
```

2.2.7 Interval Treap

```

1 /*
2
3 Maintain a map from closed, one-dimensional intervals to values while supporting
4 efficient reporting of any or all entries that intersect with a given query
5 interval. This implementation uses std::pair to represent intervals, requiring
6 operators < and == to be defined on the numeric key type. A treap is used to
7 process the entries, where keys are compared lexicographically as pairs.
8
9 - interval_treap() constructs an empty map.
10 - size() returns the size of the map.
11 - empty() returns whether the map is empty.
12 - insert(lo, hi, v) adds an entry with key [lo, hi] and value v to the map,
13   returning true if a new interval was added or false if the interval already
14   exists (in which case the map is unchanged and the old value associated with
15   the key is preserved).
16 - erase(lo, hi) removes the entry with key [lo, hi] from the map, returning true
17   if the removal was successful or false if the interval was not found.
18 - find_key(lo, hi) returns a pointer to a const std::pair representing the key
19   of some interval in the map which intersects with [lo, hi], or NULL if no such
20   entry was found.
21 - find_value(lo, hi) returns a pointer to a const value of some entry in the map
22   with a key that intersects with [lo, hi], or NULL if no such entry was found.
23 - find_all(lo, hi, f) calls the function f(lo, hi, v) on each entry in the map
24   that overlaps with [lo, hi], in lexicographically ascending order of intervals.
25 - walk(f) calls the function f(lo, hi, v) on each interval in the map, in
26   lexicographically ascending order of intervals.
27
28 Time Complexity:
29 - O(1) per call to the constructor, size(), and empty().
30 - O(log n) on average per call to insert(), erase(), and find_any(), where n is
31   the number of intervals currently in the set.
32 - O(log n + m) on average per call to find_all(), where m is the number of
33   intersecting intervals that are reported.
34 - O(n) per call to walk().
```

```

35
36 Space Complexity:
37 - O(n) for storage of the map elements.
38 - O(1) auxiliary for size() and empty().
39 - O(log n) auxiliary stack space on average for all other operations.
40
41 */
42
43 #include <cstdlib>
44 #include <utility>
45
46 template<class K, class V>
47 class interval_treap {
48     typedef std::pair<K, K> interval_t;
49
50     struct node_t {
51         static inline int rand32() {
52             return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
53         }
54
55         interval_t interval;
56         V value;
57         K max;
58         int priority;
59         node_t *left, *right;
60
61         node_t(const interval_t &i, const V &v)
62             : interval(i), value(v), max(i.second), priority(rand32()), left(NULL),
63             right(NULL) {}
64
65         void update() {
66             max = interval.second;
67             if (left != NULL && left->max > max) {
68                 max = left->max;
69             }
70             if (right != NULL && right->max > max) {
71                 max = right->max;
72             }
73         }
74     } *root;
75
76     int num_nodes;
77
78     static void rotate_left(node_t *&n) {
79         node_t *tmp = n;
80         n = n->right;
81         tmp->right = n->left;
82         n->left = tmp;
83         tmp->update();
84     }
85
86     static void rotate_right(node_t *&n) {
87         node_t *tmp = n;
88         n = n->left;
89         tmp->left = n->right;
90         n->right = tmp;
91         tmp->update();
92     }
93

```

```

94     static bool insert(node_t *&n, const interval_t &i, const V &v) {
95         if (n == NULL) {
96             n = new node_t(i, v);
97             return true;
98         }
99         if (i < n->interval && insert(n->left, i, v)) {
100             if (n->left->priority < n->priority) {
101                 rotate_right(n);
102             }
103             n->update();
104             return true;
105         }
106         if (i > n->interval && insert(n->right, i, v)) {
107             if (n->right->priority < n->priority) {
108                 rotate_left(n);
109             }
110             n->update();
111             return true;
112         }
113         return false;
114     }
115
116     static bool erase(node_t *&n, const interval_t &i) {
117         if (n == NULL) {
118             return false;
119         }
120         if (i < n->interval) {
121             return erase(n->left, i);
122         }
123         if (i > n->interval) {
124             return erase(n->right, i);
125         }
126         if (n->left != NULL && n->right != NULL) {
127             bool res;
128             if (n->left->priority < n->right->priority) {
129                 rotate_right(n);
130                 res = erase(n->right, i);
131             } else {
132                 rotate_left(n);
133                 res = erase(n->left, i);
134             }
135             n->update();
136             return res;
137         }
138         node_t *tmp = (n->left != NULL) ? n->left : n->right;
139         delete n;
140         n = tmp;
141         return true;
142     }
143
144     static node_t* find_any(node_t *n, const interval_t &i) {
145         if (n == NULL) {
146             return NULL;
147         }
148         if (n->interval.first <= i.second && i.first <= n->interval.second) {
149             return n;
150         }
151         if (n->left != NULL && i.first <= n->left->max) {
152             return find_any(n->left, i);

```

```

153     }
154     return find_any(n->right, i);
155 }
156
157 template<class KVFunction>
158 static void find_all(node_t *n, const interval_t &i, KVFunction f) {
159     if (n == NULL || n->max < i.first) {
160         return;
161     }
162     if (n->interval.first <= i.second && i.first <= n->interval.second) {
163         f(n->interval.first, n->interval.second, n->value);
164     }
165     find_all(n->left, i, f);
166     find_all(n->right, i, f);
167 }
168
169 template<class KVFunction>
170 static void walk(node_t *n, KVFunction f) {
171     if (n != NULL) {
172         walk(n->left, f);
173         f(n->interval.first, n->interval.second, n->value);
174         walk(n->right, f);
175     }
176 }
177
178 static void clean_up(node_t *n) {
179     if (n != NULL) {
180         clean_up(n->left);
181         clean_up(n->right);
182         delete n;
183     }
184 }
185
186 public:
187     interval_treap() : root(NULL), num_nodes(0) {}
188
189     ~interval_treap() {
190         clean_up(root);
191     }
192
193     int size() const {
194         return num_nodes;
195     }
196
197     bool empty() const {
198         return root == NULL;
199     }
200
201     bool insert(const K &lo, const K &hi, const V &v) {
202         if (insert(root, std::make_pair(lo, hi), v)) {
203             num_nodes++;
204             return true;
205         }
206         return false;
207     }
208
209     bool erase(const K &lo, const K &hi) {
210         if (erase(root, std::make_pair(lo, hi))) {
211             num_nodes--;

```

```

212     return true;
213 }
214 return false;
215 }
216
217 const interval_t* find_key(const K &lo, const K &hi) const {
218     node_t *n = find_any(root, std::make_pair(lo, hi));
219     return (n == NULL) ? NULL : &(n->interval);
220 }
221
222 const V* find_value(const K &lo, const K &hi) const {
223     node_t *n = find_any(root, std::make_pair(lo, hi));
224     return (n == NULL) ? NULL : &(n->value);
225 }
226
227 template<class KVFunction>
228 void find_all(const K &lo, const K &hi, KVFunction f) const {
229     find_all(root, std::make_pair(lo, hi), f);
230 }
231
232 template<class KVFunction>
233 void walk(KVFunction f) const {
234     walk(root, f);
235 }
236 };
237
238 /** Example Usage and Output:
239
240 Intervals intersecting [16, 20]: [15, 20] [10, 30] [5, 20] [10, 40]
241 All intervals: [5, 20] [10, 30] [10, 40] [12, 15] [15, 20]
242
243 */
244
245 #include <cassert>
246 #include <iostream>
247 using namespace std;
248
249 void print(int lo, int hi, char v) {
250     cout << " [" << lo << ", " << hi << "]";
251 }
252
253 int main() {
254     interval_treap<int, char> t;
255     t.insert(15, 20, 'a');
256     t.insert(10, 30, 'b');
257     t.insert(17, 19, 'c');
258     t.insert(5, 20, 'd');
259     t.insert(12, 15, 'e');
260     t.insert(10, 40, 'f');
261     assert(t.size() == 6);
262     assert(!t.insert(5, 20, 'x'));
263     t.erase(17, 19);
264     assert(t.size() == 5);
265     assert(*t.find_key(3, 9) == make_pair(5, 20));
266     assert(*t.find_value(3, 9) == 'd');
267     cout << "Intervals intersecting [16, 20]:";
268     t.find_all(16, 20, print);
269     cout << "\nAll intervals:";
270     t.walk(print);

```

```

271     cout << endl;
272     return 0;
273 }
```

2.2.8 Hash Map

```

1  /*
2
3  Maintain a map, that is, a collection of key-value pairs such that each possible
4  key appears at most once in the collection. This implementation requires the ==
5  operator to be defined on the key type. A hash map implements a map by hashing
6  keys into buckets using a hash function. This implementation resolves collisions
7  by chaining entries hashed to the same bucket into a linked list.
8
9  - hash_map() constructs an empty map.
10 - size() returns the size of the map.
11 - empty() returns whether the map is empty.
12 - insert(k, v) adds an entry with key k and value v to the map, returning true
13   if a new entry was added or false if the key already exists (in which case
14   the map is unchanged and the old value associated with the key is preserved).
15 - erase(k) removes the entry with key k from the map, returning true if the
16   removal was successful or false if the key to be removed was not found.
17 - find(k) returns a pointer to a const value associated with key k, or NULL if
18   the key was not found.
19 - operator[k] returns a reference to key k's associated value (which may be
20   modified), or if necessary, inserts and returns a new entry with the default
21   constructed value if key k was not originally found.
22 - walk(f) calls the function f(k, v) on each entry of the map, in no guaranteed
23   order.
24
25 Time Complexity:
26 - O(1) per call to the constructor, size(), and empty().
27 - O(1) amortized per call to insert(), erase(), find(), and operator[].
28 - O(n) per call to walk(), where n is the number of entries in the map.
29
30 Space Complexity:
31 - O(n) for storage of the map elements.
32 - O(n) auxiliary heap space for insert().
33 - O(1) auxiliary for all other operations.
34 */
35
36
37 #include <cstddef>
38 #include <list>
39
40 template<class K, class V, class Hash>
41 class hash_map {
42     struct entry_t {
43         K key;
44         V value;
45
46         entry_t(const K &k, const V &v) : key(k), value(v) {}
47     };
48
49     std::list<entry_t> *table;
```

```

50     int table_size, num_entries;
51
52     void double_capacity_and_rehash() {
53         std::list<entry_t> *old = table;
54         int old_size = table_size;
55         table_size = 2*table_size;
56         table = new std::list<entry_t>[table_size];
57         num_entries = 0;
58         typename std::list<entry_t>::iterator it;
59         for (int i = 0; i < old_size; i++) {
60             for (it = old[i].begin(); it != old[i].end(); ++it) {
61                 insert(it->key, it->value);
62             }
63         }
64         delete[] old;
65     }
66
67 public:
68     hash_map(int size = 128) : table_size(size), num_entries(0) {
69         table = new std::list<entry_t>[table_size];
70     }
71
72     ~hash_map() {
73         delete[] table;
74     }
75
76     int size() const {
77         return num_entries;
78     }
79
80     bool empty() const {
81         return num_entries == 0;
82     }
83
84     bool insert(const K &k, const V &v) {
85         if (find(k) != NULL) {
86             return false;
87         }
88         if (num_entries >= table_size) {
89             double_capacity_and_rehash();
90         }
91         unsigned int i = Hash()(k) % table_size;
92         table[i].push_back(entry_t(k, v));
93         num_entries++;
94         return true;
95     }
96
97     bool erase(const K &k) {
98         unsigned int i = Hash()(k) % table_size;
99         typename std::list<entry_t>::iterator it = table[i].begin();
100        while (it != table[i].end() && !(it->key == k)) {
101            ++it;
102        }
103        if (it == table[i].end()) {
104            return false;
105        }
106        table[i].erase(it);
107        num_entries--;
108        return true;

```

```

109     }
110
111 V* find(const K &k) const {
112     unsigned int i = Hash()(k) % table_size;
113     typename std::list<entry_t>::iterator it = table[i].begin();
114     while (it != table[i].end() && !(it->key == k)) {
115         ++it;
116     }
117     if (it == table[i].end()) {
118         return NULL;
119     }
120     return &(it->value);
121 }
122
123 V& operator[](const K &k) {
124     V *ret = find(k);
125     if (ret != NULL) {
126         return *ret;
127     }
128     insert(k, V());
129     return *find(k);
130 }
131
132 template<class KVFunction>
133 void walk(KVFunction f) const {
134     for (int i = 0; i < table_size; i++) {
135         typename std::list<entry_t>::iterator it;
136         for (it = table[i].begin(); it != table[i].end(); ++it) {
137             f(it->key, it->value);
138         }
139     }
140 }
141 };
142
143 /** Example Usage and Output:
144
145 cab
146
147 */
148
149 #include <cassert>
150 #include <iostream>
151 using namespace std;
152
153 struct class_hash {
154     unsigned int operator()(int k) {
155         return class_hash()((unsigned int)k);
156     }
157
158     unsigned int operator()(long long k) {
159         return class_hash()((unsigned long long)k);
160     }
161
162     // Knuth's one-to-one multiplicative method.
163     unsigned int operator()(unsigned int k) {
164         return k * 2654435761u; // Or just return k.
165     }
166
167     // Jenkins's 64-bit hash.

```

```

168     unsigned int operator()(unsigned long long k) {
169         k += ~(k << 32);
170         k ^= (k >> 22);
171         k += ~(k << 13);
172         k ^= (k >> 8);
173         k += ~(k << 3);
174         k ^= (k >> 15);
175         k += ~(k << 27);
176         k ^= (k >> 31);
177         return k;
178     }
179
180     // Jenkins's one-at-a-time hash.
181     unsigned int operator()(const std::string &k) {
182         unsigned int hash = 0;
183         for (unsigned int i = 0; i < k.size(); i++) {
184             hash += ((hash + k[i]) << 10);
185             hash ^= (hash >> 6);
186         }
187         hash += (hash << 3);
188         hash ^= (hash >> 11);
189         return hash + (hash << 15);
190     }
191 };
192
193     void printch(const string &k, char v) {
194     cout << v;
195 }
196
197 int main() {
198     hash_map<string, char, class_hash> m;
199     m["foo"] = 'a';
200     m.insert("bar", 'b');
201     assert(m["foo"] == 'a');
202     assert(m["bar"] == 'b');
203     assert(m["baz"] == '\0');
204     m["baz"] = 'c';
205     m.walk(printch);
206     cout << endl;
207     assert(m.erase("foo"));
208     assert(m.size() == 2);
209     assert(m["foo"] == '\0');
210     assert(m.size() == 3);
211     return 0;
212 }
```

2.2.9 Skip List

```

1  /*
2
3  Maintain a map, that is, a collection of key-value pairs such that each possible
4  key appears at most once in the collection. This implementation requires both
5  the < and the == operators to be defined on the key type. A skip list maintains
6  a linked hierarchy of sorted subsequences with each successive subsequence
7  skipping over fewer elements than the previous one.
```

```

8
9 - skip_list() constructs an empty map.
10 - size() returns the size of the map.
11 - empty() returns whether the map is empty.
12 - insert(k, v) adds an entry with key k and value v to the map, returning true
13   if an new entry was added or false if the key already exists (in which case
14   the map is unchanged and the old value associated with the key is preserved).
15 - erase(k) removes the entry with key k from the map, returning true if the
16   removal was successful or false if the key to be removed was not found.
17 - find(k) returns a pointer to a const value associated with key k, or NULL if
18   the key was not found.
19 - operator[k] returns a reference to key k's associated value (which may be
20   modified), or if necessary, inserts and returns a new entry with the default
21   constructed value if key k was not originally found.
22 - walk(f) calls the function f(k, v) on each entry of the map, in ascending
23   order of keys.
24
25 Time Complexity:
26 - O(1) per call to the constructor, size(), and empty().
27 - O(log n) on average per call to insert(), erase(), find(), and operator[],
28   where n is the number of entries currently in the map.
29 - O(n) per call to walk().
30
31 Space Complexity:
32 - O(n) on average for storage of the map elements.
33 - O(n) auxiliary heap space for insert() and erase().
34 - O(1) auxiliary for all other operations.
35
36 */
37
38 #include <cmath>
39 #include <cstdlib>
40 #include <vector>
41
42 template<class K, class V>
43 class skip_list {
44     static const int MAX_LEVELS = 32; // log2(max possible keys)
45
46     struct node_t {
47         K key;
48         V value;
49         std::vector<node_t*> next;
50
51         node_t(const K &k, const V &v, int levels)
52             : key(k), value(v), next(levels, (node_t*)NULL) {}
53     } *head;
54
55     int num_nodes;
56
57     static int random_level() {
58         static const double p = 0.5;
59         int level = 1;
60         while (((double)rand() / RAND_MAX) < p && std::abs(level) < MAX_LEVELS) {
61             level++;
62         }
63         return std::abs(level);
64     }
65
66     static int node_level(const std::vector<node_t*> &v) {

```

```

67     int i = 0;
68     while (i < (int)v.size() && v[i] != NULL) {
69         i++;
70     }
71     return i + 1;
72 }
73
74 public:
75 skip_list() : head(new node_t(K(), V(), MAX_LEVELS)), num_nodes(0) {
76     for (int i = 0; i < (int)head->next.size(); i++) {
77         head->next[i] = NULL;
78     }
79 }
80
81 ~skip_list() {
82     delete head;
83 }
84
85 int size() const {
86     return num_nodes;
87 }
88
89 bool empty() const {
90     return num_nodes == 0;
91 }
92
93 bool insert(const K &k, const V &v) {
94     std::vector<node_t*> update(head->next);
95     int curr_level = node_level(update);
96     node_t *n = head;
97     for (int i = curr_level; i-- > 0; ) {
98         while (n->next[i] != NULL && n->next[i]->key < k) {
99             n = n->next[i];
100        }
101        update[i] = n;
102    }
103    n = n->next[0];
104    if (n != NULL && n->key == k) {
105        return false;
106    }
107    int new_level = random_level();
108    if (new_level > curr_level) {
109        for (int i = curr_level; i < new_level; i++) {
110            update[i] = head;
111        }
112    }
113    n = new node_t(k, v, new_level);
114    for (int i = 0; i < new_level; i++) {
115        n->next[i] = update[i]->next[i];
116        update[i]->next[i] = n;
117    }
118    num_nodes++;
119    return true;
120 }
121
122 bool erase(const K &k) {
123     std::vector<node_t*> update(head->next);
124     node_t *n = head;
125     for (int i = node_level(update); i-- > 0; ) {

```

```

126     while (n->next[i] != NULL && n->next[i]->key < k) {
127         n = n->next[i];
128     }
129     update[i] = n;
130 }
131 n = n->next[0];
132 if (n != NULL && n->key == k) {
133     for (int i = 0; i < (int)update.size(); i++) {
134         if (update[i]->next[i] != n) {
135             break;
136         }
137         update[i]->next[i] = n->next[i];
138     }
139     delete n;
140     num_nodes--;
141     return true;
142 }
143 return false;
144 }
145
146 V* find(const K &k) const {
147     node_t *n = head;
148     for (int i = node_level(n->next); i-- > 0; ) {
149         while (n->next[i] != NULL && n->next[i]->key < k) {
150             n = n->next[i];
151         }
152     }
153     n = n->next[0];
154     return (n != NULL && n->key == k) ? &(n->value) : NULL;
155 }
156
157 V& operator[](const K &k) {
158     V *ret = find(k);
159     if (ret != NULL) {
160         return *ret;
161     }
162     insert(k, V());
163     return *find(k);
164 }
165
166 template<class KVFunction>
167 void walk(KVFunction f) const {
168     node_t *n = head->next[0];
169     while (n != NULL) {
170         f(n->key, n->value);
171         n = n->next[0];
172     }
173 }
174 };
175
176 /**
177 abcde
178 bcde
179 */
180
181 /**
182
183 #include <cassert>
184 #include <iostream>

```

```

185 using namespace std;
186
187 void printch(int k, char v) {
188     cout << v;
189 }
190
191 int main() {
192     skip_list<int, char> l;
193     l.insert(2, 'b');
194     l.insert(1, 'a');
195     l.insert(3, 'c');
196     l.insert(5, 'e');
197     assert(l.insert(4, 'd'));
198     assert(*l.find(4) == 'd');
199     assert(!l.insert(4, 'd'));
200     l.walk(printch);
201     cout << endl;
202     assert(l.erase(1));
203     assert(!l.erase(1));
204     assert(l.find(1) == NULL);
205     l.walk(printch);
206     cout << endl;
207     return 0;
208 }
```

2.3 Range Queries in One Dimension

2.3.1 Sparse Table (Range Minimum Query)

```

1 /*
2
3 Given a static array with indices from 0 to n - 1, precompute a table that may
4 later be used perform range minimum queries on the array in constant time. This
5 version is simplified to only work on integer arrays.
6
7 The dynamic programming state dp[i][j] holds the index of the minimum value in
8 the sub-array starting at i and having length 2^j. Each dp[i][j] will always be
9 equal to either dp[i][j - 1] or dp[i + 2^(j - 1) - 1][j - 1], whichever of the
10 indices corresponds to the smaller value in the array.
11
12 Time Complexity:
13 - O(n log n) per call to build(), where n is the size of the array.
14 - O(1) per call to query_min().
15
16 Space Complexity:
17 - O(n log n) for storage of the sparse table, where n is the size of the array.
18 - O(1) auxiliary for query().
19
20 */
21
22 #include <vector>
23
24 const int MAXN = 1000;
```

```

25 std::vector<int> table, dp[MAXN];
26
27 void build(int n, int a[]) {
28     table.resize(n + 1);
29     for (int i = 2; i <= n; i++) {
30         table[i] = table[i >> 1] + 1;
31     }
32     for (int i = 0; i < n; i++) {
33         dp[i].resize(table[n] + 1);
34         dp[i][0] = i;
35     }
36     for (int j = 1; (1 << j) < n; j++) {
37         for (int i = 0; i + (1 << j) <= n; i++) {
38             int x = dp[i][j - 1];
39             int y = dp[i + (1 << (j - 1))][j - 1];
40             dp[i][j] = (a[x] < a[y]) ? x : y;
41         }
42     }
43 }
44
45 int query(int a[], int lo, int hi) {
46     int j = table[hi - lo];
47     int x = dp[lo][j];
48     int y = dp[hi - (1 << j) + 1][j];
49     return (a[x] < a[y]) ? a[x] : a[y];
50 }
51
52 /** Example Usage ***/
53
54 #include <cassert>
55
56 int main() {
57     int arr[5] = {6, -2, 1, 8, 10};
58     build(5, arr);
59     assert(query(arr, 0, 3) == -2);
60     return 0;
61 }
```

2.3.2 Square Root Decomposition

```

1 /*
2
3 Maintain a fixed-size array (from 0 to size() - 1) while supporting dynamic
4 queries of contiguous sub-arrays and dynamic updates of individual indices.
5
6 The query operation is defined by an associative join_values() function which
7 satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)
8 for all values x, y, and z in the array. The default definition below assumes a
9 numerical array type, supporting queries for the "min" of the target range.
10 Another possible query operation is "sum", in which the join_values() function
11 should be defined to return "a + b".
12
13 The update operation is defined by the join_value_with_delta() function which
14 determines the change made to array values. The default definition below
15 supports updates that "set" the chosen array index to a new value. Another
```

```
16 possible update operation is "increment", in which join_value_with_delta(v, d)
17 should be defined to return "v + d".
18
19 The operations supported by this data structure are identical to those of the
20 point update segment tree found in this section.
21
22 Time Complexity:
23 - O(n) per call to both constructors, where n is the size of the array.
24 - O(1) per call to size().
25 - O(sqrt n) per call to at(), update(), and query().
26
27 Space Complexity:
28 - O(n) for storage of the array elements.
29 - O(1) auxiliary for all operations.
30
31 */
32
33 #include <algorithm>
34 #include <cmath>
35 #include <vector>
36
37 template<class T>
38 class sqrt_decomposition {
39     static T join_values(const T &a, const T &b) {
40         return std::min(a, b);
41     }
42
43     static T join_value_with_delta(const T &v, const T &d) {
44         return v + d;
45     }
46
47     int len, blocklen;
48     std::vector<T> value, block;
49
50     void init() {
51         blocklen = (int)sqrt(len);
52         int nbblocks = (len + blocklen - 1)/blocklen;
53         for (int i = 0; i < nbblocks; i++) {
54             T blockval = value[i*blocklen];
55             int blockhi = std::min(len, (i + 1)*blocklen);
56             for (int j = i*blocklen + 1; j < blockhi; j++) {
57                 blockval = join_values(blockval, value[j]);
58             }
59             block.push_back(blockval);
60         }
61     }
62
63 public:
64     sqrt_decomposition(int n, const T &v = T()) : len(n), value(n, v) {
65         init();
66     }
67
68     template<class It>
69     sqrt_decomposition(It lo, It hi) : len(hi - lo), value(lo, hi) {
70         init();
71     }
72
73     int size() const {
74         return len;
```

```

75     }
76
77     T at(int i) const {
78         return query(i, i);
79     }
80
81     T query(int lo, int hi) const {
82         T res;
83         int blocklo = ceil((double)lo/blocklen), blockhi = (hi + 1)/blocklen - 1;
84         if (blocklo > blockhi) {
85             res = value[lo];
86             for (int i = lo + 1; i <= hi; i++) {
87                 res = join_values(res, value[i]);
88             }
89         } else {
90             res = block[blocklo];
91             for (int i = blocklo + 1; i <= blockhi; i++) {
92                 res = join_values(res, block[i]);
93             }
94             for (int i = lo; i < blocklo*blocklen; i++) {
95                 res = join_values(res, value[i]);
96             }
97             for (int i = (blockhi + 1)*blocklen; i <= hi; i++) {
98                 res = join_values(res, value[i]);
99             }
100        }
101    }
102
103
104    void update(int i, const T &d) {
105        value[i] = join_value_with_delta(value[i], d);
106        int b = i/blocklen;
107        int blockhi = std::min(len, (b + 1)*blocklen);
108        block[b] = value[b*blocklen];
109        for (int i = b*blocklen + 1; i < blockhi; i++) {
110            block[b] = join_values(block[b], value[i]);
111        }
112    }
113 };
114
115 /**
116  * Example Usage and Output:
117  * Values: 6 -2 4 8 10
118  */
119
120
121 #include <cassert>
122 #include <iostream>
123 using namespace std;
124
125 int main() {
126     int arr[5] = {6, -2, 1, 8, 10};
127     sqrt_decomposition<int> sd(arr, arr + 5);
128     sd.update(2, 4);
129     cout << "Values:";
130     for (int i = 0; i < sd.size(); i++) {
131         cout << " " << sd.at(i);
132     }
133     cout << endl;

```

```

134     assert(sd.query(0, 3) == -2);
135     return 0;
136 }
```

2.3.3 Segment Tree (Point Update)

```

1  /*
2
3  Maintain a fixed-size array while supporting dynamic queries of contiguous
4  sub-arrays and dynamic updates of individual indices.
5
6  The query operation is defined by an associative join_values() function which
7  satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)
8  for all values x, y, and z in the array. The default code below assumes a
9  numerical array type, defining queries for the "min" of the target range.
10 Another possible query operation is "sum", in which case the join_values()
11 function should be defined to return "a + b".
12
13 The update operation is defined by the join_value_with_delta() function, which
14 determines the change made to array values. The default definition below
15 supports updates that "set" the chosen array index to a new value. Another
16 possible update operation is "increment", in which join_value_with_delta(v, d)
17 should be defined to return "v + d".
18
19 - segment_tree(n, v) constructs an array of size n with indices from 0 to n - 1,
20   inclusive, and all values initialized to v.
21 - segment_tree(lo, hi) constructs an array from two random-access iterators as a
22   range [lo, hi), initialized to the elements of the range in the same order.
23 - size() returns the size of the array.
24 - at(i) returns the value at index i.
25 - query(lo, hi) returns the result of join_values() applied to all indices from
26   lo to hi, inclusive. If the distance between lo and hi is 1, then the single
27   specified value is returned.
28 - update(i, d) assigns the value v at index i to join_value_with_delta(v, d).
29
30 Time Complexity:
31 - O(n) per call to both constructors, where n is the size of the array.
32 - O(1) per call to size().
33 - O(log n) per call to at(), update(), and query().
34
35 Space Complexity:
36 - O(n) for storage of the array elements.
37 - O(log n) auxiliary stack space for update() and query().
38 - O(1) auxiliary for size().
39
40 */
41
42 #include <algorithm>
43 #include <vector>
44
45 template<class T>
46 class segment_tree {
47     static T join_values(const T &a, const T &b) {
48         return std::min(a, b);
49     }
```

```

50
51     static T join_value_with_delta(const T &v, const T &d) {
52         return d;
53     }
54
55     int len;
56     std::vector<T> value;
57
58     void build(int i, int lo, int hi, const T &v) {
59         if (lo == hi) {
60             value[i] = v;
61             return;
62         }
63         int mid = lo + (hi - lo)/2;
64         build(i*2 + 1, lo, mid, v);
65         build(i*2 + 2, mid + 1, hi, v);
66         value[i] = join_values(value[i*2 + 1], value[i*2 + 2]);
67     }
68
69     template<class It>
70     void build(int i, int lo, int hi, It arr) {
71         if (lo == hi) {
72             value[i] = *(arr + lo);
73             return;
74         }
75         int mid = lo + (hi - lo)/2;
76         build(i*2 + 1, lo, mid, arr);
77         build(i*2 + 2, mid + 1, hi, arr);
78         value[i] = join_values(value[i*2 + 1], value[i*2 + 2]);
79     }
80
81     T query(int i, int lo, int hi, int tgt_lo, int tgt_hi) const {
82         if (lo == tgt_lo && hi == tgt_hi) {
83             return value[i];
84         }
85         int mid = lo + (hi - lo)/2;
86         if (tgt_lo <= mid && mid < tgt_hi) {
87             return join_values(
88                 query(i*2 + 1, lo, mid, tgt_lo, std::min(tgt_hi, mid)),
89                 query(i*2 + 2, mid + 1, hi, std::max(tgt_lo, mid + 1), tgt_hi));
90         }
91         if (tgt_lo <= mid) {
92             return query(i*2 + 1, lo, mid, tgt_lo, std::min(tgt_hi, mid));
93         }
94         return query(i*2 + 2, mid + 1, hi, std::max(tgt_lo, mid + 1), tgt_hi);
95     }
96
97     void update(int i, int lo, int hi, int target, const T &d) {
98         if (target < lo || target > hi) {
99             return;
100        }
101        if (lo == hi) {
102            value[i] = join_value_with_delta(value[i], d);
103            return;
104        }
105        int mid = lo + (hi - lo)/2;
106        update(i*2 + 1, lo, mid, target, d);
107        update(i*2 + 2, mid + 1, hi, target, d);
108        value[i] = join_values(value[i*2 + 1], value[i*2 + 2]);

```

```

109     }
110
111     public:
112     segment_tree(int n, const T &v = T()) : len(n), value(4*len) {
113         build(0, 0, len - 1, false, 0, v);
114     }
115
116     template<class It>
117     segment_tree(It lo, It hi) : len(hi - lo), value(4*len) {
118         build(0, 0, len - 1, lo);
119     }
120
121     int size() const {
122         return len;
123     }
124
125     T at(int i) const {
126         return query(i, i);
127     }
128
129     T query(int lo, int hi) const {
130         return query(0, 0, len - 1, lo, hi);
131     }
132
133     void update(int i, const T &d) {
134         update(0, 0, len - 1, i, d);
135     }
136 };
137
138 /** Example Usage and Output:
139
140 Values: 6 -2 4 8 10
141 The minimum value in the range [0, 3] is -2.
142
143 */
144
145 #include <cassert>
146 #include <iostream>
147 using namespace std;
148
149 int main() {
150     int arr[5] = {6, -2, 1, 8, 10};
151     segment_tree<int> t(arr, arr + 5);
152     t.update(2, 4);
153     cout << "Values:";
154     for (int i = 0; i < t.size(); i++) {
155         cout << " " << t.at(i);
156     }
157     cout << endl;
158     assert(t.query(0, 3) == -2);
159     return 0;
160 }
```

2.3.4 Segment Tree (Range Update)

```

1  /*
2
3  Maintain a fixed-size array while supporting both dynamic queries and updates of
4  contiguous subarrays via the lazy propagation technique.
5
6  The query operation is defined by an associative join_values() function which
7  satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)
8  for all values x, y, and z in the array. The default code below assumes a
9  numerical array type, defining queries for the "min" of the target range.
10 Another possible query operation is "sum", in which case the join_values()
11 function should be defined to return "a + b".
12
13 The update operation is defined by the join_value_with_delta() and join_deltas()
14 functions, which determines the change made to array values. These must satisfy:
15 - join_deltas(d1, join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).
16 - join_value_with_delta(join_values(v, ... (m times) ..., v), d, m) should be
17   equal to join_values(join_value_with_delta(v, d, 1), ... (m times)).
18 - if a sequence d_1, ..., d_m of deltas is used to update a value v, then
19   join_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent
20   to m sequential calls to join_value_with_delta(v, d_i, 1) for i = 1...m.
21 The default code below defines updates that "set" the chosen array index to a
22 new value. Another possible update operation is "increment", in which case
23 join_value_with_delta(v, d, len) should be defined to return "v + d*len" and
24 join_deltas(d1, d2) should be defined to return "d1 + d2".
25
26 - segment_tree(n, v) constructs an array of size n with indices from 0 to n - 1,
27   inclusive, and all values initialized to v.
28 - segment_tree(lo, hi) constructs an array from two random-access iterators as a
29   range [lo, hi), initialized to the elements of the range in the same order.
30 - size() returns the size of the array.
31 - at(i) returns the value at index i, where i is between 0 and size() - 1.
32 - query(lo, hi) returns the result of join_values() applied to all indices from
33   lo to hi, inclusive. If the distance between lo and hi is 1, then the single
34   specified value is returned.
35 - update(i, d) assigns the value v at index i to join_value_with_delta(v, d).
36 - update(lo, hi, d) modifies the value at each array index from lo to hi,
37   inclusive, by respectively joining them with d using join_value_with_delta().
38
39 Time Complexity:
40 - O(n) per call to both constructors, where n is the size of the array.
41 - O(1) per call to size().
42 - O(log n) per call to at(), update(), and query().
43
44 Space Complexity:
45 - O(n) for storage of the array elements.
46 - O(log n) auxiliary stack space for update() and query().
47 - O(1) auxiliary for size().
48 */
49
50
51 #include <algorithm>
52 #include <vector>
53
54 template<class T>
55 class segment_tree {
56     static T join_values(const T &a, const T &b) {
57         return std::min(a, b);
58     }
59

```

```

60     static T join_value_with_delta(const T &v, const T &d, int len) {
61         return d;
62     }
63
64     static T join_deltas(const T &d1, const T &d2) {
65         return d2; // For "set" updates, the more recent delta prevails.
66     }
67
68     int len;
69     std::vector<T> value, delta;
70     std::vector<bool> pending;
71
72     void build(int i, int lo, int hi, const T &v) {
73         if (lo == hi) {
74             value[i] = v;
75             return;
76         }
77         int mid = lo + (hi - lo)/2;
78         build(i*2 + 1, lo, mid, v);
79         build(i*2 + 2, mid + 1, hi, v);
80         value[i] = join_values(value[i*2 + 1], value[i*2 + 2]);
81     }
82
83     template<class It>
84     void build(int i, int lo, int hi, It arr) {
85         if (lo == hi) {
86             value[i] = *(arr + lo);
87             return;
88         }
89         int mid = lo + (hi - lo)/2;
90         build(i*2 + 1, lo, mid, arr);
91         build(i*2 + 2, mid + 1, hi, arr);
92         value[i] = join_values(value[i*2 + 1], value[i*2 + 2]);
93     }
94
95     void push_delta(int i, int lo, int hi) {
96         if (pending[i]) {
97             value[i] = join_value_with_delta(value[i], delta[i], hi - lo + 1);
98             if (lo != hi) {
99                 int l = 2*i + 1, r = 2*i + 2;
100                delta[l] = pending[l] ? join_deltas(delta[l], delta[i]) : delta[i];
101                delta[r] = pending[r] ? join_deltas(delta[r], delta[i]) : delta[i];
102                pending[l] = pending[r] = true;
103            }
104            pending[i] = false;
105        }
106    }
107
108    T query(int i, int lo, int hi, int tgt_lo, int tgt_hi) {
109        push_delta(i, lo, hi);
110        if (lo == tgt_lo && hi == tgt_hi) {
111            return value[i];
112        }
113        int mid = lo + (hi - lo)/2;
114        if (tgt_lo <= mid && mid < tgt_hi) {
115            return join_values(
116                query(i*2 + 1, lo, mid, tgt_lo, std::min(tgt_hi, mid)),
117                query(i*2 + 2, mid + 1, hi, std::max(tgt_lo, mid + 1), tgt_hi));
118        }

```

```

119     if (tgt_lo <= mid) {
120         return query(i*2 + 1, lo, mid, tgt_lo, std::min(tgt_hi, mid));
121     }
122     return query(i*2 + 2, mid + 1, hi, std::max(tgt_lo, mid + 1), tgt_hi);
123 }
124
125 void update(int i, int lo, int hi, int tgt_lo, int tgt_hi, const T &d) {
126     push_delta(i, lo, hi);
127     if (hi < tgt_lo || lo > tgt_hi) {
128         return;
129     }
130     if (tgt_lo <= lo && hi <= tgt_hi) {
131         delta[i] = d;
132         pending[i] = true;
133         push_delta(i, lo, hi);
134         return;
135     }
136     update(2*i + 1, lo, (lo + hi)/2, tgt_lo, tgt_hi, d);
137     update(2*i + 2, (lo + hi)/2 + 1, hi, tgt_lo, tgt_hi, d);
138     value[i] = join_values(value[2*i + 1], value[2*i + 2]);
139 }
140
141 public:
142     segment_tree(int n, const T &v = T())
143         : len(n), value(4*len), delta(4*len), pending(4*len, false) {
144         build(0, 0, len - 1, v);
145     }
146
147     template<class It>
148     segment_tree(It lo, It hi)
149         : len(hi - lo), value(4*len), delta(4*len), pending(4*len, false) {
150         build(0, 0, len - 1, lo);
151     }
152
153     int size() const {
154         return len;
155     }
156
157     T at(int i) {
158         return query(i, i);
159     }
160
161     T query(int lo, int hi) {
162         return query(0, 0, len - 1, lo, hi);
163     }
164
165     void update(int i, const T &d) {
166         update(0, 0, len - 1, i, i, d);
167     }
168
169     void update(int lo, int hi, const T &d) {
170         update(0, 0, len - 1, lo, hi, d);
171     }
172 };
173
174 /**
175  * Example Usage and Output:
176  * Values: 6 -2 4 8 10
177  * Values: 5 5 5 1 5

```

```

178
179  */
180
181 #include <cassert>
182 #include <iostream>
183 using namespace std;
184
185 int main() {
186     int arr[5] = {6, -2, 1, 8, 10};
187     segment_tree<int> t(arr, arr + 5);
188     t.update(2, 4);
189     cout << "Values:";
190     for (int i = 0; i < t.size(); i++) {
191         cout << " " << t.at(i);
192     }
193     cout << endl;
194     assert(t.query(0, 3) == -2);
195     t.update(0, 4, 5);
196     t.update(3, 2);
197     t.update(3, 1);
198     cout << "Values:";
199     for (int i = 0; i < t.size(); i++) {
200         cout << " " << t.at(i);
201     }
202     cout << endl;
203     assert(t.query(0, 3) == 1);
204     return 0;
205 }
```

2.3.5 Segment Tree (Compressed)

```

1 /*
2
3 Maintain a fixed-size array while supporting both dynamic queries and updates of
4 contiguous subarrays via the lazy propagation technique. This implementation
5 uses lazy initialization of nodes to conserve memory while supporting large
6 indices.
7
8 The query operation is defined by an associative join_values() function which
9 satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)
10 for all values x, y, and z in the array. The default code below assumes a
11 numerical array type, defining queries for the "min" of the target range.
12 Another possible query operation is "sum", in which case the join_values()
13 function should be defined to return "a + b".
14
15 The update operation is defined by the join_value_with_delta() and join_deltas()
16 functions, which determines the change made to array values. These must satisfy:
17 - join_deltas(d1, join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).
18 - join_value_with_delta(join_values(v, ...(m times)..., v), d, m) should be
19   equal to join_values(join_value_with_delta(v, d, 1), ...(m times)).
20 - if a sequence d_1, ..., d_m of deltas is used to update a value v, then
21   join_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent
22   to m sequential calls to join_value_with_delta(v, d_i, 1) for i = 1..m.
23 The default code below defines updates that "set" the chosen array index to a
24 new value. Another possible update operation is "increment", in which case
```

```

25 join_value_with_delta(v, d, len) should be defined to return "v + d*len" and
26 join_deltas(d1, d2) should be defined to return "d1 + d2".
27
28 - segment_tree(n, v) constructs an array of size n with indices from 0 to n - 1,
29   inclusive, and all values initialized to v.
30 - segment_tree(lo, hi) constructs an array from two random-access iterators as a
31   range [lo, hi), initialized to the elements of the range in the same order.
32 - size() returns the size of the array.
33 - at(i) returns the value at index i, where i is between 0 and size() - 1.
34 - query(lo, hi) returns the result of join_values() applied to all indices from
35   lo to hi, inclusive. If the distance between lo and hi is 1, then the single
36   specified value is returned.
37 - update(i, d) assigns the value v at index i to join_value_with_delta(v, d).
38 - update(lo, hi, d) modifies the value at each array index from lo to hi,
39   inclusive, by respectively joining them with d using join_value_with_delta().
40
41 Time Complexity:
42 - O(n) per call to both constructors, where n is the size of the array.
43 - O(1) per call to size().
44 - O(log n) per call to at(), update(), and query().
45
46 Space Complexity:
47 - O(n) for storage of the array elements.
48 - O(log n) auxiliary stack space for update() and query().
49 - O(1) auxiliary for size().
50
51 */
52
53 #include <algorithm>
54 #include <cstddef>
55
56 template<class T>
57 class segment_tree {
58     static const int MAXN = 1000000000;
59
60     static T join_values(const T &a, const T &b) {
61         return std::min(a, b);
62     }
63
64     static T join_segment(const T &v, int len) {
65         return v;
66     }
67
68     static T join_value_with_delta(const T &v, const T &d, int len) {
69         return d;
70     }
71
72     static T join_deltas(const T &d1, const T &d2) {
73         return d2; // For "set" updates, the more recent delta prevails.
74     }
75
76     struct node_t {
77         T value, delta;
78         bool pending;
79         node_t *left, *right;
80
81         node_t(const T &v) : value(v), pending(false), left(NULL), right(NULL) {}
82     } *root;
83

```

```

84     T init;
85
86     void update_delta(node_t *&n, const T &d, int len) {
87         if (n == NULL) {
88             n = new node_t(join_segment(init, len));
89         }
90         n->delta = n->pending ? join_deltas(n->delta, d) : d;
91         n->pending = true;
92     }
93
94     void push_delta(node_t *n, int lo, int hi) {
95         if (n->pending) {
96             n->value = join_value_with_delta(n->value, n->delta, hi - lo + 1);
97             if (lo != hi) {
98                 int mid = lo + (hi - lo)/2;
99                 update_delta(n->left, n->delta, mid - lo + 1);
100                update_delta(n->right, n->delta, hi - mid);
101            }
102        }
103        n->pending = false;
104    }
105
106    T query(node_t *n, int lo, int hi, int tgt_lo, int tgt_hi) {
107        push_delta(n, lo, hi);
108        if (lo == tgt_lo && hi == tgt_hi) {
109            return (n == NULL) ? join_segment(init, hi - lo + 1) : n->value;
110        }
111        int mid = lo + (hi - lo)/2;
112        if (tgt_lo <= mid && mid < tgt_hi) {
113            return join_values(
114                query(n->left, lo, mid, tgt_lo, std::min(tgt_hi, mid)),
115                query(n->right, mid + 1, hi, std::max(tgt_lo, mid + 1), tgt_hi));
116        }
117        if (tgt_lo <= mid) {
118            return query(n->left, lo, mid, tgt_lo, std::min(tgt_hi, mid));
119        }
120        return query(n->right, mid + 1, hi, std::max(tgt_lo, mid + 1), tgt_hi);
121    }
122
123    void update(node_t *&n, int lo, int hi, int tgt_lo, int tgt_hi, const T &d) {
124        if (n == NULL) {
125            n = new node_t(join_segment(init, hi - lo + 1));
126        }
127        push_delta(n, lo, hi);
128        if (hi < tgt_lo || lo > tgt_hi) {
129            return;
130        }
131        if (tgt_lo <= lo && hi <= tgt_hi) {
132            n->delta = d;
133            n->pending = true;
134            push_delta(n, lo, hi);
135            return;
136        }
137        int mid = lo + (hi - lo)/2;
138        update(n->left, lo, mid, tgt_lo, tgt_hi, d);
139        update(n->right, mid + 1, hi, tgt_lo, tgt_hi, d);
140        n->value = join_values(n->left->value, n->right->value);
141    }
142

```

```

143     void clean_up(node_t *n) {
144         if (n != NULL) {
145             clean_up(n->left);
146             clean_up(n->right);
147             delete n;
148         }
149     }
150
151     public:
152     segment_tree(const T &v = T()) : root(NULL), init(v) {}
153
154     ~segment_tree() {
155         clean_up(root);
156     }
157
158     T at(int i) {
159         return query(i, i);
160     }
161
162     T query(int lo, int hi) {
163         return query(root, 0, MAXN, lo, hi);
164     }
165
166     void update(int i, const T &d) {
167         return update(i, i, d);
168     }
169
170     void update(int lo, int hi, const T &d) {
171         return update(root, 0, MAXN, lo, hi, d);
172     }
173 };
174
175 /** Example Usage and Output:
176
177 Values: 6 -2 4 8 10
178 Values: 5 5 5 1 5
179
180 */
181
182 #include <cassert>
183 #include <iostream>
184 using namespace std;
185
186 int main() {
187     segment_tree<int> t(0);
188     t.update(0, 6);
189     t.update(1, -2);
190     t.update(2, 4);
191     t.update(3, 8);
192     t.update(4, 10);
193     cout << "Values:";
194     for (int i = 0; i < 5; i++) {
195         cout << " " << t.at(i);
196     }
197     cout << endl;
198     assert(t.query(0, 3) == -2);
199     t.update(0, 4, 5);
200     t.update(3, 2);
201     t.update(3, 1);

```

```

202     cout << "Values:";  

203     for (int i = 0; i < 5; i++) {  

204         cout << " " << t.at(i);  

205     }  

206     cout << endl;  

207     assert(t.query(0, 3) == 1);  

208     return 0;  

209 }
```

2.3.6 Implicit Treap

```

1  /*  

2  

3  Maintain a dynamically-sized array using a balanced binary search tree while  

4  supporting both dynamic queries and updates of contiguous subarrays via the lazy  

5  propagation technique. A treap maintains a balanced binary tree structure by  

6  preserving the heap property on the randomly generated priority values of nodes,  

7  thereby making insertions and deletions run in O(log n) with high probability.  

8  

9  The query operation is defined by an associative join_values() function which  

10 satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)  

11 for all values x, y, and z in the array. The default code below assumes a  

12 numerical array type, defining queries for the "min" of the target range.  

13 Another possible query operation is "sum", in which case the join_values()  

14 function should be defined to return "a + b".  

15  

16 The update operation is defined by the join_value_with_delta() and join_deltas()  

17 functions, which determines the change made to array values. These must satisfy:  

18 - join_deltas(d1, join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).  

19 - join_value_with_delta(join_values(v, ... (m times) ..., v), d, m) should be  

20   equal to join_values(join_value_with_delta(v, d, 1), ... (m times)).  

21 - if a sequence d_1, ..., d_m of deltas is used to update a value v, then  

22   join_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent  

23   to m sequential calls to join_value_with_delta(v, d_i, 1) for i = 1..m.  

24 The default code below defines updates that "set" the chosen array index to a  

25 new value. Another possible update operation is "increment", in which case  

26 join_value_with_delta(v, d, len) should be defined to return "v + d*len" and  

27 join_deltas(d1, d2) should be defined to return "d1 + d2".  

28  

29 This data structure shares every operation of one-dimensional segment trees in  

30 this section, with the additional operations empty(), insert(), erase(),  

31 push_back(), and pop_back() analogous to those of std::vector (here, insert()  

32 and erase() both take an index instead of an iterator).  

33  

34 Time Complexity:  

35 - O(n) per call to both constructors, where n is the size of the array.  

36 - O(1) per call to size() and empty().  

37 - O(log n) on average per call to all other operations.  

38  

39 Space Complexity:  

40 - O(n) for storage of the array elements.  

41 - O(1) auxiliary for size() and empty().  

42 - O(log n) auxiliary stack space for all other operations.  

43  

44 */
```

```

45
46 #include <cstdlib>
47
48 template<class T>
49 class implicit_treap {
50     static T join_values(const T &a, const T &b) {
51         return a < b ? a : b;
52     }
53
54     static T join_value_with_delta(const T &v, const T &d, int len) {
55         return d;
56     }
57
58     static T join_deltas(const T &d1, const T &d2) {
59         return d2;
60     }
61
62     struct node_t {
63         static inline int rand32() {
64             return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
65         }
66
67         T value, subtree_value, delta;
68         bool pending;
69         int size, priority;
70         node_t *left, *right;
71
72         node_t(const T &v)
73             : value(v), subtree_value(v), pending(false), size(1),
74               priority(rand32()), left(NULL), right(NULL) {}
75     } *root;
76
77     static int size(node_t *n) {
78         return (n == NULL) ? 0 : n->size;
79     }
80
81     static void update_value(node_t *n) {
82         if (n == NULL) {
83             return;
84         }
85         n->subtree_value = n->value;
86         if (n->left != NULL) {
87             n->subtree_value = join_values(n->subtree_value, n->left->subtree_value);
88         }
89         if (n->right != NULL) {
90             n->subtree_value = join_values(n->subtree_value, n->right->subtree_value);
91         }
92         n->size = 1 + size(n->left) + size(n->right);
93     }
94
95     static void update_delta(node_t *n, const T &d) {
96         if (n != NULL) {
97             n->delta = n->pending ? join_deltas(n->delta, d) : d;
98             n->pending = true;
99         }
100    }
101
102    static void push_delta(node_t *n) {
103        if (n == NULL || !n->pending) {

```

```

104     return;
105 }
106 n->value = join_value_with_delta(n->value, n->delta, 1);
107 n->subtree_value = join_value_with_delta(n->subtree_value, n->delta,
108                                         n->size);
109 if (n->size > 1) {
110     update_delta(n->left, n->delta);
111     update_delta(n->right, n->delta);
112 }
113 n->pending = false;
114 }
115
116 static void merge(node_t *&n, node_t *left, node_t *right) {
117     push_delta(left);
118     push_delta(right);
119     if (left == NULL) {
120         n = right;
121     } else if (right == NULL) {
122         n = left;
123     } else if (left->priority < right->priority) {
124         merge(left->right, left->right, right);
125         n = left;
126     } else {
127         merge(right->left, left, right->left);
128         n = right;
129     }
130     update_value(n);
131 }
132
133 static void split(node_t *&n, node_t *&left, node_t *&right, int i) {
134     push_delta(n);
135     if (n == NULL) {
136         left = right = NULL;
137     } else if (i <= size(n->left)) {
138         split(n->left, left, n->left, i);
139         right = n;
140     } else {
141         split(n->right, n->right, right, i - size(n->left) - 1);
142         left = n;
143     }
144     update_value(n);
145 }
146
147 static void insert(node_t *&n, node_t *new_node, int i) {
148     push_delta(n);
149     if (n == NULL) {
150         n = new_node;
151     } else if (new_node->priority < n->priority) {
152         split(n, new_node->left, new_node->right, i);
153         n = new_node;
154     } else if (i <= size(n->left)) {
155         insert(n->left, new_node, i);
156     } else {
157         insert(n->right, new_node, i - size(n->left) - 1);
158     }
159     update_value(n);
160 }
161
162 static void erase(node_t *&n, int i) {

```

```

163     push_delta(n);
164     if (i == size(n->left)) {
165         delete n;
166         merge(n, n->left, n->right);
167     } else if (i < size(n->left)) {
168         erase(n->left, i);
169     } else {
170         erase(n->right, i - size(n->left) - 1);
171     }
172     update_value(n);
173 }
174
175 static node_t* select(node_t *n, int i) {
176     push_delta(n);
177     if (i < size(n->left)) {
178         return select(n->left, i);
179     }
180     if (i > size(n->left)) {
181         return select(n->right, i - size(n->left) - 1);
182     }
183     return n;
184 }
185
186 void clean_up(node_t *&n) {
187     if (n != NULL) {
188         clean_up(n->left);
189         clean_up(n->right);
190         delete n;
191     }
192 }
193
194 public:
195     implicit_treap(int n = 0, const T &v = T()) : root(NULL) {
196         for (int i = 0; i < n; i++) {
197             push_back(v);
198         }
199     }
200
201     template<class It>
202     implicit_treap(It lo, It hi) : root(NULL) {
203         for (; lo != hi; ++lo) {
204             push_back(*lo);
205         }
206     }
207
208     ~implicit_treap() {
209         clean_up(root);
210     }
211
212     int size() const {
213         return size(root);
214     }
215
216     bool empty() const {
217         return root == NULL;
218     }
219
220     void insert(int i, const T &v) {
221         insert(root, new node_t(v), i);

```

```

222     }
223
224     void erase(int i) {
225         erase(root, i);
226     }
227
228     void push_back(const T &v) {
229         insert(size(), v);
230     }
231
232     void pop_back() {
233         erase(size() - 1);
234     }
235
236     T at(int i) const {
237         return select(root, i)->value;
238     }
239
240     T query(int lo, int hi) {
241         node_t *l1, *r1, *l2, *r2, *t;
242         split(root, l1, r1, hi + 1);
243         split(l1, l2, r2, lo);
244         T res = r2->subtree_value;
245         merge(t, l2, r2);
246         merge(root, t, r1);
247         return res;
248     }
249
250     void update(int i, const T &d) {
251         update(i, i, d);
252     }
253
254     void update(int lo, int hi, const T &d) {
255         node_t *l1, *r1, *l2, *r2, *t;
256         split(root, l1, r1, hi + 1);
257         split(l1, l2, r2, lo);
258         update_delta(r2, d);
259         merge(t, l2, r2);
260         merge(root, t, r1);
261     }
262 };
263
264 /**
265  * Example Usage and Output:
266  * Values: -99 -2 1 8 10 11 (min: -99)
267  * Values: -90 -2 1 8 10 11 (min: -90)
268  * Values: 2 2 1 8 10 11 (min: 1)
269 */
270
271 #include <iostream>
272 using namespace std;
273
274 void print(implicit_treap<int> &t) {
275     cout << "Values:";
276     for (int i = 0; i < t.size(); i++) {
277         cout << " " << t.at(i);
278     }
279     cout << " (min: " << t.query(0, t.size() - 1) << ")" << endl;

```

```

281 }
282
283 int main() {
284     int arr[5] = {99, -2, 1, 8, 10};
285     implicit_treap<int> t(arr, arr + 5);
286     t.push_back(11);
287     t.push_back(12);
288     t.pop_back();
289     print(t);
290     t.insert(0, 90);
291     t.erase(1);
292     print(t);
293     t.update(0, 1, 2);
294     print(t);
295     return 0;
296 }
```

2.4 Range Queries in Two Dimensions

2.4.1 Quadtree (Point Update)

```

1 /*
2
3 Maintain a two-dimensional array while supporting dynamic queries of rectangular
4 sub-arrays and dynamic updates of individual indices. This implementation uses
5 lazy initialization of nodes to conserve memory while supporting large indices.
6
7 The query operation is defined by the join_values() and join_region() functions
8 where join_values(x, join_values(y, z)) = join_values(join_values(x, y), z) for
9 all values x, y, and z in the array. The join_region(v, area) function must be
10 defined in conjunction to efficiently return the result of join_values() applied
11 to a rectangular sub-array of area elements. The default code below assumes a
12 numerical array type, defining queries for the "min" of the target range.
13 Another possible query operation is "sum", in which case join_values(a, b)
14 should return "a + b" and join_region(v, area) should return "v*area".
15
16 The update operation is defined by the join_value_with_delta() function, which
17 determines the change made to array values. The default code below defines
18 updates that "set" the chosen array index to a new value. Another possible
19 update operation is "increment", in which join_value_with_delta(v, d) should be
20 defined to return "v + d".
21
22 - quadtree(v) constructs a two-dimensional array with rows from 0 to MAXR and
23   columns from 0 to MAXC, inclusive. All values are implicitly initialized to v.
24 - at(r, c) returns the value at row r, column c.
25 - query(r1, c1, r2, c2) returns the result of join_values() applied to every
26   value in the rectangular region consisting of rows from r1 to r2, inclusive,
27   and columns from c1 to c2, inclusive.
28 - update(r, c, d) assigns the value v at (r, c) to join_value_with_delta(v, d).
29
30 Time Complexity:
31 - O(1) per call to the constructor.
32 - O(max(MAXR, MAXC)) per call to at(), update(), and query().
```

```

33
34 Space Complexity:
35 - O(n) for storage of the array elements, where n is the number of updated
36   entries in the array.
37 - O(sqrt(max(MAXR, MAXC))) auxiliary stack space for update(), query(), and
38   at().
39
40 */
41
42 #include <algorithm>
43 #include <cstddef>
44
45 template<class T>
46 class quadtree {
47     static const int MAXR = 1000000000;
48     static const int MAXC = 1000000000;
49
50     static T join_values(const T &a, const T &b) {
51         return std::min(a, b);
52     }
53
54     static T join_region(const T &v, int area) {
55         return v;
56     }
57
58     static T join_value_with_delta(const T &v, const T &d) {
59         return d;
60     }
61
62     struct node_t {
63         T value;
64         node_t *child[4];
65
66         node_t(const T &v) : value(v) {
67             for (int i = 0; i < 4; i++) {
68                 child[i] = NULL;
69             }
70         }
71     };
72
73     node_t *root;
74     T init;
75
76     // Helper variables for query().
77     int tgt_r1, tgt_c1, tgt_r2, tgt_c2;
78     T res;
79     bool found;
80
81     void query(node_t *n, int r1, int c1, int r2, int c2) {
82         if (tgt_r2 < r1 || tgt_r1 > r2 || tgt_c2 < c1 || tgt_c1 > c2) {
83             return;
84         }
85         if (n == NULL) {
86             int rlen = std::min(r2, tgt_r2) - std::max(r1, tgt_r1) + 1;
87             int clen = std::min(c2, tgt_c2) - std::max(c1, tgt_c1) + 1;
88             T v = join_region(init, rlen*clen);
89             res = found ? join_values(res, v) : v;
90             found = true;
91             return;

```

```

92     }
93     if (tgt_r1 <= r1 && r2 <= tgt_r2 && tgt_c1 <= c1 && c2 <= tgt_c2) {
94         res = found ? join_values(res, n->value) : n->value;
95         found = true;
96         return;
97     }
98     int rmid = r1 + (r2 - r1)/2, cmid = c1 + (c2 - c1)/2;
99     query(n->child[0], r1, c1, rmid, cmid);
100    query(n->child[1], rmid + 1, c1, r2, cmid);
101    query(n->child[2], r1, cmid + 1, rmid, c2);
102    query(n->child[3], rmid + 1, cmid + 1, r2, c2);
103 }
104
105 // Helper variables for update().
106 int tgt_r, tgt_c;
107 T delta;
108
109 void update(node_t *&n, int r1, int c1, int r2, int c2) {
110     if (n == NULL) {
111         n = new node_t(join_region(init, (r2 - r1 + 1)*(c2 - c1 + 1)));
112     }
113     if (tgt_r < r1 || tgt_r > r2 || tgt_c < c1 || tgt_c > c2) {
114         return;
115     }
116     if (r1 == r2 && c1 == c2) {
117         n->value = join_value_with_delta(n->value, delta);
118         return;
119     }
120     int rmid = r1 + (r2 - r1)/2, cmid = c1 + (c2 - c1)/2;
121     update(n->child[0], r1, c1, rmid, cmid);
122     update(n->child[1], rmid + 1, c1, r2, cmid);
123     update(n->child[2], r1, cmid + 1, rmid, c2);
124     update(n->child[3], rmid + 1, cmid + 1, r2, c2);
125     bool found = false;
126     for (int i = 0; i < 4; i++) {
127         n->value = found ? join_values(n->value, n->child[i]->value)
128                           : n->child[i]->value;
129         found = true;
130     }
131 }
132
133 static void clean_up(node_t *n) {
134     if (n != NULL) {
135         for (int i = 0; i < 4; i++) {
136             clean_up(n->child[i]);
137         }
138         delete n;
139     }
140 }
141
142 public:
143     quadtree(const T &v = T()) : root(NULL), init(v) {}
144
145     ~quadtree() {
146         clean_up(root);
147     }
148
149     T at(int r, int c) {
150         return query(r, c, r, c);

```

```

151     }
152
153 T query(int r1, int c1, int r2, int c2) {
154     tgt_r1 = r1;
155     tgt_c1 = c1;
156     tgt_r2 = r2;
157     tgt_c2 = c2;
158     found = false;
159     query(root, 0, 0, MAXR, MAXC);
160     return found ? res : join_region(init, (r2 - r1 + 1)*(c2 - c1 + 1));
161 }
162
163 void update(int r, int c, const T &d) {
164     tgt_r = r;
165     tgt_c = c;
166     delta = d;
167     update(root, 0, 0, MAXR, MAXC);
168 }
169 };
170
171 /** Example Usage and Output:
172
173 Values:
174 7 6 0
175 5 4 0
176 0 1 9
177
178 ***
179
180 #include <cassert>
181 #include <iostream>
182 using namespace std;
183
184 int main() {
185     quadtree<int> t(0);
186     t.update(0, 0, 7);
187     t.update(0, 1, 6);
188     t.update(1, 0, 5);
189     t.update(1, 1, 4);
190     t.update(2, 1, 1);
191     t.update(2, 2, 9);
192     cout << "Values:" << endl;
193     for (int i = 0; i < 3; i++) {
194         for (int j = 0; j < 3; j++) {
195             cout << t.at(i, j) << " ";
196         }
197         cout << endl;
198     }
199     assert(t.query(0, 0, 0, 1) == 6);
200     assert(t.query(0, 0, 1, 0) == 5);
201     assert(t.query(1, 1, 2, 2) == 0);
202     assert(t.query(0, 0, 1000000000, 1000000000) == 0);
203     t.update(500000000, 500000000, -100);
204     assert(t.query(0, 0, 1000000000, 1000000000) == -100);
205     return 0;
206 }
```

2.4.2 Quadtree (Range Update)

```

1  /*
2
3  Maintain a two-dimensional array while supporting both dynamic queries and
4  updates of rectangular sub-arrays via the lazy propagation technique. This
5  implementation uses lazy initialization of nodes to conserve memory while
6  supporting large indices.
7
8  The query operation is defined by the join_values() and join_region() functions
9  where join_values(x, join_values(y, z)) = join_values(join_values(x, y), z) for
10 all values x, y, and z in the array. The join_region(v, area) function must be
11 defined in conjunction to efficiently return the result of join_values() applied
12 to a rectangular sub-array of area elements. The default code below assumes a
13 numerical array type, defining queries for the "min" of the target range.
14 Another possible query operation is "sum", in which case join_values(a, b)
15 should return "a + b" and join_region(v, area) should return "v*area".
16
17 The update operation is defined by the join_value_with_delta() and join_deltas()
18 functions, which determines the change made to array values. These must satisfy:
19 - join_deltas(d1, join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).
20 - join_value_with_delta(join_values(v, ... (m times) ..., v), d, m)) should be
21   equal to join_values(join_value_with_delta(v, d, 1), ... (m times)).
22 - if a sequence d_1, ..., d_m of deltas is used to update a value v, then
23   join_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent
24   to m sequential calls to join_value_with_delta(v, d_i, 1) for i = 1 .. m.
25 The default code below defines updates that "set" the chosen array index to a
26 new value. Another possible update operation is "increment", in which case
27 join_value_with_delta(v, d, area) should be defined to return "v + d*area" and
28 join_deltas(d1, d2) should be defined to return "d1 + d2".
29
30 - quadtree(v) constructs a two-dimensional array with rows from 0 to MAXR and
31   columns from 0 to MAXC, inclusive. All values are implicitly initialized to v.
32 - at(r, c) returns the value at row r, column c.
33 - query(r1, c1, r2, c2) returns the result of join_values() applied to every
34   value in the rectangular region consisting of rows from r1 to r2 and columns
35   from c1 to c2, inclusive.
36 - update(r, c, d) assigns the value v at (r, c) to join_value_with_delta(v, d).
37 - update(r1, c1, r2, c2) modifies the value at each index of the rectangular
38   region consisting of rows from r1 to r2 and columns from c1 to c2, inclusive,
39   by respectively joining them with d using join_value_with_delta().
40
41 Time Complexity:
42 - O(1) per call to the constructor.
43 - O(max(MAXR, MAXC)) per call to at(), update(), and query().
44
45 Space Complexity:
46 - O(n) for storage of the array elements, where n is the number of updated
47   entries in the array.
48 - O(sqrt(max(MAXR, MAXC))) auxiliary stack space for update(), query(), and
49   at().
50
51 */
52
53 #include <algorithm>
54 #include <cstddef>
55

```

```

56 template<class T>
57 class quadtree {
58     static const int MAXR = 1000000000;
59     static const int MAXC = 1000000000;
60
61     static T join_values(const T &a, const T &b) {
62         return std::min(a, b);
63     }
64
65     static T join_region(const T &v, int area) {
66         return v;
67     }
68
69     static T join_value_with_delta(const T &v, const T &d, int area) {
70         return d;
71     }
72
73     static T join_deltas(const T &d1, const T &d2) {
74         return d2; // For "set" updates, the more recent delta prevails.
75     }
76
77     struct node_t {
78         T value, delta;
79         bool pending;
80         node_t *child[4];
81
82         node_t(const T &v) : value(v), pending(false) {
83             for (int i = 0; i < 4; i++) {
84                 child[i] = NULL;
85             }
86         }
87     } *root;
88
89     T init;
90
91     void update_delta(node_t *&n, const T &d, int area) {
92         if (n == NULL) {
93             n = new node_t(join_region(init, area));
94         }
95         n->delta = n->pending ? join_deltas(n->delta, d) : d;
96         n->pending = true;
97     }
98
99     void push_delta(node_t *n, int r1, int c1, int r2, int c2) {
100         if (n->pending) {
101             int rmid = r1 + (r2 - r1)/2, cmid = c1 + (c2 - c1)/2;
102             int rlen = r2 - r1 + 1, clen = c2 - c1 + 1;
103             n->value = join_value_with_delta(n->value, n->delta, rlen*clen);
104             if (rlen*clen > 1) {
105                 int rlen1 = rmid - r1 + 1, rlen2 = rlen - rlen1;
106                 int clen1 = cmid - c1 + 1, clen2 = clen - clen1;
107                 update_delta(n->child[0], n->delta, rlen1*clen1);
108                 update_delta(n->child[1], n->delta, rlen2*clen1);
109                 update_delta(n->child[2], n->delta, rlen1*clen2);
110                 update_delta(n->child[3], n->delta, rlen2*clen2);
111             }
112             n->pending = false;
113         }
114     }

```

```

115
116 // Helper variables for query() and update().
117 int tgt_r1, tgt_c1, tgt_r2, tgt_c2;
118 T res, delta;
119 bool found;
120
121 void query(node_t *n, int r1, int c1, int r2, int c2) {
122 if (tgt_r2 < r1 || tgt_r1 > r2 || tgt_c2 < c1 || tgt_c1 > c2) {
123     return;
124 }
125 if (n == NULL) {
126     int rlen = std::min(r2, tgt_r2) - std::max(r1, tgt_r1) + 1;
127     int clen = std::min(c2, tgt_c2) - std::max(c1, tgt_c1) + 1;
128     T v = join_region(init, rlen*clen);
129     res = found ? join_values(res, v) : v;
130     found = true;
131     return;
132 }
133 push_delta(n, r1, c1, r2, c2);
134 if (tgt_r1 <= r1 && r2 <= tgt_r2 && tgt_c1 <= c1 && c2 <= tgt_c2) {
135     res = found ? join_values(res, n->value) : n->value;
136     found = true;
137     return;
138 }
139 int rmid = r1 + (r2 - r1)/2, cmid = c1 + (c2 - c1)/2;
140 query(n->child[0], r1, c1, rmid, cmid);
141 query(n->child[1], rmid + 1, c1, r2, cmid);
142 query(n->child[2], r1, cmid + 1, rmid, c2);
143 query(n->child[3], rmid + 1, cmid + 1, r2, c2);
144 }
145
146 void update(node_t *&n, int r1, int c1, int r2, int c2) {
147 if (n == NULL) {
148     n = new node_t(join_region(init, (r2 - r1 + 1)*(c2 - r1 + 1)));
149 }
150 if (tgt_r2 < r1 || tgt_r1 > r2 || tgt_c2 < c1 || tgt_c1 > c2) {
151     return;
152 }
153 push_delta(n, r1, c1, r2, c2);
154 if (tgt_r1 <= r1 && r2 <= tgt_r2 && tgt_c1 <= c1 && c2 <= tgt_c2) {
155     n->delta = delta;
156     n->pending = true;
157     push_delta(n, r1, c1, r2, c2);
158     return;
159 }
160 int rmid = r1 + (r2 - r1)/2, cmid = c1 + (c2 - c1)/2;
161 update(n->child[0], r1, c1, rmid, cmid);
162 update(n->child[1], rmid + 1, c1, r2, cmid);
163 update(n->child[2], r1, cmid + 1, rmid, c2);
164 update(n->child[3], rmid + 1, cmid + 1, r2, c2);
165 n->value = n->child[0]->value;
166 for (int i = 1; i < 4; i++) {
167     n->value = join_values(n->value, n->child[i]->value);
168 }
169
170 static void clean_up(node_t *n) {
171 if (n != NULL) {
172     for (int i = 0; i < 4; i++) {

```

```

174         clean_up(n->child[i]);
175     }
176     delete n;
177 }
178 }
179
180 public:
181     quadtree(const T &v = T()) : root(NULL), init(v) {}
182
183     ~quadtree() {
184         clean_up(root);
185     }
186
187     T at(int r, int c) {
188         return query(r, c, r, c);
189     }
190
191     T query(int r1, int c1, int r2, int c2) {
192         tgt_r1 = r1;
193         tgt_c1 = c1;
194         tgt_r2 = r2;
195         tgt_c2 = c2;
196         found = false;
197         query(root, 0, 0, MAXR, MAXC);
198         return found ? res : join_region(init, (r2 - r1 + 1)*(c2 - c1 + 1));
199     }
200
201     void update(int r, int c, const T &d) {
202         update(r, c, r, c, d);
203     }
204
205     void update(int r1, int c1, int r2, int c2, const T &d) {
206         tgt_r1 = r1;
207         tgt_c1 = c1;
208         tgt_r2 = r2;
209         tgt_c2 = c2;
210         delta = d;
211         update(root, 0, 0, MAXR, MAXC);
212     }
213 };
214
215 /**
216  * Example Usage and Output:
217  * Values:
218  * 7 6 0
219  * 5 4 0
220  * 0 1 9
221  */
222
223
224 #include <cassert>
225 #include <iostream>
226 using namespace std;
227
228 int main() {
229     quadtree<int> t(0);
230     t.update(0, 0, 7);
231     t.update(0, 1, 6);
232     t.update(1, 0, 5);

```

```

233     t.update(1, 1, 4);
234     t.update(2, 1, 1);
235     t.update(2, 2, 9);
236     cout << "Values:" << endl;
237     for (int i = 0; i < 3; i++) {
238         for (int j = 0; j < 3; j++) {
239             cout << t.at(i, j) << " ";
240         }
241     cout << endl;
242 }
243 assert(t.query(0, 0, 0, 1) == 6);
244 assert(t.query(0, 0, 1, 0) == 5);
245 assert(t.query(1, 1, 2, 2) == 0);
246 assert(t.query(0, 0, 1000000000, 1000000000) == 0);
247 t.update(500000000, 500000000, -100);
248 assert(t.query(0, 0, 1000000000, 1000000000) == -100);
249 return 0;
250 }
```

2.4.3 2D Segment Tree

```

1 /*
2
3 Maintain a two-dimensional array while supporting dynamic queries of rectangular
4 sub-arrays and dynamic updates of individual indices. This implementation uses
5 lazy initialization of nodes to conserve memory while supporting large indices.
6
7 The query operation is defined by the join_values() and join_region() functions
8 where join_values(x, join_values(y, z)) = join_values(join_values(x, y), z) for
9 all values x, y, and z in the array. The join_region(v, area) function must be
10 defined in conjunction to efficiently return the result of join_values() applied
11 to a rectangular sub-array of area elements. The default code below assumes a
12 numerical array type, defining queries for the "min" of the target range.
13 Another possible query operation is "sum", in which case join_values(a, b)
14 should return "a + b" and join_region(v, area) should return "v*area".
15
16 The update operation is defined by the join_value_with_delta() function, which
17 determines the change made to array values. The default code below defines
18 updates that "set" the chosen array index to a new value. Another possible
19 update operation is "increment", in which join_value_with_delta(v, d) should be
20 defined to return "v + d".
21
22 - segment_tree_2d(v) constructs a two-dimensional array with rows from 0 to
23 MAXR and columns from 0 to MAXC, inclusive. All values are implicitly
24 initialized to v.
25 - at(r, c) returns the value at row r, column c.
26 - query(r1, c1, r2, c2) returns the result of join_values() applied to every
27 value in the rectangular region consisting of rows from r1 to r2, inclusive,
28 and columns from c1 to c2, inclusive.
29 - update(r, c, d) assigns the value v at (r, c) to join_value_with_delta(v, d).
30
31 Time Complexity:
32 - O(1) per call to the constructor.
33 - O(log(MAXR)*log(MAXC)) per call to at(), update(), and query().
34
```

```

35 Space Complexity:
36 - O(n) for storage of the array elements, where n is the number of updated
37   entries in the array.
38 - O(log(MAXR) + log(MAXC)) auxiliary stack space for update(), query(), and
39   at().
40
41 */
42
43 #include <algorithm>
44 #include <cstddef>
45
46 template<class T>
47 class segment_tree_2d {
48     static const int MAXR = 1000000000;
49     static const int MAXC = 1000000000;
50
51     static T join_values(const T &a, const T &b) {
52         return std::min(a, b);
53     }
54
55     static T join_region(const T &v, int area) {
56         return v;
57     }
58
59     static T join_value_with_delta(const T &v, const T &d) {
60         return d;
61     }
62
63     struct inner_node_t {
64         T value;
65         int low, high;
66         inner_node_t *left, *right;
67
68         inner_node_t(int lo, int hi, const T &v)
69             : value(v), low(lo), high(hi), left(NULL), right(NULL) {}
70     };
71
72     struct outer_node_t {
73         inner_node_t root;
74         int low, high;
75         outer_node_t *left, *right;
76
77         outer_node_t(int lo, int hi, const T &v)
78             : root(0, MAXC, v), low(lo), high(hi), left(NULL), right(NULL) {}
79     } *root;
80
81     T init;
82
83     // Helper variables for query() and update().
84     int tgt_r1, tgt_c1, tgt_r2, tgt_c2, width;
85
86     template<class node_t>
87     inline T call_query(node_t *n, int area) {
88         return (n != NULL) ? query(n) : join_region(init, area);
89     }
90
91     T query(inner_node_t *n) {
92         int lo = n->low, hi = n->high, mid = lo + (hi - lo)/2;
93         if (tgt_c1 <= lo && hi <= tgt_c2) {

```

```

94     T res = n->value;
95     if (tgt_c1 < lo) {
96         res = join_values(res, join_region(init, lo - tgt_c1 + 1));
97     }
98     if (hi < tgt_c2) {
99         res = join_values(res, join_region(init, tgt_c2 - hi + 1));
100    }
101   return res;
102 } else if (tgt_c2 <= mid) {
103     return call_query(n->left, tgt_c2 - tgt_c1 + 1);
104 } else if (mid < tgt_c1) {
105     return call_query(n->right, tgt_c2 - tgt_c1 + 1);
106 }
107 return join_values(
108     call_query(n->left, std::min(tgt_c2, mid) - tgt_c1 + 1),
109     call_query(n->right, tgt_c2 - std::max(tgt_c1, mid + 1) + 1));
110 }
111
112 T query(outer_node_t *n) {
113     int lo = n->low, hi = n->high, mid = lo + (hi - lo)/2;
114     if (tgt_r1 <= lo && hi <= tgt_r2) {
115         T res = query(&(n->root));
116         if (tgt_r1 < lo) {
117             res = join_values(res, join_region(init, width*(lo - tgt_r1 + 1)));
118         }
119         if (hi < tgt_r2) {
120             res = join_values(res, join_region(init, width*(tgt_r2 - hi + 1)));
121         }
122         return res;
123     } else if (tgt_r2 <= mid) {
124         return call_query(n->left, tgt_r2 - tgt_r1 + 1);
125     } else if (mid < tgt_r1) {
126         return call_query(n->right, tgt_r2 - tgt_r1 + 1);
127     }
128     return join_values(
129         call_query(n->left, width*(std::min(tgt_r2, mid) - tgt_r1 + 1)),
130         call_query(n->right, width*(tgt_r2 - std::max(tgt_r1, mid + 1) + 1)));
131 }
132
133 void update(inner_node_t *n, int c, const T &d, bool leaf_row) {
134     int lo = n->low, hi = n->high, mid = lo + (hi - lo)/2;
135     if (lo == hi) {
136         if (leaf_row) {
137             n->value = join_value_with_delta(n->value, d);
138         } else {
139             n->value = d;
140         }
141         return;
142     }
143     inner_node_t *&target = (c <= mid) ? n->left : n->right;
144     if (target == NULL) {
145         target = new inner_node_t(c, c, init);
146     }
147     if (target->low <= c && c <= target->high) {
148         update(target, c, d, leaf_row);
149     } else {
150         do {
151             if (c <= mid) {
152                 hi = mid;

```

```

153     } else {
154         lo = mid + 1;
155     }
156     mid = lo + (hi - lo)/2;
157 } while ((c <= mid) == (target->low <= mid));
158 inner_node_t *tmp = new inner_node_t(lo, hi, init);
159 if (target->low <= mid) {
160     tmp->left = target;
161 } else {
162     tmp->right = target;
163 }
164 target = tmp;
165 update(tmp, c, d, leaf_row);
166 }
167 T left_value = (n->left != NULL) ? n->left->value
168                               : join_region(init, mid - lo + 1);
169 T right_value = (n->right != NULL) ? n->right->value
170                           : join_region(init, hi - mid);
171 n->value = join_values(left_value, right_value);
172 }
173
174 void update(outer_node_t *n, int r, int c, const T &d) {
175     int lo = n->low, hi = n->high, mid = lo + (hi - lo)/2;
176     if (lo == hi) {
177         update(&(n->root), c, d, true);
178         return;
179     }
180     if (r <= mid) {
181         if (n->left == NULL) {
182             n->left = new outer_node_t(lo, mid, init);
183         }
184         update(n->left, r, c, d);
185     } else {
186         if (n->right == NULL) {
187             n->right = new outer_node_t(mid + 1, hi, init);
188         }
189         update(n->right, r, c, d);
190     }
191     T value = join_region(init, hi - lo + 1);
192     if (n->left != NULL || n->right != NULL) {
193         tgt_c1 = tgt_c2 = c;
194         T left_value = (n->left != NULL) ? query(&(n->left->root))
195                                         : join_region(init, mid - lo + 1);
196         T right_value = (n->right != NULL) ? query(&(n->right->root))
197                                         : join_region(init, hi - mid);
198         value = join_values(left_value, right_value);
199     }
200     update(&(n->root), c, value, false);
201 }
202
203 static void clean_up(inner_node_t *n) {
204     if (n != NULL) {
205         clean_up(n->left);
206         clean_up(n->right);
207         delete n;
208     }
209 }
210
211 static void clean_up(outer_node_t *n) {

```

```

212     if (n != NULL) {
213         clean_up(n->root.left);
214         clean_up(n->root.right);
215         clean_up(n->left);
216         clean_up(n->right);
217         delete n;
218     }
219 }
220
221 public:
222     segment_tree_2d(const T &v = T())
223         : root(new outer_node_t(0, MAXR, v)), init(v) {}
224
225     ~segment_tree_2d() {
226         clean_up(root);
227     }
228
229     T at(int r, int c) {
230         return query(r, c, r, c);
231     }
232
233     T query(int r1, int c1, int r2, int c2) {
234         tgt_r1 = r1;
235         tgt_c1 = c1;
236         tgt_r2 = r2;
237         tgt_c2 = c2;
238         width = c2 - c1;
239         return query(root);
240     }
241
242     void update(int r, int c, const T &d) {
243         update(root, r, c, d);
244     }
245 };
246
247 /** Example Usage and Output:
248
249 Values:
250 7 6 0
251 5 4 0
252 0 1 9
253
254 */
255
256 #include <cassert>
257 #include <iostream>
258 using namespace std;
259
260 int main() {
261     segment_tree_2d<int> t(0);
262     t.update(0, 0, 7);
263     t.update(0, 1, 6);
264     t.update(1, 0, 5);
265     t.update(1, 1, 4);
266     t.update(2, 1, 1);
267     t.update(2, 2, 9);
268     cout << "Values:" << endl;
269     for (int i = 0; i < 3; i++) {
270         for (int j = 0; j < 3; j++) {

```

```

271     cout << t.at(i, j) << " ";
272 }
273 cout << endl;
274 }
275 assert(t.query(0, 0, 0, 1) == 6);
276 assert(t.query(0, 0, 1, 0) == 5);
277 assert(t.query(1, 1, 2, 2) == 0);
278 assert(t.query(0, 0, 1000000000, 1000000000) == 0);
279 t.update(500000000, 500000000, -100);
280 assert(t.query(0, 0, 1000000000, 1000000000) == -100);
281 return 0;
282 }
```

2.4.4 2D Range Tree

```

1 /*
2
3 Maintain a set of two-dimensional points while supporting queries for all points
4 that fall inside given rectangular regions. This implementation uses std::pair
5 to represent points, requiring operators < and == to be defined on the numeric
6 template type.
7
8 - range_tree(lo, hi) constructs a set from two random-access iterators to
9   std::pair as a range [lo, hi) of points.
10 - query(x1, y1, x2, y2, f) calls the function f(i, p) on each point in the set
11   that falls into the rectangular region consisting of rows from x1 to x2,
12   inclusive, and columns from y1 to y2, inclusive. The first argument to f is
13   the zero-based index of the point in the original range given to the
14   constructor. The second argument is the point itself as an std::pair.
15
16 Time Complexity:
17 - O(n log n) per call to the constructor, where n is the number of points.
18 - O(log^2(n) + m) per call to query(), where m is the number of points that are
19   reported by the query.
20
21 Space Complexity:
22 - O(n log n) for storage of the points.
23 - O(log^2(n)) auxiliary stack space for query().
24 */
25
26
27 #include <algorithm>
28 #include <iterator>
29 #include <utility>
30 #include <vector>
31
32 template<class T>
33 class range_tree {
34     typedef std::pair<T, T> point;
35     typedef std::pair<int, T> colindex;
36
37     std::vector<point> points;
38     std::vector<std::vector<colindex> > columns;
39
40     static inline bool comp1(const colindex &a, const colindex &b) {
```

```

41     return a.second < b.second;
42 }
43
44 static inline bool comp2(const colindex &a, const T &v) {
45     return a.second < v;
46 }
47
48 void build(int n, int lo, int hi) {
49     if (points[lo].first == points[hi].first) {
50         for (int i = lo; i <= hi; i++) {
51             columns[n].push_back(point(i, points[i].second));
52         }
53         return;
54     }
55     int l = n*2 + 1, r = n*2 + 2, mid = lo + (hi - lo)/2;
56     build(l, lo, mid);
57     build(r, mid + 1, hi);
58     columns[n].resize(columns[l].size() + columns[r].size());
59     std::merge(columns[l].begin(), columns[l].end(),
60                columns[r].begin(), columns[r].end(),
61                columns[n].begin(), comp1);
62 }
63
64 // Helper variables for query().
65 T x1, y1, x2, y2;
66
67 template<class ReportFunction>
68 void query(int n, int lo, int hi, ReportFunction f) {
69     if (points[hi].first < x1 || x2 < points[lo].first) {
70         return;
71     }
72     if (!(points[lo].first < x1 || x2 < points[hi].first)) {
73         if (!columns[n].empty() && !(y2 < y1)) {
74             typename std::vector<point>::iterator it;
75             it = std::lower_bound(columns[n].begin(), columns[n].end(), y1, comp2);
76             for (; it != columns[n].end() && it->second <= y2; ++it) {
77                 f(it->first, points[it->first]);
78             }
79         }
80     } else if (lo != hi) {
81         int mid = lo + (hi - lo)/2;
82         query(n*2 + 1, lo, mid, f);
83         query(n*2 + 2, mid + 1, hi, f);
84     }
85 }
86
87 public:
88 template<class It>
89 range_tree(It lo, It hi) : points(lo, hi) {
90     int n = std::distance(lo, hi);
91     columns.resize(4*n + 1);
92     std::sort(points.begin(), points.end());
93     build(0, 0, n - 1);
94 }
95
96 template<class ReportFunction>
97 void query(const T &x1, const T &y1, const T &x2, const T &y2,
98            ReportFunction f) {
99     this->x1 = x1;

```

```

100     this->y1 = y1;
101     this->x2 = x2;
102     this->y2 = y2;
103     query(0, 0, points.size() - 1, f);
104 }
105 };
106
107 /** Example Usage and Output:
108
109 (-1, -1) (2, -1) (2, 2) (1, 4)
110 (1, 4) (2, 2) (3, 1)
111
112 ***/
113
114 #include <iostream>
115 using namespace std;
116
117 void print(int i, const pair<int, int> &p) {
118     cout << "(" << p.first << ", " << p.second << ")";
119 }
120
121 int main() {
122     const int n = 10;
123     int points[n][2] = {{1, 4}, {5, 4}, {2, 2}, {3, 1}, {6, -5}, {5, -1},
124                         {3, -3}, {-1, -2}, {-1, -1}, {2, -1}};
125     vector<pair<int, int> > v;
126     for (int i = 0; i < n; i++) {
127         v.push_back(make_pair(points[i][0], points[i][1]));
128     }
129     range_tree<int> t(v.begin(), v.end());
130     t.query(-1, -1, 2, 5, print);
131     cout << endl;
132     t.query(1, 1, 4, 8, print);
133     cout << endl;
134     return 0;
135 }
```

2.4.5 K-d Tree (2D Range Query)

```

1 /*
2
3 Maintain a set of two-dimensional points while supporting queries for all points
4 that fall inside given rectangular regions. This implementation uses std::pair
5 to represent points, requiring operators < and == to be defined on the numeric
6 template type.
7
8 - kd_tree(lo, hi) constructs a set from two random-access iterators to std::pair
9   as a range [lo, hi) of points.
10 - query(x1, y1, x2, y2, f) calls the function f(i, p) on each point in the set
11   that falls into the rectangular region consisting of rows from x1 to x2,
12   inclusive, and columns from y1 to y2, inclusive. The first argument to f is
13   the zero-based index of the point in the original range given to the
14   constructor. The second argument is the point itself as an std::pair.
15
16 Time Complexity:
```

```

17 - O(n log n) per call to the constructor, where n is the number of points.
18 - O(log(n) + m) on average per call to query(), where m is the number of points
19 that are reported by the query.
20
21 Space Complexity:
22 - O(n) for storage of the points.
23 - O(log n) auxiliary stack space for query().
24
25 */
26
27 #include <algorithm>
28 #include <utility>
29 #include <vector>
30
31 template<class T>
32 class kd_tree {
33     typedef std::pair<T, T> point;
34
35     static inline bool comp1(const point &a, const point &b) {
36         return a.first < b.first;
37     }
38
39     static inline bool comp2(const point &a, const point &b) {
40         return a.second < b.second;
41     }
42
43     std::vector<point> tree, minp, maxp;
44     std::vector<int> l_index, h_index;
45
46     void build(int lo, int hi, bool div_x) {
47         if (lo >= hi) {
48             return;
49         }
50         int mid = lo + (hi - lo)/2;
51         std::nth_element(tree.begin() + lo, tree.begin() + mid, tree.begin() + hi,
52                         div_x ? comp1 : comp2);
53         l_index[mid] = lo;
54         h_index[mid] = hi;
55         minp[mid].first = maxp[mid].first = tree[lo].first;
56         minp[mid].second = maxp[mid].second = tree[lo].second;
57         for (int i = lo + 1; i < hi; i++) {
58             minp[mid].first = std::min(minp[mid].first, tree[i].first);
59             minp[mid].second = std::min(minp[mid].second, tree[i].second);
60             maxp[mid].first = std::max(maxp[mid].first, tree[i].first);
61             maxp[mid].second = std::max(maxp[mid].second, tree[i].second);
62         }
63         build(lo, mid, !div_x);
64         build(mid + 1, hi, !div_x);
65     }
66
67 // Helper variables for query().
68 T x1, y1, x2, y2;
69
70 template<class ReportFunction>
71 void query(int lo, int hi, ReportFunction f) {
72     if (lo >= hi) {
73         return;
74     }
75     int mid = lo + (hi - lo)/2;

```

```

76     T ax = minp[mid].first, ay = minp[mid].second;
77     T bx = maxp[mid].first, by = maxp[mid].second;
78     if (x2 < ax || bx < x1 || y2 < ay || by < y1) {
79         return;
80     }
81     if (!(ax < x1 || x2 < bx || ay < y1 || y2 < by)) {
82         for (int i = l_index[mid]; i < h_index[mid]; i++) {
83             f(tree[i]);
84         }
85         return;
86     }
87     query(lo, mid, f);
88     query(mid + 1, hi, f);
89     if (tree[mid].first < x1 || x2 < tree[mid].first ||
90         tree[mid].second < y1 || y2 < tree[mid].second) {
91         return;
92     }
93     f(tree[mid]);
94 }
95
96 public:
97     template<class It>
98     kd_tree(It lo, It hi) : tree(lo, hi) {
99         int n = std::distance(lo, hi);
100        l_index.resize(n);
101        h_index.resize(n);
102        minp.resize(n);
103        maxp.resize(n);
104        build(0, n, true);
105    }
106
107    template<class ReportFunction>
108    void query(const T &x1, const T &y1, const T &x2, const T &y2,
109               ReportFunction f) {
110        this->x1 = x1;
111        this->y1 = y1;
112        this->x2 = x2;
113        this->y2 = y2;
114        query(0, tree.size(), f);
115    }
116 };
117
118 /**
119  (2, -1) (1, 4) (2, 2) (-1, -1)
120  (1, 4) (2, 2) (3, 1)
121 */
122
123
124 #include <iostream>
125 using namespace std;
126
127
128 void print(const pair<int, int> &p) {
129     cout << "(" << p.first << ", " << p.second << ")";
130 }
131
132 int main() {
133     const int n = 10;
134     int points[n][2] = {{1, 4}, {5, 4}, {2, 2}, {3, 1}, {6, -5}, {5, -1},

```

```

135             {3, -3}, {-1, -2}, {-1, -1}, {2, -1}};
136     vector<pair<int, int>> v;
137     for (int i = 0; i < n; i++) {
138         v.push_back(make_pair(points[i][0], points[i][1]));
139     }
140     kd_tree<int> t(v.begin(), v.end());
141     t.query(-1, -1, 2, 5, print);
142     cout << endl;
143     t.query(1, 1, 4, 8, print);
144     cout << endl;
145     return 0;
146 }
```

2.4.6 K-d Tree (Nearest Neighbor)

```

1  /*
2
3  Maintain a set of two-dimensional points while supporting queries for the
4  closest point in the set to a given query point. This implementation uses
5  std::pair to represent points, requiring operators <, ==, -, and long double
6  casting to be defined on the numeric template type.
7
8  - kd_tree(lo, hi) constructs a set from two random-access iterators to std::pair
9    as a range [lo, hi) of points.
10 - nearest(x, y, can_equal) returns a point in the set that is closest to (x, y)
11   by Euclidean distance. This may be equal to (x, y) only if can_equal is true.
12
13 Time Complexity:
14 - O(n log n) per call to the constructor, where n is the number of points.
15 - O(log n) on average per call to nearest().
16
17 Space Complexity:
18 - O(n) for storage of the points.
19 - O(log n) auxiliary stack space for nearest().
20 */
21
22 #include <algorithm>
23 #include <limits>
24 #include <stdexcept>
25 #include <utility>
26 #include <vector>
27
28 template<class T>
29 class kd_tree {
30     typedef std::pair<T, T> point;
31
32     static inline bool comp1(const point &a, const point &b) {
33         return a.first < b.first;
34     }
35
36     static inline bool comp2(const point &a, const point &b) {
37         return a.second < b.second;
38     }
39 }
```

```

41     std::vector<point> tree;
42     std::vector<bool> div_x;
43
44     void build(int lo, int hi) {
45         if (lo >= hi) {
46             return;
47         }
48         int mid = lo + (hi - lo)/2;
49         T minx, maxx, miny, maxy;
50         minx = maxx = tree[lo].first;
51         miny = maxy = tree[lo].second;
52         for (int i = lo + 1; i < hi; i++) {
53             minx = std::min(minx, tree[i].first);
54             miny = std::min(miny, tree[i].second);
55             maxx = std::max(maxx, tree[i].first);
56             maxy = std::max(maxy, tree[i].second);
57         }
58         div_x[mid] = !((maxx - minx) < (maxy - miny));
59         std::nth_element(tree.begin() + lo, tree.begin() + mid, tree.begin() + hi,
60                         div_x[mid] ? comp1 : comp2);
61         if (lo + 1 == hi) {
62             return;
63         }
64         build(lo, mid);
65         build(mid + 1, hi);
66     }
67
68     // Helper variables for nearest().
69     long double min_dist;
70     int id;
71
72     void nearest(int lo, int hi, const T &x, const T &y, bool can_equal) {
73         if (lo >= hi) {
74             return;
75         }
76         int mid = lo + (hi - lo)/2;
77         T dx = x - tree[mid].first, dy = y - tree[mid].second;
78         long double d = dx*(long double)dx + dy*(long double)dy;
79         if (d < min_dist && (can_equal || d != 0)) {
80             min_dist = d;
81             id = mid;
82         }
83         if (lo + 1 == hi) {
84             return;
85         }
86         d = (long double)(div_x[mid] ? dx : dy);
87         int l1 = lo, r1 = mid, l2 = mid + 1, r2 = hi;
88         if (d > 0) {
89             std::swap(l1, l2);
90             std::swap(r1, r2);
91         }
92         nearest(l1, r1, x, y, can_equal);
93         if (d*(long double)d < min_dist) {
94             nearest(l2, r2, x, y, can_equal);
95         }
96     }
97
98     public:
99     template<class It>

```

```

100 kd_tree(It lo, It hi) : tree(lo, hi) {
101     int n = std::distance(lo, hi);
102     if (n <= 1) {
103         throw std::runtime_error("K-d tree must be have at least 2 points.");
104     }
105     div_x.resize(n);
106     build(0, n);
107 }
108
109 point nearest(const T &x, const T &y, bool can_equal = true) {
110     min_dist = std::numeric_limits<long double>::max();
111     nearest(0, tree.size(), x, y, can_equal);
112     return tree[id];
113 }
114 };
115
116 /** Example Usage **/
117
118 #include <cassert>
119 using namespace std;
120
121 int main() {
122     pair<int, int> p[3];
123     p[0] = make_pair(0, 2);
124     p[1] = make_pair(0, 3);
125     p[2] = make_pair(-1, 0);
126     kd_tree<int> t(p, p + 3);
127     assert(t.nearest(0, 2, true) == make_pair(0, 2));
128     assert(t.nearest(0, 2, false) == make_pair(0, 3));
129     assert(t.nearest(0, 0) == make_pair(-1, 0));
130     assert(t.nearest(-10000, 0) == make_pair(-1, 0));
131     return 0;
132 }
```

2.4.7 R-Tree (Nearest Segment)

```

1 /*
2
3 Maintain a set of two-dimensional line segments while supporting queries for the
4 closest segment in the set to a given query point. This implementation uses
5 integer points and long doubles for intermediate calculations.
6
7 - r_tree(lo, hi) constructs a set from two random-access iterators as a range
8   [lo, hi) of segments.
9 - nearest(x, y) returns a segment in the set that contains some point which is
10   as close or closer to (x, y) by Euclidean distance than any point on any
11   other segment in the set.
12
13 Time Complexity:
14 - O(n log n) per call to the constructor, where n is the number of segments.
15 - O(log n) on average per call to nearest().
16
17 Space Complexity:
18 - O(n) for storage of the segments.
19 - O(log n) auxiliary stack space for nearest().
```

```

20
21 */
22
23 #include <algorithm>
24 #include <limits>
25 #include <stdexcept>
26 #include <vector>
27
28 struct segment {
29     int x1, y1, x2, y2;
30
31     segment() : x1(0), y1(0), x2(0), y2(0) {}
32     segment(int x1, int y1, int x2, int y2) : x1(x1), y1(y1), x2(x2), y2(y2) {}
33
34     bool operator==(const segment &s) const {
35         return (x1 == s.x1) && (y1 == s.y1) && (x2 == s.x2) && (y2 == s.y2);
36     }
37 };
38
39 class r_tree {
40     static inline bool cmp_x(const segment &a, const segment &b) {
41         return a.x1 + a.x2 < b.x1 + b.x2;
42     }
43
44     static inline bool cmp_y(const segment &a, const segment &b) {
45         return a.y1 + a.y2 < b.y1 + b.y2;
46     }
47
48     static inline int seg_dist(int v, int lo, int hi) {
49         return (v <= lo) ? (lo - v) : (v >= hi ? v - hi : 0);
50     }
51
52     static long double point_to_segment_squared(int x, int y, const segment &s) {
53         long long dx = s.x2 - s.x1, dy = s.y2 - s.y1;
54         long long px = x - s.x1, py = y - s.y1;
55         long long sqdist = dx*dx + dy*dy;
56         long long dot = dx*px + dy*py;
57         if (dot <= 0 || sqdist == 0) {
58             return px*px + py*py;
59         }
60         if (dot >= sqdist) {
61             return (px - dx)*(px - dx) + (py - dy)*(py - dy);
62         }
63         double q = (double)dot / sqdist;
64         return (px - q*dx)*(px - q*dx) + (py - q*dy)*(py - q*dy);
65     }
66
67     std::vector<segment> tree;
68     std::vector<int> minx, maxx, miny, maxy;
69
70     void build(int lo, int hi, bool div_x) {
71         if (lo >= hi) {
72             return;
73         }
74         int mid = lo + (hi - lo)/2;
75         std::nth_element(tree.begin() + lo, tree.begin() + mid, tree.begin() + hi,
76                         div_x ? cmp_x : cmp_y);
77         for (int i = lo; i < hi; i++) {
78             minx[mid] = std::min(minx[mid], std::min(tree[i].x1, tree[i].x2));

```

```

79     miny[mid] = std::min(miny[mid], std::min(tree[i].y1, tree[i].y2));
80     maxx[mid] = std::max(maxx[mid], std::max(tree[i].x1, tree[i].x2));
81     maxy[mid] = std::max(maxy[mid], std::max(tree[i].y1, tree[i].y2));
82 }
83 build(lo, mid, !div_x);
84 build(mid + 1, hi, !div_x);
85 }
86
87 // Helper variables for nearest().
88 double min_dist;
89 int id;
90
91 void nearest(int lo, int hi, int x, int y, bool div_x) {
92     if (lo >= hi) {
93         return;
94     }
95     int mid = lo + (hi - lo)/2;
96     long double d = point_to_segment_squared(x, y, tree[mid]);
97     if (min_dist > d) {
98         min_dist = d;
99         id = mid;
100    }
101    long long delta = div_x ? (2*x - tree[mid].x1 - tree[mid].x2)
102                           : (2*y - tree[mid].y1 - tree[mid].y2);
103    if (delta <= 0) {
104        nearest(lo, mid, x, y, !div_x);
105        if (mid + 1 < hi) {
106            int midi1 = (mid + hi + 1)/2;
107            long long dist = div_x ? seg_dist(x, minx[midi1], maxx[midi1])
108                           : seg_dist(y, miny[midi1], maxy[midi1]);
109            if (dist*dist < min_dist) {
110                nearest(mid + 1, hi, x, y, !div_x);
111            }
112        }
113    } else {
114        nearest(mid + 1, hi, x, y, !div_x);
115        if (lo < mid) {
116            int midi1 = lo + (mid - lo)/2;
117            long long dist = div_x ? seg_dist(x, minx[midi1], maxx[midi1])
118                           : seg_dist(y, miny[midi1], maxy[midi1]);
119            if (dist*dist < min_dist) {
120                nearest(lo, mid, x, y, !div_x);
121            }
122        }
123    }
124 }
125
126 public:
127     template<class It>
128     r_tree(It lo, It hi) : tree(lo, hi) {
129         int n = std::distance(lo, hi);
130         if (n <= 1) {
131             throw std::runtime_error("R-tree must be have at least 2 segments.");
132         }
133         minx.assign(n, std::numeric_limits<int>::max());
134         maxx.assign(n, std::numeric_limits<int>::min());
135         miny.assign(n, std::numeric_limits<int>::max());
136         maxy.assign(n, std::numeric_limits<int>::min());
137         build(0, n, true);

```

```

138     }
139
140     segment nearest(int x, int y) {
141         min_dist = std::numeric_limits<long double>::max();
142         nearest(0, tree.size(), x, y, true);
143         return tree[id];
144     }
145 };
146
147 /** Example Usage **/
148
149 #include <cassert>
150 using namespace std;
151
152 int main() {
153     segment s[4];
154     s[0] = segment(0, 0, 0, 4);
155     s[1] = segment(0, 4, 4, 4);
156     s[2] = segment(4, 4, 4, 0);
157     s[3] = segment(4, 0, 0, 0);
158     r_tree t(s, s + 4);
159     assert(t.nearest(-1, 2) == segment(0, 0, 0, 4));
160     assert(t.nearest(100, 100) == segment(4, 4, 4, 0));
161     return 0;
162 }
```

2.5 Fenwick Trees

2.5.1 Fenwick Tree (Simple)

```

1 /*
2
3 Maintain an array of numerical type, allowing for updates of individual indices
4 (point update) and queries for the sum of contiguous sub-arrays (range queries).
5 This implementation assumes that the array is 1-based (i.e. has valid indices
6 from 1 to MAXN - 1, inclusive).
7
8 - initialize() resets the data structure.
9 - a[i] stores the value at index i.
10 - add(i, x) adds x to the value at index i.
11 - set(i, x) assigns the value at index i to x.
12 - sum(hi) returns the sum of all values at indices from 1 to hi, inclusive.
13 - sum(lo, hi) returns the sum of all values at indices from lo to hi, inclusive.
14
15 Time Complexity:
16 - O(n) per call to initialize(), where n is the size of the array.
17 - O(log n) per call to all other operations.
18
19 Space Complexity:
20 - O(n) for storage of the array elements.
21 - O(1) auxiliary for all operations.
22
23 */
```

```

24
25 const int MAXN = 1000;
26 int a[MAXN + 1], t[MAXN + 1];
27
28 void initialize() {
29     for (int i = 0; i <= MAXN; i++) {
30         a[i] = t[i] = 0;
31     }
32 }
33
34 void add(int i, int x) {
35     a[i] += x;
36     for (; i <= MAXN; i += i & -i) {
37         t[i] += x;
38     }
39 }
40
41 void set(int i, int x) {
42     add(i, x - a[i]);
43 }
44
45 int sum(int hi) {
46     int res = 0;
47     for (; hi > 0; hi -= hi & -hi) {
48         res += t[hi];
49     }
50     return res;
51 }
52
53 int sum(int lo, int hi) {
54     return sum(hi) - sum(lo - 1);
55 }
56
57 /** Example Usage and Output:
58
59 Values: 5 1 2 3 4
60
61 */
62
63 #include <cassert>
64 #include <iostream>
65 using namespace std;
66
67 int main() {
68     int v[] = {10, 1, 2, 3, 4};
69     initialize();
70     for (int i = 1; i <= 5; i++) {
71         set(i, v[i - 1]);
72     }
73     add(1, -5);
74     cout << "Values: ";
75     for (int i = 1; i <= 5; i++) {
76         cout << a[i] << " ";
77     }
78     cout << endl;
79     assert(sum(2, 4) == 6);
80     return 0;
81 }
```

2.5.2 Fenwick Tree (Range Update, Point Query)

```

1  /*
2
3  Maintain an array of numerical type, allowing for contiguous sub-arrays to be
4  simultaneously incremented by arbitrary values (range update) and values at
5  individual indices to be queried (point query). This implementation assumes that
6  the array is 0-based (i.e. has valid indices from 0 to size() - 1, inclusive).
7
8  - size() returns the size of the array.
9  - at(i) returns the value at index i.
10 - add(i, x) adds x to the value at index i.
11 - add(lo, hi, x) adds x to the values at all indices from lo to hi, inclusive.
12
13 Time Complexity:
14 - O(n) per call to the constructor, where n is the size of the array.
15 - O(1) per call to size().
16 - O(log n) per call to at() and both add() functions.
17
18 Space Complexity:
19 - O(n) for storage of the array elements.
20 - O(1) auxiliary for all operations.
21
22 */
23
24 #include <vector>
25
26 template<class T>
27 class fenwick_tree {
28     int len;
29     std::vector<int> t;
30
31 public:
32     fenwick_tree(int n) : len(n), t(n + 2) {}
33
34     int size() const {
35         return len;
36     }
37
38     T at(int i) const {
39         T res = 0;
40         for (i++; i > 0; i -= i & -i) {
41             res += t[i];
42         }
43         return res;
44     }
45
46     void add(int i, const T &x) {
47         for (i++; i <= len + 1; i += i & -i) {
48             t[i] += x;
49         }
50     }
51
52     void add(int lo, int hi, const T &x) {
53         add(lo, x);
54         add(hi + 1, -x);
55     }

```

```

56 };
57
58 /** Example Usage and Output:
59
60 Values: 5 10 15 10 10
61
62 ***
63
64 #include <iostream>
65 using namespace std;
66
67 int main() {
68     fenwick_tree<int> t(5);
69     t.add(0, 1, 5);
70     t.add(1, 2, 5);
71     t.add(2, 4, 10);
72     cout << "Values: ";
73     for (int i = 0; i < t.size(); i++) {
74         cout << t.at(i) << " ";
75     }
76     cout << endl;
77     return 0;
78 }
```

2.5.3 Fenwick Tree (Point Update, Range Query)

```

1 /*
2
3 Maintain an array of numerical type, allowing for updates of individual indices
4 (point update) and queries for the sum of contiguous sub-arrays (range queries).
5 This implementation assumes that the array is 0-based (i.e. has valid indices
6 from 0 to size() - 1, inclusive).
7
8 - size() returns the size of the array.
9 - at(i) returns the value at index i.
10 - add(i, x) adds x to the value at index i.
11 - set(i, x) assigns the value at index i to x.
12 - sum(hi) returns the sum of all values at indices from 0 to hi, inclusive.
13 - sum(lo, hi) returns the sum of all values at indices from lo to hi, inclusive.
14
15 Time Complexity:
16 - O(n) per call to the constructor, where n is the size of the array.
17 - O(1) per call to size() and at().
18 - O(log n) per call to add(), set(), and both sum() functions.
19
20 Space Complexity:
21 - O(n) for storage of the array elements.
22 - O(1) auxiliary for all operations.
23
24 */
25
26 #include <vector>
27
28 template<class T>
29 class fenwick_tree {
```

```
30     int len;
31     std::vector<int> a, t;
32
33 public:
34     fenwick_tree(int n) : len(n), a(n + 1), t(n + 1) {}
35
36     int size() const {
37         return len;
38     }
39
40     T at(int i) const {
41         return a[i + 1];
42     }
43
44     void add(int i, const T &x) {
45         a[++i] += x;
46         for (; i <= len; i += i & -i) {
47             t[i] += x;
48         }
49     }
50
51     void set(int i, const T &x) {
52         T inc = x - a[i + 1];
53         add(i, inc);
54     }
55
56     T sum(int hi) {
57         T res = 0;
58         for (hi++; hi > 0; hi -= hi & -hi) {
59             res += t[hi];
60         }
61         return res;
62     }
63
64     T sum(int lo, int hi) {
65         return sum(hi) - sum(lo - 1);
66     }
67 };
68
69 /** Example Usage and Output:
70
71 Values: 5 1 2 3 4
72
73 */
74
75 #include <cassert>
76 #include <iostream>
77 using namespace std;
78
79 int main() {
80     int a[] = {10, 1, 2, 3, 4};
81     fenwick_tree<int> t(5);
82     for (int i = 0; i < 5; i++) {
83         t.set(i, a[i]);
84     }
85     t.add(0, -5);
86     cout << "Values: ";
87     for (int i = 0; i < t.size(); i++) {
88         cout << t.at(i) << " ";
```

```

89     }
90     cout << endl;
91     assert(t.sum(1, 3) == 6);
92     return 0;
93 }
```

2.5.4 Fenwick Tree (Range Update, Range Query)

```

1  /*
2
3  Maintain an array of numerical type, allowing for contiguous sub-arrays to be
4  simultaneously incremented by arbitrary values (range update) and queries for
5  the sum of contiguous sub-arrays (range query). This implementation assumes that
6  the array is 0-based (i.e. has valid indices from 0 to size() - 1, inclusive).
7
8  - size() returns the size of the array.
9  - at(i) returns the value at index i.
10 - add(i, x) increments the value at index i by x.
11 - add(lo, hi, x) adds x to the values at all indices from lo to hi, inclusive.
12 - set(i, x) assigns the value at index i to x.
13 - sum(hi) returns the sum of all values at indices from 0 to hi, inclusive.
14 - sum(lo, hi) returns the sum of all values at indices from lo to hi, inclusive.
15
16 Time Complexity:
17 - O(n) per call to the constructor, where n is the size of the array.
18 - O(1) per call to size().
19 - O(log n) per call to all other operations.
20
21 Space Complexity:
22 - O(n) for storage of the array elements.
23 - O(1) auxiliary for all operations.
24
25 */
26
27 #include <vector>
28
29 template<class T>
30 class fenwick_tree {
31     int len;
32     std::vector<T> t1, t2;
33
34     T sum(const std::vector<T> &t, int i) {
35         T res = 0;
36         for (; i != 0; i -= i & -i) {
37             res += t[i];
38         }
39         return res;
40     }
41
42     void add(std::vector<T> &t, int i, const T &x) {
43         for (; i <= len + 1; i += i & -i) {
44             t[i] += x;
45         }
46     }
47 }
```

```
48     public:
49     fenwick_tree(int n) : len(n), t1(n + 2), t2(n + 2) {}
50
51     int size() const {
52         return len;
53     }
54
55     void add(int lo, int hi, const T &x) {
56         lo++;
57         hi++;
58         add(t1, lo, x);
59         add(t1, hi + 1, -x);
60         add(t2, lo, x*(lo - 1));
61         add(t2, hi + 1, -x*hi);
62     }
63
64     void add(int i, const T &x) {
65         return add(i, i, x);
66     }
67
68     void set(int i, const T &x) {
69         add(i, x - at(i));
70     }
71
72     T sum(int hi) {
73         hi++;
74         return hi*sum(t1, hi) - sum(t2, hi);
75     }
76
77     T sum(int lo, int hi) {
78         return sum(hi) - sum(lo - 1);
79     }
80
81     T at(int i) {
82         return sum(i, i);
83     }
84 };
85
86 /**
87  Values: 15 6 7 -5 4
88 */
89
90
91 #include <cassert>
92 #include <iostream>
93 using namespace std;
94
95
96 int main() {
97     int a[] = {10, 1, 2, 3, 4};
98     fenwick_tree<int> t(5);
99     for (int i = 0; i < t.size(); i++) {
100         t.set(i, a[i]);
101     }
102     t.add(0, 2, 5);
103     t.set(3, -5);
104     cout << "Values: ";
105     for (int i = 0; i < t.size(); i++) {
106         cout << t.at(i) << " ";
```

```

107     }
108     cout << endl;
109     assert(t.sum(0, 4) == 27);
110     return 0;
111 }
```

2.5.5 Fenwick Tree (Compressed)

```

1  /*
2
3  Maintain an array of numerical type, allowing for contiguous sub-arrays to be
4  simultaneously incremented by arbitrary values (range update) and queries for
5  the sum of contiguous sub-arrays (range query). This implementation uses
6  std::map for coordinate compression, allowing for large indices to be accessed
7  with efficient space complexity. That is, all array indices from 0 to MAXN,
8  inclusive, are accessible.
9
10 - at(i) returns the value at index i.
11 - add(i, x) adds x to the value at index i.
12 - add(lo, hi, x) adds x to the values at all indices from lo to hi, inclusive.
13 - set(i, x) assigns the value at index i to x.
14 - sum(hi) returns the sum of all values at indices from 0 to hi, inclusive.
15 - sum(lo, hi) returns the sum of all values at indices from lo to hi, inclusive.
16
17 Time Complexity:
18 - O(log^2 MAXN) per call to all member functions. If std::map is replaced with
19   std::unordered_map, then the amortized running time will become O(log MAXN).
20
21 Space Complexity:
22 - O(n log MAXN) for storage of the array elements, where n is the number of
23   distinct indices that have been accessed across all of the operations so far.
24 - O(1) auxiliary for all operations.
25
26 */
27
28 #include <map>
29
30 template<class T>
31 class fenwick_tree {
32     static const int MAXN = 1000000001;
33     std::map<int, T> tmul, tadd;
34
35     void add_helper(int at, int mul, T add) {
36         for (int i = at; i <= MAXN; i |= i + 1) {
37             tmul[i] += mul;
38             tadd[i] += add;
39         }
40     }
41
42 public:
43     void add(int lo, int hi, const T &x) {
44         add_helper(lo, x, -x*(lo - 1));
45         add_helper(hi, -x, x*hi);
46     }
47 }
```

```
48 void add(int i, const T &x) {
49     return add(i, i, x);
50 }
51
52 void set(int i, const T &x) {
53     add(i, x - at(i));
54 }
55
56 T sum(int hi) {
57     T mul = 0, add = 0;
58     for (int i = hi; i >= 0; i = (i & (i + 1)) - 1) {
59         if (tmul.find(i) != tmul.end()) {
60             mul += tmul[i];
61         }
62         if (tadd.find(i) != tadd.end()) {
63             add += tadd[i];
64         }
65     }
66     return mul*hi + add;
67 }
68
69 T sum(int lo, int hi) {
70     return sum(hi) - sum(lo - 1);
71 }
72
73 T at(int i) {
74     return sum(i, i);
75 }
76 };
77
78 /**
79  * Example Usage and Output:
80  * Values: 15 6 7 -5 4
81  */
82
83
84 #include <cassert>
85 #include <iostream>
86 using namespace std;
87
88 int main() {
89     int a[] = {10, 1, 2, 3, 4};
90     fenwick_tree<int> t;
91     for (int i = 0; i < 5; i++) {
92         t.set(i, a[i]);
93     }
94     t.add(0, 2, 5);
95     t.set(3, -5);
96     cout << "Values: ";
97     for (int i = 0; i < 5; i++) {
98         cout << t.at(i) << " ";
99     }
100    cout << endl;
101    assert(t.sum(0, 4) == 27);
102    t.add(500000001, 500000010, 3);
103    t.add(500000011, 500000015, 5);
104    t.set(500000000, 10);
105    assert(t.sum(0, 1000000000) == 92);
106    return 0;
```

107 }

2.5.6 2D Fenwick Tree (Simple)

```

1  /*
2
3  Maintain a 2D array of numerical type, allowing for updates of individual cells
4  in the matrix (point update) and queries for the sum of rectangular sub-matrices
5  (range query). This implementation assumes that array dimensions are 1-based
6  (i.e. rows have valid indices from 1 to MAXR, inclusive, and columns have valid
7  indices from 1 to MAXC, inclusive).
8
9 - initialize() resets the data structure.
10 - a[r][c] stores the value at index (r, c).
11 - add(r, c, x) adds x to the value at index (r, c).
12 - set(r, c, x) assigns x to the value at index (r, c).
13 - sum(r, c) returns the sum of the rectangle with upper-left corner (1, 1) and
14   lower-right corner (r, c).
15 - sum(r1, c1, r2, c2) returns the sum of the rectangle with upper-left corner
16   (r1, c1) and lower-right corner (r2, c2).
17
18 Time Complexity:
19 - O(n*m) per call to initialize(), where n is the number of rows and m is the
20   number of columns.
21 - O(log(n)*log(m)) per call to all other operations.
22
23 Space Complexity:
24 - O(n*m) for storage of the array elements.
25 - O(1) auxiliary for all operations.
26
27 */
28
29 const int MAXR = 100, MAXC = 100;
30 int a[MAXR + 1][MAXC + 1];
31 int bits[MAXR + 1][MAXC + 1];
32
33 void initialize() {
34     for (int i = 0; i <= MAXR; i++) {
35         for (int j = 0; j <= MAXC; j++) {
36             a[i][j] = bits[i][j] = 0;
37         }
38     }
39 }
40
41 void add(int r, int c, int x) {
42     a[r][c] += x;
43     for (int i = r; i <= MAXR; i += i & -i) {
44         for (int j = c; j <= MAXC; j += j & -j) {
45             bits[i][j] += x;
46         }
47     }
48 }
49
50 void set(int r, int c, int x) {
51     add(r, c, x - a[r][c]);

```

```

52 }
53
54 int sum(int r, int c) {
55     int res = 0;
56     for (int i = r; i > 0; i -= i & -i) {
57         for (int j = c; j > 0; j -= j & -j) {
58             res += bits[i][j];
59         }
60     }
61     return res;
62 }
63
64 int sum(int r1, int c1, int r2, int c2) {
65     return sum(r2, c2) + sum(r1 - 1, c1 - 1) -
66         sum(r1 - 1, c2) - sum(r2, c1 - 1);
67 }
68
69 /** Example Usage and Output:
70
71 Values:
72 5 6 0
73 3 0 0
74 0 0 9
75
76 */
77
78 #include <cassert>
79 #include <iostream>
80 using namespace std;
81
82 int main() {
83     initialize();
84     set(1, 1, 5);
85     set(1, 2, 6);
86     set(2, 1, 7);
87     add(3, 3, 9);
88     add(2, 1, -4);
89     cout << "Values:" << endl;
90     for (int i = 1; i <= 3; i++) {
91         for (int j = 1; j <= 3; j++) {
92             cout << a[i][j] << " ";
93         }
94         cout << endl;
95     }
96     assert(sum(1, 1, 2) == 11);
97     assert(sum(1, 1, 2, 1) == 8);
98     assert(sum(1, 1, 3, 3) == 23);
99     return 0;
100 }
```

2.5.7 2D Fenwick Tree (Compressed)

```

1 /*
2
3 Maintain a 2D array of numerical type, allowing for rectangular sub-matrices to
```

```

4 be simultaneously incremented by arbitrary values (range update) and queries for
5 the sum of rectangular sub-matrices (range query). This implementation uses
6 std::map for coordinate compression, allowing for large indices to be accessed
7 with efficient space complexity. That is, rows have valid indices from 0 to
8 MAXR, inclusive, and columns have valid indices from 0 to MAXC, inclusive.
9
10 - add(r, c, x) adds x to the value at index (r, c).
11 - add(r1, c1, r2, c2, x) adds x to all indices in the rectangle with upper-left
12   corner (r1, c1) and lower-right corner (r2, c2).
13 - set(r, c, x) assigns x to the value at index (r, c).
14 - sum(r, c) returns the sum of the rectangle with upper-left corner (0, 0) and
15   lower-right corner (r, c).
16 - sum(r1, c1, r2, c2) returns the sum of the rectangle with upper-left corner
17   (r1, c1) and lower-right corner (r2, c2).
18 - at(r, c) returns the value at index (r, c).
19
20 Time Complexity:
21 - O(log^2(MAXR)*log^2(MAXC)) per call to all member functions. If std::map is
22   replaced with std::unordered_map, then the amortized running time will become
23   O(log(MAXR)*log(MAXC)).
24
25 Space Complexity:
26 - O(n*log(MAXR)*log(MAXC)) for storage of the array elements, where n is the
27   number of distinct indices that have been accessed across all of the
28   operations so far.
29 - O(1) auxiliary for all operations.
30
31 */
32
33 #include <map>
34 #include <utility>
35
36 template<class T>
37 class fenwick_tree_2d {
38     static const int MAXR = 1000000001;
39     static const int MAXC = 1000000001;
40     std::map<std::pair<int, int>, T> t1, t2, t3, t4;
41
42     template<class Map>
43     void add(Map &tree, int r, int c, const T &x) {
44         for (int i = r + 1; i <= MAXR; i += i & -i) {
45             for (int j = c + 1; j <= MAXC; j += j & -j) {
46                 tree[std::make_pair(i, j)] += x;
47             }
48         }
49     }
50
51     void add_helper(int r, int c, const T &x) {
52         add(t1, 0, 0, x);
53         add(t1, 0, c, -x);
54         add(t2, 0, c, x*c);
55         add(t1, r, 0, -x);
56         add(t3, r, 0, x*r);
57         add(t1, r, c, x);
58         add(t2, r, c, -x*c);
59         add(t3, r, c, -x*r);
60         add(t4, r, c, x*r*c);
61     }
62 }
```

```

63     public:
64     void add(int r1, int c1, int r2, int c2, const T &x) {
65         add_helper(r2 + 1, c2 + 1, x);
66         add_helper(r1, c2 + 1, -x);
67         add_helper(r2 + 1, c1, -x);
68         add_helper(r1, c1, x);
69     }
70
71     void add(int r, int c, const T &x) {
72         add(r, c, r, c, x);
73     }
74
75     void set(int r, int c, const T &x) {
76         add(r, c, x - at(r, c));
77     }
78
79     T sum(int r, int c) {
80         r++;
81         c++;
82         T s1 = 0, s2 = 0, s3 = 0, s4 = 0;
83         for (int i = r; i > 0; i -= i & -i) {
84             for (int j = c; j > 0; j -= j & -j) {
85                 const std::pair<int, int> ij(i, j);
86                 s1 += t1[ij];
87                 s2 += t2[ij];
88                 s3 += t3[ij];
89                 s4 += t4[ij];
90             }
91         }
92         return s1*r*c + s2*r + s3*c + s4;
93     }
94
95     T sum(int r1, int c1, int r2, int c2) {
96         return sum(r2, c2) + sum(r1 - 1, c1 - 1) -
97             sum(r1 - 1, c2) - sum(r2, c1 - 1);
98     }
99
100    T at(int r, int c) {
101        return sum(r, c, r, c);
102    }
103 };
104
105 /**
106  * Example Usage and Output:
107  * Values:
108  * 5 6 0
109  * 3 5 5
110  * 0 5 14
111  */
112
113
114 #include <cassert>
115 #include <iostream>
116 using namespace std;
117
118 int main() {
119     fenwick_tree_2d<int> t;
120     t.set(0, 0, 5);
121     t.set(0, 1, 6);

```

```

122     t.set(1, 0, 7);
123     t.add(2, 2, 9);
124     t.add(1, 0, -4);
125     t.add(1, 1, 2, 2, 5);
126     cout << "Values:" << endl;
127     for (int i = 0; i < 3; i++) {
128         for (int j = 0; j < 3; j++) {
129             cout << t.at(i, j) << " ";
130         }
131         cout << endl;
132     }
133     assert(t.sum(0, 0, 0, 1) == 11);
134     assert(t.sum(0, 0, 1, 0) == 8);
135     assert(t.sum(1, 1, 2, 2) == 29);
136     t.set(500000000, 500000000, 100);
137     assert(t.sum(0, 0, 1000000000, 1000000000) == 143);
138     return 0;
139 }
```

2.6 Tree Data Structures

2.6.1 Disjoint Set Forest (Simple)

```

1  /*
2
3  Maintain a set of elements partitioned into non-overlapping subsets. Each
4  partition is assigned a unique representative known as the parent, or root. The
5  following implements two well-known optimizations known as union-by-rank and
6  path compression. This version is simplified to only work on integer elements.
7
8  - initialize() resets the data structure.
9  - make_set(u) creates a new partition consisting of the single element u, which
10   must not have been previously added to the data structure.
11  - find_root(u) returns the unique representative of the partition containing u.
12  - is_united(u, v) returns whether elements u and v belong to the same partition.
13  - unite(u, v) replaces the partitions containing u and v with a single new
14   partition consisting of the union of elements in the original partitions.
15
16 A precondition to the last three operations is that make_set() must have been
17 previously called on their arguments.
18
19 Time Complexity:
20 - O(1) per call to initialize() and make_set().
21 - O(a(n)) per call to find_root(), is_united(), and unite(), where n is the
22   number of elements that has been added via make_set() so far, and a(n) is the
23   extremely slow growing inverse of the Ackermann function (effectively a very
24   small constant for all practical values of n).
25
26 Space Complexity:
27 - O(n) for storage of the disjoint set forest elements.
28 - O(1) auxiliary for all operations.
29
30 */
```

```
31 const int MAXN = 1000;
32 int num_sets, root[MAXN], rank[MAXN];
33
34 void initialize() {
35     num_sets = 0;
36 }
37
38 void make_set(int u) {
39     root[u] = u;
40     rank[u] = 0;
41     num_sets++;
42 }
43
44 int find_root(int u) {
45     if (root[u] != u) {
46         root[u] = find_root(root[u]);
47     }
48     return root[u];
49 }
50
51
52 bool is_united(int u, int v) {
53     return find_root(u) == find_root(v);
54 }
55
56 void unite(int u, int v) {
57     int ru = find_root(u), rv = find_root(v);
58     if (ru == rv) {
59         return;
60     }
61     num_sets--;
62     if (rank[ru] < rank[rv]) {
63         root[ru] = rv;
64     } else {
65         root[rv] = ru;
66         if (rank[ru] == rank[rv]) {
67             rank[ru]++;
68         }
69     }
70 }
71
72 /**
73  * Example Usage */
74 #include <cassert>
75
76 int main() {
77     initialize();
78     for (char c = 'a'; c <= 'g'; c++) {
79         make_set(c);
80     }
81     unite('a', 'b');
82     unite('b', 'f');
83     unite('d', 'e');
84     unite('d', 'g');
85     assert(num_sets == 3);
86     assert(is_united('a', 'b'));
87     assert(!is_united('a', 'c'));
88     assert(!is_united('b', 'g'));
89     assert(is_united('e', 'g'));
```

```

90     return 0;
91 }
```

2.6.2 Disjoint Set Forest (Compressed)

```

1  /*
2
3  Maintain a set of elements partitioned into non-overlapping subsets using a
4  collection of trees. Each partition is assigned a unique representative known as
5  the parent, or root. The following implements two well-known optimizations known
6  as union-by-rank and path compression. This version uses an std::map for storage
7  and coordinate compression (thus, element types must meet the requirements of
8  key types for std::map).
9
10 - make_set(u) creates a new partition consisting of the single element u, which
11   must not have been previously added to the data structure.
12 - is_united(u, v) returns whether elements u and v belong to the same partition.
13 - unite(u, v) replaces the partitions containing u and v with a single new
14   partition consisting of the union of elements in the original partitions.
15 - get_all_sets() returns all current partitions as a vector of vectors.
16
17 A precondition to the last three operations is that make_set() must have been
18 previously called on their arguments.
19
20 Time Complexity:
21 - O(1) per call to the constructor.
22 - O(log n) per call to make_set(), where n is the number of elements that have
23   been added via make_set() so far.
24 - O(a(n) log n) per call to is_united() and unite(), where n is the number of
25   elements that have been added via make_set() so far, and a(n) is the extremely
26   slow growing inverse of the Ackermann function (effectively a very small
27   constant for all practical values of n).
28 - O(n) per call to get_all_sets().
29
30 Space Complexity:
31 - O(n) for storage of the disjoint set forest elements.
32 - O(n) auxiliary heap space for get_all_sets().
33 - O(1) auxiliary for all other operations.
34 */
35
36
37 #include <map>
38 #include <vector>
39
40 template<class T>
41 class disjoint_set_forest {
42     int num_elements, num_sets;
43     std::map<T, int> id;
44     std::vector<int> root, rank;
45
46     int find_root(int u) {
47         if (root[u] != u) {
48             root[u] = find_root(root[u]);
49         }
50         return root[u];
51 }
```

```

51     }
52
53 public:
54     disjoint_set_forest() : num_elements(0), num_sets(0) {}
55
56     int size() const {
57         return num_elements;
58     }
59
60     int sets() const {
61         return num_sets;
62     }
63
64     void make_set(const T &u) {
65         if (id.find(u) != id.end()) {
66             return;
67         }
68         id[u] = num_elements;
69         root.push_back(num_elements++);
70         rank.push_back(0);
71         num_sets++;
72     }
73
74     bool is_united(const T &u, const T &v) {
75         return find_root(id[u]) == find_root(id[v]);
76     }
77
78     void unite(const T &u, const T &v) {
79         int ru = find_root(id[u]), rv = find_root(id[v]);
80         if (ru == rv) {
81             return;
82         }
83         num_sets--;
84         if (rank[ru] < rank[rv]) {
85             root[ru] = rv;
86         } else {
87             root[rv] = ru;
88             if (rank[ru] == rank[rv]) {
89                 rank[ru]++;
90             }
91         }
92     }
93
94     std::vector<std::vector<T>> get_all_sets() {
95         std::map<int, std::vector<T>> tmp;
96         for (typename std::map<T, int>::iterator it = id.begin(); it != id.end();
97             ++it) {
98             tmp[find_root(it->second)].push_back(it->first);
99         }
100        std::vector<std::vector<T>> res;
101        for (typename std::map<int, std::vector<T>>::iterator it = tmp.begin();
102            it != tmp.end(); ++it) {
103            res.push_back(it->second);
104        }
105        return res;
106    }
107 };
108
109 /** Example Usage and Output:

```

```

110 [a, b, f], [c], [d, e, g]
111 */
112
113 #include <cassert>
114 #include <iostream>
115 using namespace std;
116
117 int main() {
118     disjoint_set_forest<char> dsf;
119     for (char c = 'a'; c <= 'g'; c++) {
120         dsf.make_set(c);
121     }
122     dsf.unite('a', 'b');
123     dsf.unite('b', 'f');
124     dsf.unite('d', 'e');
125     dsf.unite('d', 'g');
126     assert(dsf.size() == 7);
127     assert(dsf.sets() == 3);
128     vector<vector<char>> s = dsf.get_all_sets();
129     for (int i = 0; i < (int)s.size(); i++) {
130         cout << (i > 0 ? ", [" : "[");
131         for (int j = 0; j < (int)s[i].size(); j++) {
132             cout << (j > 0 ? ", " : "") << s[i][j];
133         }
134         cout << "]";
135     }
136     cout << endl;
137     return 0;
138 }
139 }
```

2.6.3 Lowest Common Ancestor (Sparse Table)

```

1 /*
2
3 Given a tree, determine the lowest common ancestor of any two nodes in the tree.
4 The lowest common ancestor of two nodes u and v is the node that has the longest
5 distance from the root while having both u and v as its descendant. A nodes is
6 considered to be a descendant of itself. build() applies to a global,
7 pre-populated adjacency list adj[] which must only consist of nodes numbered
8 with integers between 0 (inclusive) and the total number of nodes (exclusive),
9 as passed in the function argument.
10
11 Time Complexity:
12 - O(n log n) per call to build(), where n is the number of nodes.
13 - O(log n) per call to lca().
14
15 Space Complexity:
16 - O(n log n) to store the sparse table, where n is the number of nodes.
17 - O(n) auxiliary stack space for build().
18 - O(1) auxiliary for lca().
19
20 */
21
```

```

22 #include <vector>
23
24 const int MAXN = 1000;
25 std::vector<int> adj[MAXN], dp[MAXN];
26 int len, timer, tin[MAXN], tout[MAXN];
27
28 void dfs(int u, int p) {
29     tin[u] = timer++;
30     dp[u][0] = p;
31     for (int i = 1; i < len; i++) {
32         dp[u][i] = dp[dp[u][i - 1]][i - 1];
33     }
34     for (int j = 0; j < (int)adj[u].size(); j++) {
35         int v = adj[u][j];
36         if (v != p) {
37             dfs(v, u);
38         }
39     }
40     tout[u] = timer++;
41 }
42
43 void build(int nodes, int root = 0) {
44     len = 1;
45     while ((1 << len) <= nodes) {
46         len++;
47     }
48     for (int i = 0; i < nodes; i++) {
49         dp[i].resize(len);
50     }
51     timer = 0;
52     dfs(root, root);
53 }
54
55 bool is_ancestor(int parent, int child) {
56     return (tin[parent] <= tin[child]) && (tout[child] <= tout[parent]);
57 }
58
59 int lca(int u, int v) {
60     if (is_ancestor(u, v)) {
61         return u;
62     }
63     if (is_ancestor(v, u)) {
64         return v;
65     }
66     for (int i = len - 1; i >= 0; i--) {
67         if (!is_ancestor(dp[u][i], v)) {
68             u = dp[u][i];
69         }
70     }
71     return dp[u][0];
72 }
73
74 /** Example Usage ***/
75
76 #include <cassert>
77 using namespace std;
78
79 int main() {
80     adj[0].push_back(1);

```

```

81     adj[1].push_back(0);
82     adj[1].push_back(2);
83     adj[2].push_back(1);
84     adj[3].push_back(1);
85     adj[1].push_back(3);
86     adj[0].push_back(4);
87     adj[4].push_back(0);
88     build(5, 0);
89     assert(lca(3, 2) == 1);
90     assert(lca(2, 4) == 0);
91     return 0;
92 }
```

2.6.4 Lowest Common Ancestor (Segment Tree)

```

1  /*
2
3 Given a tree, determine the lowest common ancestor of any two nodes in the tree.
4 The lowest common ancestor of two nodes u and v is the node that has the longest
5 distance from the root while having both u and v as its descendant. A node is
6 considered to be a descendant of itself. build() applies to a global,
7 pre-populated adjacency list adj[] which must only consist of nodes numbered
8 with integers between 0 (inclusive) and the total number of nodes (exclusive),
9 as passed in the function argument.
10
11 Time Complexity:
12 - O(n log n) per call to build(), where n is the number of nodes.
13 - O(log n) per call to lca().
14
15 Space Complexity:
16 - O(n) for storage of the segment tree, where n is the number of nodes.
17 - O(n log n) auxiliary stack space for build().
18 - O(log n) auxiliary stack space for lca().
19 */
20
21 #include <algorithm>
22 #include <vector>
23
24 const int MAXN = 1000;
25 std::vector<int> adj[MAXN];
26 int len, counter, depth[MAXN], dfs_order[2*MAXN], first[MAXN], minpos[8*MAXN];
27
28 void dfs(int u, int d) {
29     depth[u] = d;
30     dfs_order[counter++] = u;
31     for (int j = 0; j < (int)adj[u].size(); j++) {
32         int v = adj[u][j];
33         if (depth[v] == -1) {
34             dfs(v, d + 1);
35             dfs_order[counter++] = u;
36         }
37     }
38 }
39 }
```

```

41 void build(int n, int lo, int hi) {
42     if (lo == hi) {
43         minpos[n] = dfs_order[lo];
44         return;
45     }
46     int lchild = 2*n + 1, rchild = 2*n + 2, mid = lo + (hi - lo)/2;
47     build(lchild, lo, mid);
48     build(rchild, mid + 1, hi);
49     minpos[n] = depth[minpos[lchild]] < depth[minpos[rchild]] ? minpos[lchild]
50                                         : minpos[rchild];
51 }
52
53 void build(int nodes, int root) {
54     std::fill(depth, depth + nodes, -1);
55     std::fill(first, first + nodes, -1);
56     len = 2*nodes - 1;
57     counter = 0;
58     dfs(root, 0);
59     build(0, 0, len - 1);
60     for (int i = 0; i < len; i++) {
61         if (first[dfs_order[i]] == -1) {
62             first[dfs_order[i]] = i;
63         }
64     }
65 }
66
67 int get_minpos(int a, int b, int n, int lo, int hi) {
68     if (a == lo && b == hi) {
69         return minpos[n];
70     }
71     int mid = lo + (hi - lo)/2;
72     if (a <= mid && mid < b) {
73         int p1 = get_minpos(a, std::min(b, mid), 2*n + 1, lo, mid);
74         int p2 = get_minpos(std::max(a, mid + 1), b, 2*n + 2, mid + 1, hi);
75         return depth[p1] < depth[p2] ? p1 : p2;
76     }
77     if (a <= mid) {
78         return get_minpos(a, std::min(b, mid), 2*n + 1, lo, mid);
79     }
80     return get_minpos(std::max(a, mid + 1), b, 2*n + 2, mid + 1, hi);
81 }
82
83 int lca(int u, int v) {
84     return get_minpos(std::min(first[u], first[v]), std::max(first[u], first[v]),
85                       0, 0, len - 1);
86 }
87
88 /** Example Usage ***/
89
90 #include <cassert>
91 using namespace std;
92
93 int main() {
94     adj[0].push_back(1);
95     adj[1].push_back(0);
96     adj[1].push_back(2);
97     adj[2].push_back(1);
98     adj[3].push_back(1);
99     adj[1].push_back(3);

```

```

100    adj[0].push_back(4);
101    adj[4].push_back(0);
102    build(5, 0);
103    assert(lca(3, 2) == 1);
104    assert(lca(2, 4) == 0);
105    return 0;
106 }
```

2.6.5 Heavy Light Decomposition

```

1  /*
2
3  Maintain a tree with values associated with either its edges or nodes, while
4  supporting both dynamic queries and dynamic updates of all values on a given
5  path between two nodes in the tree. Heavy-light decomposition partitions the
6  nodes of the tree into disjoint paths where all nodes have degree two, except
7  the endpoints of a path which has degree one.
8
9  The query operation is defined by an associative join_values() function which
10 satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)
11 for all values x, y, and z in the tree. The default code below assumes a
12 numerical tree type, defining queries for the "min" of the target range.
13 Another possible query operation is "sum", in which case the join_values()
14 function should be defined to return "a + b".
15
16 The update operation is defined by the join_value_with_delta() and join_deltas()
17 functions, which determines the change made to values. These must satisfy:
18 - join_deltas(d1, join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).
19 - join_value_with_delta(join_values(v, ... (m times) ..., v), d, m) should be
20   equal to join_values(join_value_with_delta(v, d, 1), ... (m times)).
21 - if a sequence d_1, ..., d_m of deltas is used to update a value v, then
22   join_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent
23   to m sequential calls to join_value_with_delta(v, d_i, 1) for i = 1..m.
24 The default code below defines updates that "set" a path's edges or nodes to a
25 new value. Another possible update operation is "increment", in which case
26 join_value_with_delta(v, d, len) should be defined to return "v + d*len" and
27 join_deltas(d1, d2) should be defined to return "d1 + d2".
28
29 - heavy_light(n, adj[], v) constructs a new heavy light decomposition on a tree
30   with n nodes defined by the adjacency list adj[], with all values initialized
31   to v. The adjacency list must be a size n array of vectors consisting of only
32   the integers from 0 to n - 1, inclusive. No duplicate edges should exist, and
33   the graph must be connected.
34 - query(u, v) returns the result of join_values() applied to all values on the
35   path from node u to node v.
36 - update(u, v, d) modifies all values on the path from node u to node v by
37   respectively joining them with d using join_value_with_delta().
38
39 Time Complexity:
40 - O(n) per call to the constructor, where n is the number of nodes.
41 - O(log n) per call to query() and update();
42
43 Space Complexity:
44 - O(n) for storage of the decomposition.
45 - O(n) auxiliary stack space for the constructor.
```

```

46 - O(1) auxiliary for query() and update().
47 */
48
49
50 #include <algorithm>
51 #include <stdexcept>
52 #include <vector>
53
54 template<class T>
55 class heavy_light {
56     // Set this to true to store values on edges, false to store values on nodes.
57     static const bool VALUES_ON_EDGES = true;
58
59     static T join_values(const T &a, const T &b) {
60         return std::min(a, b);
61     }
62
63     static T join_value_with_delta(const T &v, const T &d, int len) {
64         return d;
65     }
66
67     static T join_deltas(const T &d1, const T &d2) {
68         return d2; // For "set" updates, the more recent delta prevails.
69     }
70
71     int counter, paths;
72     std::vector<std::vector<T>> value, delta;
73     std::vector<std::vector<bool>> pending;
74     std::vector<std::vector<int>> len;
75     std::vector<int> size, parent, tin, tout, path, pathlen, pathpos, pathroot;
76     std::vector<int> *adj;
77
78     void dfs(int u, int p) {
79         tin[u] = counter++;
80         parent[u] = p;
81         size[u] = 1;
82         for (int j = 0; j < (int)adj[u].size(); j++) {
83             int v = adj[u][j];
84             if (v != p) {
85                 dfs(v, u);
86                 size[u] += size[v];
87             }
88         }
89         tout[u] = counter++;
90     }
91
92     int new_path(int u) {
93         pathroot[paths] = u;
94         return paths++;
95     }
96
97     void build_paths(int u, int path) {
98         this->path[u] = path;
99         pathpos[u] = pathlen[path]++;
100        for (int j = 0; j < (int)adj[u].size(); j++) {
101            int v = adj[u][j];
102            if (v != parent[u]) {
103                build_paths(v, (2 * size[v] >= size[u]) ? path : new_path(v));
104            }
105        }
106    }

```

```

105     }
106 }
107
108 inline T join_value_with_delta(int path, int i) {
109     return pending[path][i]
110     ? join_value_with_delta(value[path][i], delta[path][i], len[path][i])
111     : value[path][i];
112 }
113
114 void push_delta(int path, int i) {
115     int d = 0;
116     while ((i >> d) > 0) {
117         d++;
118     }
119     for (d -= 2; d >= 0; d--) {
120         int l = (i >> d), r = (l ^ 1), n = l/2;
121         if (pending[path][n]) {
122             value[path][n] = join_value_with_delta(path, n);
123             delta[path][l] =
124                 pending[path][l] ? join_deltas(delta[path][l], delta[path][n])
125                 : delta[path][n];
126             delta[path][r] =
127                 pending[path][r] ? join_deltas(delta[path][r], delta[path][n])
128                 : delta[path][n];
129             pending[path][l] = pending[path][r] = true;
130             pending[path][n] = false;
131         }
132     }
133 }
134
135 bool query(int path, int u, int v, T *res) {
136     push_delta(path, u += value[path].size()/2);
137     push_delta(path, v += value[path].size()/2);
138     bool found = false;
139     for (; u <= v; u = (u + 1)/2, v = (v - 1)/2) {
140         if ((u & 1) != 0) {
141             T value = join_value_with_delta(path, u);
142             *res = found ? join_values(*res, value) : value;
143             found = true;
144         }
145         if ((v & 1) == 0) {
146             T value = join_value_with_delta(path, v);
147             *res = found ? join_values(*res, value) : value;
148             found = true;
149         }
150     }
151     return found;
152 }
153
154 void update(int path, int u, int v, const T &d) {
155     push_delta(path, u += value[path].size()/2);
156     push_delta(path, v += value[path].size()/2);
157     int tu = -1, tv = -1;
158     for (; u <= v; u = (u + 1)/2, v = (v - 1)/2) {
159         if ((u & 1) != 0) {
160             delta[path][u] = pending[path][u] ? join_deltas(delta[path][u], d) : d;
161             pending[path][u] = true;
162             if (tu == -1) {
163                 tu = u;

```

```

164         }
165     }
166     if ((v & 1) == 0) {
167         delta[path][v] = pending[path][v] ? join_deltas(delta[path][v], d) : d;
168         pending[path][v] = true;
169         if (tv == -1) {
170             tv = v;
171         }
172     }
173 }
174 for (int i = tu; i > 1; i /= 2) {
175     value[path][i/2] = join_values(join_value_with_delta(path, i),
176                                     join_value_with_delta(path, i ^ 1));
177 }
178 for (int i = tv; i > 1; i /= 2) {
179     value[path][i/2] = join_values(join_value_with_delta(path, i),
180                                     join_value_with_delta(path, i ^ 1));
181 }
182 }
183
184 inline bool is_ancestor(int parent, int child) {
185     return (tin[parent] <= tin[child]) && (tout[child] <= tout[parent]);
186 }
187
188 public:
189 heavy_light(int n, std::vector<int> adj[], const T &v = T())
190     : counter(0), paths(0), size(n), parent(n), tin(n), tout(n), path(n),
191       pathlen(n), pathpos(n), pathroot(n), adj(adj) {
192     dfs(0, -1);
193     build_paths(0, new_path(0));
194     value.resize(paths);
195     delta.resize(paths);
196     pending.resize(paths);
197     len.resize(paths);
198     for (int i = 0; i < paths; i++) {
199         int m = pathlen[i];
200         value[i].assign(2*m, v);
201         delta[i].resize(2*m);
202         pending[i].assign(2*m, false);
203         len[i].assign(2*m, 1);
204         for (int j = 2*m - 1; j > 1; j -= 2) {
205             value[i][j/2] = join_values(value[i][j], value[i][j ^ 1]);
206             len[i][j/2] = len[i][j] + len[i][j ^ 1];
207         }
208     }
209 }
210
211 T query(int u, int v) {
212     if (VALUES_ON_EDGES && u == v) {
213         throw std::runtime_error("No edge between u and v to be queried.");
214     }
215     bool found = false;
216     T res = T(), value;
217     int root;
218     while (!is_ancestor(root = pathroot[path[u]], v)) {
219         if (query(path[u], 0, pathpos[u], &value)) {
220             res = found ? join_values(res, value) : value;
221             found = true;
222         }

```

```

223     u = parent[root];
224 }
225 while (!is_ancestor(root = pathroot[path[v]], u)) {
226     if (query(path[v], 0, pathpos[v], &value)) {
227         res = found ? join_values(res, value) : value;
228         found = true;
229     }
230     v = parent[root];
231 }
232 if (query(path[u], std::min(pathpos[u], pathpos[v]) + (int)VALUES_ON_EDGES,
233             std::max(pathpos[u], pathpos[v]), &value)) {
234     res = found ? join_values(res, value) : value;
235     found = true;
236 }
237 if (!found) {
238     throw std::runtime_error("Unexpected error: No values found.");
239 }
240 return res;
241 }

242 void update(int u, int v, const T &d) {
243     if (VALUES_ON_EDGES && u == v) {
244         return;
245     }
246     int root;
247     while (!is_ancestor(root = pathroot[path[u]], v)) {
248         update(path[u], 0, pathpos[u], d);
249         u = parent[root];
250     }
251     while (!is_ancestor(root = pathroot[path[v]], u)) {
252         update(path[v], 0, pathpos[v], d);
253         v = parent[root];
254     }
255     update(path[u], std::min(pathpos[u], pathpos[v]) + (int)VALUES_ON_EDGES,
256             std::max(pathpos[u], pathpos[v]), d);
257 }
258 };
259 };

260 /**
261  * Example Usage */
262
263 #include <cassert>
264 using namespace std;
265
266 int main() {
267     //      w=40      w=20      w=10
268     // 0-----1-----2-----3
269     //                   |
270     //                   -----
271     //                           w=30
272     vector<int> adj[5];
273     adj[0].push_back(1);
274     adj[1].push_back(0);
275     adj[1].push_back(2);
276     adj[2].push_back(1);
277     adj[2].push_back(3);
278     adj[3].push_back(2);
279     adj[2].push_back(4);
280     adj[4].push_back(2);
281     heavy_light<int> hld(5, adj, 0);

```

```

282     hld.update(0, 1, 40);
283     hld.update(1, 2, 20);
284     hld.update(2, 3, 10);
285     hld.update(2, 4, 30);
286     assert(hld.query(0, 3) == 10);
287     assert(hld.query(2, 4) == 30);
288     hld.update(3, 4, 5);
289     assert(hld.query(1, 4) == 5);
290     return 0;
291 }
```

2.6.6 Link-Cut Tree

```

1  /*
2
3  Maintain a forest of trees with values associated with its nodes, while
4  supporting both dynamic queries and dynamic updates of all values on any path
5  between two nodes in a given tree. In addition, support testing of whether two
6  nodes are connected in the forest, as well as the merging and splitting of trees
7  by adding or removing specific edges. Link/cut forests divide each of its trees
8  into vertex-disjoint paths, each represented by a splay tree.
9
10 The query operation is defined by an associative join_values() function which
11 satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)
12 for all values x, y, and z in the forest. The default code below assumes a
13 numerical forest type, defining queries for the "min" of the target range.
14 Another possible query operation is "sum", in which case the join_values()
15 function should be defined to return "a + b".
16
17 The update operation is defined by the join_value_with_delta() and join_deltas()
18 functions, which determines the change made to values. These must satisfy:
19 - join_deltas(d1, join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).
20 - join_value_with_delta(join_values(v, ... (m times) ..., v), d, m) should be
21   equal to join_values(join_value_with_delta(v, d, 1), ... (m times)).
22 - if a sequence d_1, ..., d_m of deltas is used to update a value v, then
23   join_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent
24   to m sequential calls to join_value_with_delta(v, d_i, 1) for i = 1..m.
25 The default code below defines updates that "set" a path's nodes to a new value.
26 Another possible update operation is "increment", in which case
27 join_value_with_delta(v, d, len) should be defined to return "v + d*len" and
28 join_deltas(d1, d2) should be defined to return "d1 + d2".
29
30 - link_cut_forest() constructs an empty forest with no trees.
31 - size() returns the number of nodes in the forest.
32 - trees() returns the number of trees in the forest.
33 - make_root(i, v) creates a new tree in the forest consisting of a single node
34   labeled with the integer i and value initialized to v.
35 - is_connected(a, b) returns whether nodes a and b are connected.
36 - link(a, b) adds an edge between the nodes a and b, both of which must exist
37   and not be connected.
38 - cut(a, b) removes the edge between the nodes a and b, both of which must
39   exist and be connected.
40 - query(a, b) returns the result of join_values() applied to all values on the
41   path from the node a to node b.
42 - update(a, b, d) modifies all the values on the path from node a to node b by
```

```

43     respectively joining them with d using join_value_with_delta().
44
45 Time Complexity:
46 - O(1) per call to the constructor, size(), and trees().
47 - O(log n) amortized per call to all other operations, where n is the number of
48 nodes.
49
50 Space Complexity:
51 - O(n) for storage of the forest, where n is the number of nodes.
52 - O(1) auxiliary for all operations.
53
54 */
55
56 #include <algorithm>
57 #include <cstddef>
58 #include <map>
59 #include <stdexcept>
60
61 template<class T>
62 class link_cut_forest {
63     static T join_values(const T &a, const T &b) {
64         return std::min(a, b);
65     }
66
67     static T join_value_with_delta(const T &v, const T &d, int len) {
68         return d;
69     }
70
71     static T join_deltas(const T &d1, const T &d2) {
72         return d2; // For "set" updates, the more recent delta prevails.
73     }
74
75     struct node_t {
76         T value, subtree_value, delta;
77         int size;
78         bool rev, pending;
79         node_t *left, *right, *parent;
80
81         node_t(const T &v)
82             : value(v), subtree_value(v), size(1), rev(false), pending(false),
83               left(NULL), right(NULL), parent(NULL) {}
84
85         inline bool is_root() const {
86             return parent == NULL || (parent->left != this && parent->right != this);
87         }
88
89         inline T get_subtree_value() const {
90             return pending ? join_value_with_delta(subtree_value, delta, size)
91                           : subtree_value;
92         }
93
94         void push() {
95             if (rev) {
96                 rev = false;
97                 std::swap(left, right);
98                 if (left != NULL) {
99                     left->rev = !left->rev;
100                }
101                if (right != NULL) {

```

```

102         right->rev = !right->rev;
103     }
104 }
105 if (pending) {
106     value = join_value_with_delta(value, delta, 1);
107     subtree_value = join_value_with_delta(subtree_value, delta, size);
108     if (left != NULL) {
109         left->delta = left->pending ? join_deltas(left->delta, delta) : delta;
110         left->pending = true;
111     }
112     if (right != NULL) {
113         right->delta = right->pending ? join_deltas(right->delta, delta)
114                                         : delta;
115         right->pending = true;
116     }
117     pending = false;
118 }
119 }
120
121 void update() {
122     size = 1;
123     subtree_value = value;
124     if (left != NULL) {
125         subtree_value = join_values(subtree_value, left->get_subtree_value());
126         size += left->size;
127     }
128     if (right != NULL) {
129         subtree_value = join_values(subtree_value, right->get_subtree_value());
130         size += right->size;
131     }
132 }
133 };
134
135 int num_trees;
136 std::map<int, node_t*> nodes;
137
138 static void connect(node_t *child, node_t *parent, bool is_left) {
139     if (child != NULL) {
140         child->parent = parent;
141     }
142     if (is_left) {
143         parent->left = child;
144     } else {
145         parent->right = child;
146     }
147 }
148
149 static void rotate(node_t *n) {
150     node_t *parent = n->parent, *grandparent = parent->parent;
151     bool parent_is_root = parent->is_root(), is_left = (n == parent->left);
152     connect(is_left ? n->right : n->left, parent, is_left);
153     connect(parent, n, !is_left);
154     if (parent_is_root) {
155         if (n != NULL) {
156             n->parent = grandparent;
157         }
158     } else {
159         connect(n, grandparent, parent == grandparent->left);
160     }

```

```

161     parent->update();
162 }
163
164 static void splay(node_t *n) {
165     while (!n->is_root()) {
166         node_t *parent = n->parent, *grandparent = parent->parent;
167         if (!parent->is_root()) {
168             grandparent->push();
169         }
170         parent->push();
171         n->push();
172         if (!parent->is_root()) {
173             if ((n == parent->left) == (parent == grandparent->left)) {
174                 rotate(parent);
175             } else {
176                 rotate(n);
177             }
178         }
179         rotate(n);
180     }
181     n->push();
182     n->update();
183 }
184
185 static node_t* expose(node_t *n) {
186     node_t *prev = NULL;
187     for (node_t *curr = n; curr != NULL; curr = curr->parent) {
188         splay(curr);
189         curr->left = prev;
190         prev = curr;
191     }
192     splay(n);
193     return prev;
194 }
195
196 // Helper variables.
197 node_t *u, *v;
198
199 void get_uv(int a, int b) {
200     typename std::map<int, node_t*>::iterator it1, it2;
201     it1 = nodes.find(a);
202     it2 = nodes.find(b);
203     if (it1 == nodes.end() || it2 == nodes.end()) {
204         throw std::runtime_error("Queried node ID does not exist in forest.");
205     }
206     u = it1->second;
207     v = it2->second;
208 }
209
210 public:
211     link_cut_forest() : num_trees(0) {}
212
213     ~link_cut_forest() {
214         typename std::map<int, node_t*>::iterator it;
215         for (it = nodes.begin(); it != nodes.end(); ++it) {
216             delete it->second;
217         }
218     }
219

```

```

220     int size() const {
221         return nodes.size();
222     }
223
224     int trees() const {
225         return num_trees;
226     }
227
228     void make_root(int i, const T &v = T()) {
229         if (nodes.find(i) != nodes.end()) {
230             throw std::runtime_error("Cannot make a root with an existing ID.");
231         }
232         node_t *n = new node_t(v);
233         expose(n);
234         n->rev = !n->rev;
235         nodes[i] = n;
236         num_trees++;
237     }
238
239     bool is_connected(int a, int b) {
240         get_uv(a, b);
241         if (a == b) {
242             return true;
243         }
244         expose(u);
245         expose(v);
246         return u->parent != NULL;
247     }
248
249     void link(int a, int b) {
250         if (is_connected(a, b)) {
251             throw std::runtime_error("Cannot link nodes that are already connected.");
252         }
253         get_uv(a, b);
254         expose(u);
255         u->rev = !u->rev;
256         u->parent = v;
257         num_trees--;
258     }
259
260     void cut(int a, int b) {
261         get_uv(a, b);
262         expose(u);
263         u->rev = !u->rev;
264         expose(v);
265         if (v->right != u || u->left != NULL) {
266             throw std::runtime_error("Cannot cut edge that does not exist.");
267         }
268         v->right->parent = NULL;
269         v->right = NULL;
270         num_trees++;
271     }
272
273     T query(int a, int b) {
274         if (!is_connected(a, b)) {
275             throw std::runtime_error("Cannot query nodes that are not connected.");
276         }
277         get_uv(a, b);
278         expose(u);

```

```

279     u->rev = !u->rev;
280     expose(v);
281     return v->get_subtree_value();
282 }
283
284 void update(int a, int b, const T &d) {
285     if (!is_connected(a, b)) {
286         throw std::runtime_error("Cannot update nodes that are not connected.");
287     }
288     get_uv(a, b);
289     expose(u);
290     u->rev = !u->rev;
291     expose(v);
292     v->delta = v->pending ? join_deltas(v->delta, d) : d;
293     v->pending = true;
294 }
295 };
296
297 /** Example Usage ***/
298
299 #include <cassert>
300 using namespace std;
301
302 int main() {
303     link_cut_forest<int> lcf;
304     lcf.make_root(0, 10);
305     lcf.make_root(1, 40);
306     lcf.make_root(2, 20);
307     lcf.make_root(3, 10);
308     lcf.make_root(4, 30);
309     assert(lcf.size() == 5);
310     assert(lcf.trees() == 5);
311     lcf.link(0, 1);
312     lcf.link(1, 2);
313     lcf.link(2, 3);
314     lcf.link(2, 4);
315     assert(lcf.trees() == 1);
316
317     // v=10      v=40      v=20      v=10
318     // 0-----1-----2-----3
319     //                   |
320     //                   -----
321     //                           v=30
322     assert(lcf.query(1, 4) == 20);
323     lcf.update(1, 1, 100);
324     lcf.update(2, 4, 100);
325
326     // v=10      v=100     v=100      v=10
327     // 0-----1-----2-----3
328     //                   |
329     //                   -----
330     //                           v=100
331     assert(lcf.query(4, 4) == 100);
332     assert(lcf.query(0, 4) == 10);
333     assert(lcf.query(3, 4) == 10);
334     lcf.cut(1, 2);
335
336     // v=10      v=100     v=100      v=0
337     // 0-----1           2-----3

```

```
338 // |  
339 // -----4  
340 // v=100  
341 assert(lcf.trees() == 2);  
342 assert(!lcf.is_connected(1, 2));  
343 assert(!lcf.is_connected(0, 4));  
344 assert(lcf.is_connected(2, 3));  
345 return 0;  
346 }
```

Chapter 3

Strings

3.1 String Utilities

```
1  /*
2
3 Common string functions, many of which are substitutes for features which are
4 not available in standard C++, or may not be available on compilers that do not
5 support C++11 and later. These operations are naive implementations and often
6 depend on certain std::string functions that have unspecified complexity.
7
8 */
9
10 #include <cctype>
11 #include <sstream>
12 #include <stdexcept>
13 #include <string>
14 #include <vector>
15 using std::string;
16
17 /*
18 Integer Conversion
19
20 - to_str(i) returns the string representation of integer i, much like
21   std::to_string() in C++11 and later.
22 - to_int(s) returns the integer representation of string s, much like atoi(),
23   except handling special cases of overflow by throwing an exception.
24 - itoa(value, &str, base) implements the non-standard C function which converts
25   value into a C string, storing it into pointer str in the given base. For more
26   generalized base conversion, see the math utilities section.
27
28 */
29
30
31 template<class Int>
32 string to_str(Int i) {
33     std::ostringstream oss;
34     oss << i;
35     return oss.str();
36 }
```

```
37
38 int to_int(const string &s) {
39     std::istringstream iss(s);
40     int res;
41     if (!(iss >> res)) {
42         throw std::runtime_error("to_int failed");
43     }
44     return res;
45 }
46
47 char* itoa(int value, char *str, int base = 10) {
48     if (base < 2 || base > 36) {
49         *str = '\0';
50         return str;
51     }
52     char *ptr = str, *ptr1 = str, tmp_c;
53     int tmp_v;
54     do {
55         tmp_v = value;
56         value /= base;
57         *ptr++ = "zyxwvutsrqponmlkjihgfedcba9876543210123456789"
58             "abcdefghijklmnopqrstuvwxyz"[35 + (tmp_v - value * base)];
59     } while (value);
60     if (tmp_v < 0) {
61         *ptr++ = '-';
62     }
63     for (*ptr-- = '\0'; ptr1 < ptr; *ptr1++ = tmp_c) {
64         tmp_c = *ptr;
65         *ptr-- = *ptr1;
66     }
67     return str;
68 }
69 /*
70 Case Conversion
71
72 - to_upper(s) returns s with all alphabetical characters converted to uppercase.
73 - to_lower(s) returns s with all alphabetical characters converted to lowercase.
74 - to_title(s) returns the title case representation of string s, where the first
75   letter of every word (consecutive alphabetical characters) is capitalized.
76
77 */
78
79 string to_upper(const string &s) {
80     string res;
81     for (int i = 0; i < (int)s.size(); i++) {
82         res.push_back(toupper(s[i]));
83     }
84     return res;
85 }
86
87 string to_lower(const string &s) {
88     string res;
89     for (int i = 0; i < (int)s.size(); i++) {
90         res.push_back(tolower(s[i]));
91     }
92     return res;
93 }
94 }
```

```

96
97 string to_title(const string &s) {
98     string res;
99     char prev = '\0';
100    for (int i = 0; i < (int)s.size(); i++) {
101        if (isalpha(prev)) {
102            res.push_back(tolower(s[i]));
103        } else {
104            res.push_back(toupper(s[i]));
105        }
106        prev = res.back();
107    }
108    return res;
109 }
110
111 /*
112
113 Stripping
114
115 - lstrip(s) strips the left side of s in-place (that is, the input is modified)
116   using the given delimiters and returns a reference to the stripped string.
117 - rstrip(s) strips the right side of s in-place using the given delimiters and
118   returns a reference to the stripped string.
119 - strip(s) strips both sides of s in-place and returns a reference to the
120   stripped string.
121 */
122
123
124 string& lstrip(string &s, const string &delim = " \n\t\v\f\r") {
125     size_t pos = s.find_first_not_of(delim);
126     if (pos != string::npos) {
127         s.erase(0, pos);
128     }
129     return s;
130 }
131
132 string& rstrip(string &s, const string &delim = " \n\t\v\f\r") {
133     size_t pos = s.find_last_not_of(delim);
134     if (pos != string::npos) {
135         s.erase(pos);
136     }
137     return s;
138 }
139
140 string& strip(string &s, const string &delim = " \n\t\v\f\r") {
141     return lstrip(rstrip(s));
142 }
143
144 /*
145
146 Find and Replace
147
148 - find_all(haystack, needle) returns a vector of all positions where the string
149   needle appears in the string haystack.
150 - replace(s, old, replacement) returns a copy of s with all occurrences of the
151   string old replaced with the given replacement.
152
153 */
154

```

```
155 std::vector<int> find_all(const string &haystack, const string &needle) {
156     std::vector<int> res;
157     size_t pos = haystack.find(needle, 0);
158     while (pos != string::npos) {
159         res.push_back(pos);
160         pos = haystack.find(needle, pos + 1);
161     }
162     return res;
163 }
164
165 string replace(const string &s, const string &old, const string &replacement) {
166     if (old.empty()) {
167         return s;
168     }
169     string res(s);
170     size_t pos = 0;
171     while ((pos = res.find(old, pos)) != string::npos) {
172         res.replace(pos, old.length(), replacement);
173         pos += replacement.length();
174     }
175     return res;
176 }
177
178 /*
179 Joining and Splitting
180
181 - join(v, delim) returns the strings in vector v concatenated, separated by the
182   given delimiter.
183 - split(s, char delim) returns a vector of tokens of s, split on a single
184   character delimiter. Note that this version will not skip empty tokens. For
185   example, split("a::b", ":") returns {"a", "b"}, not {"a", "", "b"}.
186 - split(s, string delim) returns a vector of tokens of s, split on a set of many
187   possible single character delimiters. All characters ofz delim will be removed
188   from s, and the remaining token(s) of s will be added sequentially to a vector
189   and returned. Unlike the first version, empty tokens are skipped. For example,
190   split("a::b", ":") returns {"a", "b"}, not {"a", "", "b"}.
191 - explode(s, delim) returns a vector of tokens of s, split on the entire
192   delimiter string delim. Unlike the split() functions above, delim is treated
193   as a contiguous boundary string, not merely a set of possible boundary
194   characters. This will not skip empty tokens. For example,
195   explode("a::::b", "::") yields {"a", "", "b"}, not {"a", "b"}.
196
197 */
198
199 string join(const std::vector<string> &v, const string &delim = " ") {
200     string res;
201     for (int i = 0; i < (int)v.size(); i++) {
202         if (i > 0) {
203             res += delim;
204         }
205         res += v[i];
206     }
207     return res;
208 }
209
210
211 std::vector<string> split(const string &s, char delim) {
212     std::vector<string> res;
213     std::stringstream ss(s);
```

```

214     string curr;
215     while (std::getline(ss, curr, delim)) {
216         res.push_back(curr);
217     }
218     return res;
219 }
220
221 std::vector<string> split(const string &s,
222                             const string &delim = " \n\t\v\f\r") {
223     std::vector<string> res;
224     string curr;
225     for (int i = 0; i < (int)s.size(); i++) {
226         if (delim.find(s[i]) == string::npos) {
227             curr += s[i];
228         } else if (!curr.empty()) {
229             res.push_back(curr);
230             curr = "";
231         }
232     }
233     if (!curr.empty()) {
234         res.push_back(curr);
235     }
236     return res;
237 }
238
239 std::vector<string> explode(const string &s, const string &delim) {
240     std::vector<string> res;
241     size_t last = 0, next = 0;
242     while ((next = s.find(delim, last)) != string::npos) {
243         res.push_back(s.substr(last, (int)next - last));
244         last = next + delim.size();
245     }
246     res.push_back(s.substr(last));
247     return res;
248 }
249
250 /*** Example Usage ***/
251
252 #include <cassert>
253 using namespace std;
254
255 int main() {
256     assert(to_str(123) + "4" == "1234");
257     assert(to_int("1234") == 1234);
258     char buffer[50];
259     assert(string(itoa(1750, buffer, 10)) == "1750");
260     assert(string(itoa(1750, buffer, 16)) == "6d6");
261     assert(string(itoa(1750, buffer, 2)) == "11011010110");
262
263     assert(to_upper("Hello world") == "HELLO WORLD");
264     assert(to_lower("Hello World") == "hello world");
265     assert(to_title("hello world") == "Hello World");
266
267     string s("    abc \n");
268     string t = s;
269     assert(lstrip(s) == "abc \n");
270     assert.rstrip(s) == strip(t));
271
272     vector<int> pos;

```

```

273     pos.push_back(0);
274     pos.push_back(7);
275     assert(find_all("abracadabra", "ab") == pos);
276     assert(replace("abcdabba", "ab", "00") == "00cd00ba");
277
278     assert(join(split("a\nb\ncde\nf", '\n'), "|") == "a|b|cde|f"); // split v1
279     assert(join(split("a::b,cde:,f", ":"), "|") == "a|b|cde|f"); // split v2
280     assert(join(explode("a..b.cde....f", ".."), "|") == "a|b.cde||f");
281
282     return 0;
283 }
```

3.2 Expression Parsing

3.2.1 String Searching (KMP)

```

1  /*
2
3 Given a single string (needle) and subsequent queries of texts (haystacks) to be
4 searched, determine the first positions in which the needle occurs within the
5 given haystacks in linear time using the Knuth-Morris-Pratt algorithm. In
6 comparison, std::string::find runs in quadratic time.
7
8 - kmp(needle) constructs the partial match table for a string needle that is to
9 be searched for subsequently in haystack queries.
10 - find_in(haystack) returns the first position that needle occurs in haystack,
11 or std::string::npos if it cannot be found. Note that the function can be
12 modified to return all matches by simply letting the loop run and storing
13 the results instead of returning early.
14
15 Time Complexity:
16 - O(m) per call to the constructor, where m is the length of needle.
17 - O(n) per call to find_in(haystack), where n is the length of haystack.
18
19 Space Complexity:
20 - O(m) for storage of the partial match table, where m is the length of needle.
21 - O(1) auxiliary space per call to find_in(haystack).
22 */
23
24
25 #include <string>
26 #include <vector>
27 using std::string;
28
29 class kmp {
30     string needle;
31     std::vector<int> table;
32
33 public:
34     kmp(const string &needle) : needle(needle) {
35         table.resize(needle.size());
36         int i = 0, j = table[0] = -1;
37         while (i < (int)needle.size()) {
38             while (j >= 0 && needle[i] != needle[j]) {
```

```

39         j = table[j];
40     }
41     i++;
42     j++;
43     table[i] = (needle[i] == needle[j]) ? table[j] : j;
44 }
45 }
46
47 size_t find_in(const string &haystack) {
48     if (needle.empty()) {
49         return 0;
50     }
51     for (int i = 0, j = 0; j < (int)haystack.size(); ) {
52         while (i >= 0 && needle[i] != haystack[j]) {
53             i = table[i];
54         }
55         i++;
56         j++;
57         if (i >= (int)needle.size()) {
58             return j - i;
59         }
60     }
61     return string::npos;
62 }
63 };
64
65 /** Example Usage ***/
66
67 #include <cassert>
68
69 int main() {
70     assert(15 == kmp("ABCDABD").find_in("ABC ABCDAB ABCDABCDABDE"));
71     return 0;
72 }
```

3.2.2 String Searching (Z Algorithm)

```

1 /*
2
3 Given a single string (needle) and a single text (haystack) to be searched,
4 determine the first position in which the needle occurs within the haystack in
5 linear time using the Z algorithm. In comparison, std::string::find runs in
6 quadratic time.
7
8 The find function below calls the Z algorithm on the concatenation of the needle
9 and the haystack, separated by a sentinel character (in this case '\0'), which
10 should be chosen such that it does not occur within either of the input strings.
11
12 - z_array(s) constructs the Z array for a string needle that can be used for
13   string searching. The Z array on an input string s is an array z where z[i] is
14   the length of the longest substring starting from s[i] which is also a prefix
15   of s.
16 - find(needle, haystack) returns the first position that needle occurs in
17   haystack, or std::string::npos if it cannot be found. Note that the function
18   can be modified to return all matches by simply letting the loop run and
```

```
19     storing the results instead of returning early.
20
21 Time Complexity:
22 - O(n) per call to z_array(s), where n is the length of s.
23 - O(n + m) per call to find(haystack, needle), where n is the length of haystack
24 and m is the length of needle.
25
26 Space Complexity:
27 - O(n) auxiliary heap space for z_array(s), where n is the length of s.
28 - O(n + m) auxiliary heap space for find(haystack, needle) where n is the length
29 of haystack and m is the length of needle.
30
31 */
32
33 #include <algorithm>
34 #include <string>
35 #include <vector>
36 using std::string;
37
38 std::vector<int> z_array(const string &s) {
39     std::vector<int> z(s.size());
40     for (int i = 1, l = 0, r = 0; i < (int)z.size(); i++) {
41         if (i <= r) {
42             z[i] = std::min(r - i + 1, z[i - 1]);
43         }
44         while (i + z[i] < (int)z.size() && s[z[i]] == s[i + z[i]]) {
45             z[i]++;
46         }
47         if (r < i + z[i] - 1) {
48             l = i;
49             r = i + z[i] - 1;
50         }
51     }
52     return z;
53 }
54
55 size_t find(const string &haystack, const string &needle) {
56     std::vector<int> z = z_array(needle + '\0' + haystack);
57     for (int i = (int)needle.size() + 1; i < (int)z.size(); i++) {
58         if (z[i] == (int)needle.size()) {
59             return i - (int)needle.size() - 1;
60         }
61     }
62     return string::npos;
63 }
64
65 /** Example Usage ***/
66
67 #include <cassert>
68
69 int main() {
70     assert(15 == find("ABC ABCDAB ABCDABCDABDE", "ABCDABD"));
71     return 0;
72 }
```

3.2.3 String Searching (Aho-Corasick)

```

1  /*
2
3 Given a set of strings (needles) and subsequent queries of texts (haystacks)
4 to be searched, determine all positions in which needles occur within the given
5 haystacks in linear time using the Aho-Corasick algorithm.
6
7 Note that this implementation uses an ordered map for storage of the graph,
8 adding an additional log k factor to the time complexities of all operations,
9 where k is the size of the alphabet (number of distinct characters used across
10 the needles). It also uses an ordered set for storage of the precomputed output
11 tables, adding an additional log m factor to the time complexities, where m is
12 the number of needles. In C++11 and later, both of these containers should be
13 replaced by their unordered versions for constant time access, thus eliminating
14 the log factors from the time complexities.
15
16 - aho_corasick(needles) constructs the finite-state automaton for a set of
17   needle strings that are to be searched for subsequently in haystack queries.
18 - find_all_in(haystack, report_match) calls the function report_match(s, pos)
19   once on each occurrence of each needle that occurs in the haystack, where pos
20   is the starting position in the haystack at which string s (a matched needle)
21   occurs. The matches will be reported in increasing order of their ending
22   positions within the haystack.
23
24 Time Complexity:
25 - O(m*((log m) + l*log k)) per call to the constructor, where m is the number of
26   needles, l is the maximum length for any needle, and k is the size of the
27   alphabet used by the needles. If unordered containers are used, then the time
28   complexity reduces to O(m*l), or linear on the input size.
29 - O(n*(log k) + z) per call to find_all_in(haystack, report_match), where n is
30   the length of haystack, k is the size of the alphabet used by the needles, and
31   z is the number of matches. If unordered containers are used, then the time
32   complexity reduces to O(n + z), or linear on the input size.
33
34 Space Complexity:
35 - O(m*l) for storage of the automaton, where where m is the number of needles
36   and l is the maximum length for any needle.
37 - O(1) auxiliary space per call to find_all_in(haystack, report_match).
38 */
39
40 #include <map>
41 #include <queue>
42 #include <set>
43 #include <string>
44 #include <vector>
45 using std::string;
46
47 class aho_corasick {
48     std::vector<string> needles;
49     std::vector<int> fail;
50     std::vector<std::map<char, int>> graph;
51     std::vector<std::set<int>> out;
52
53     int next_state(int curr, char c) {
54         int next = curr;
55

```

```

56     while (graph[next].find(c) == graph[next].end()) {
57         next = fail[next];
58     }
59     return graph[next][c];
60 }
61
62 public:
63     aho_corasick(const std::vector<string> &needles) : needles(needles) {
64         int total_len = 0;
65         for (int i = 0; i < (int)needles.size(); i++) {
66             total_len += needles[i].size();
67         }
68         fail.resize(total_len, -1);
69         graph.resize(total_len);
70         out.resize(total_len);
71         int states = 1;
72         std::map<char, int>::iterator it;
73         for (int i = 0; i < (int)needles.size(); i++) {
74             int curr = 0;
75             for (int j = 0; j < (int)needles[i].size(); j++) {
76                 char c = needles[i][j];
77                 if ((it = graph[curr].find(c)) != graph[curr].end()) {
78                     curr = it->second;
79                 } else {
80                     curr = graph[curr][c] = states++;
81                 }
82             }
83             out[curr].insert(i);
84         }
85         std::queue<int> q;
86         for (it = graph[0].begin(); it != graph[0].end(); ++it) {
87             if (it->second != 0) {
88                 fail[it->second] = 0;
89                 q.push(it->second);
90             }
91         }
92         while (!q.empty()) {
93             int u = q.front();
94             q.pop();
95             for (it = graph[u].begin(); it != graph[u].end(); ++it) {
96                 int v = it->second, f = fail[u];
97                 while (graph[f].find(it->first) == graph[f].end()) {
98                     f = fail[f];
99                 }
100                f = graph[f].find(it->first)->second;
101                fail[v] = f;
102                out[v].insert(out[f].begin(), out[f].end());
103                q.push(v);
104            }
105        }
106    }
107
108    template<class ReportFunction>
109    void find_all_in(const string &haystack, ReportFunction report_match) {
110        int state = 0;
111        std::set<int>::iterator it;
112        for (int i = 0; i < (int)haystack.size(); i++) {
113            state = next_state(state, haystack[i]);
114            for (it = out[state].begin(); it != out[state].end(); ++it) {

```

```

115         report_match(needles[*it], i - needles[*it].size() + 1);
116     }
117 }
118 }
119 };
120
121 /** Example Usage and Output:
122
123 Matched "a" at position 0.
124 Matched "ab" at position 0.
125 Matched "bc" at position 1.
126 Matched "c" at position 2.
127 Matched "c" at position 3.
128 Matched "a" at position 4.
129 Matched "ab" at position 4.
130 Matched "abccab" at position 0.
131
132 ***
133
134 #include <iostream>
135 using namespace std;
136
137 void report_match(const string &needle, int pos) {
138     cout << "Matched \""
139         << needle << "\" at position "
140         << pos << "."
141         << endl;
142 }
143
144 int main() {
145     vector<string> needles;
146     needles.push_back("a");
147     needles.push_back("ab");
148     needles.push_back("bab");
149     needles.push_back("bc");
150     needles.push_back("bca");
151     needles.push_back("c");
152     needles.push_back("caa");
153     needles.push_back("abccab");
154
155     aho_corasick(needles).find_all_in("abccab", report_match);
156     return 0;
157 }
```

3.3 String Searching

3.3.1 Recursive Descent Parsing (Simple)

```

1 /*
2
3 Evaluate an expression in accordance to the order of operations (parentheses,
4 unary plus and minus signs, multiplication/division, addition/subtraction). The
5 following is a minimalistic recursive descent implementation using iterators.
6
7 - eval(s) returns an evaluation of the arithmetic expression s.
8
```

```

9 Time Complexity:
10 - O(n) per call to eval(s), where n is the length of s.
11
12 Space Complexity:
13 - O(n) auxiliary stack space for eval(s), where n is the length of s.
14
15 */
16
17 #include <string>
18
19 template<class It>
20 int eval(It &it, int prec) {
21     if (prec == 0) {
22         int sign = 1, ret = 0;
23         for (; *it == '-' ; it++) {
24             sign *= -1;
25         }
26         if (*it == '(') {
27             ret = eval(++it, 2);
28             it++;
29         } else while (*it >= '0' && *it <= '9') {
30             ret = 10*ret + (*(it++) - '0');
31         }
32         return sign*ret;
33     }
34     int num = eval(it, prec - 1);
35     while (!((prec == 2 && *it != '+' && *it != '-') ||
36               (prec == 1 && *it != '*' && *it != '/'))) {
37         switch ((*it++)) {
38             case '+': num += eval(it, prec - 1); break;
39             case '-': num -= eval(it, prec - 1); break;
40             case '*': num *= eval(it, prec - 1); break;
41             case '/': num /= eval(it, prec - 1); break;
42         }
43     }
44     return num;
45 }
46
47 int eval(const std::string &s) {
48     std::string::iterator it = std::string(s).begin();
49     return eval(it, 2);
50 }
51
52 /** Example Usage ***/
53
54 #include <cassert>
55
56 int main() {
57     assert(eval("1+1") == 2);
58     assert(eval("1+2*3*4+3*(2+2)-100") == -63);
59     return 0;
60 }
```

3.3.2 Recursive Descent Parsing (Generic)

```

1  /*
2
3 Evaluate an expression using a generalized parser class for custom-defined
4 operand types, prefix unary operators, binary operators, and precedences.
5 Typical parentheses behavior is supported, but multiplication by juxtaposition
6 is not. Evaluation is performed using the recursive descent algorithm.
7
8 An arbitrary operand type is supported, with its string representation defined
9 by a user-specified is_operand() and eval_operand() functions. For maximum
10 reliability, the string representation of operands should not use characters
11 shared by any operator. For instance, the best practice instead of accepting
12 "-1" as a valid operand (since the "-" sign may conflict with the identical
13 binary operator), is to specify non-negative number as operands alongside the
14 unary operator "-".
15
16 Operators may be non-empty strings of any length, but should not contain any
17 parentheses or shared characters with the string representations of operands.
18 Ideally, operators should not be prefixes or suffixes of one another, else the
19 tokenization process may be ambiguous. For example, if ++ and + are both
20 operators, then ++ may be split into either ["+", "+"] or ["++"] depending on
21 the lexicographical ordering of conflicting operators.
22
23 - parser(unary_op, binary_op) initializes a parser with operators specified by
24 maps unary_op (of operator to function pointer) and binary_op (of operator to
25 pair of function pointer and operator precedence). Operator precedences should
26 be numbered upwards starting at 0 (lowest precedence, evaluated last).
27 - split(s) returns a vector of tokens for the expression s, split on the given
28 operators during construction. Each parenthesis, operator, and operand
29 satisfying is_operand() will be split into a separate token. The algorithm is
30 naive, matching operators lazily in the case of overlapping operators as
31 mentioned above. Under these circumstances, the parse may not always succeed.
32 - eval(lo, hi) returns the evaluation of a range [lo, hi] of already split-up
33 expression tokens, where lo and hi must be random-access iterators.
34 - eval(s) returns the evaluation of expression s, after first calling split(s)
35 to obtain the tokens.
36
37 Time Complexity:
38 - O(m) per call to the constructor, where m is the total number of operators.
39 - O(nmk) per call to split(s), where n is the length of s, m is the total number
40 of operators defined for the parser instance, and k is the maximum length for
41 any operator representation.
42 - O(n log m) per call to eval(lo, hi), where n is the distance between lo and hi
43 and m is the total number of operators defined for the parser instance. In
44 C++11 and later, std::unordered_map may be used in place of std::map for
45 storing the unary_ops and binary_ops, which will eliminate the log m factor
46 for a time complexity of O(n) per call.
47 - O(nmk + n log m) per call to eval(s), where n is the distance between lo and
48 hi, and m and k are as defined previous.
49
50 Space Complexity:
51 - O(mk) for storage of the m operators, of maximum length k.
52 - O(n) auxiliary stack space for split(s), eval(lo, hi), and eval(s), where n is
53 the length of the argument.
54 */
55
56
57 #include <algorithm>
58 #include <cctype>
59 #include <functional>
```

```

60 #include <map>
61 #include <set>
62 #include <sstream>
63 #include <stdexcept>
64 #include <string>
65 #include <utility>
66 #include <vector>
67 using std::string;
68
69 // Define the custom operand type and representation below.
70 typedef double Operand;
71 typedef Operand (*UnaryOp)(Operand a);
72 typedef Operand (*BinaryOp)(Operand a, Operand b);
73
74 bool is_operand(const string &s) {
75     int npoints = 0;
76     for (int i = 0; i < (int)s.size(); i++) {
77         if (s[i] == '.') {
78             if (++npoints > 1) {
79                 return false;
80             }
81         } else if (!isdigit(s[i])) {
82             return false;
83         }
84     }
85     return !s.empty();
86 }
87
88 Operand eval_operand(const string &s) {
89     Operand res;
90     std::stringstream ss(s);
91     ss >> res;
92     return res;
93 }
94
95 class parser {
96     typedef std::map<string, UnaryOp> unary_op_map;
97     typedef std::map<string, std::pair<BinaryOp, int> > binary_op_map;
98     unary_op_map unary_ops;
99     binary_op_map binary_ops;
100    std::set<string> ops;
101    int max_precedence;
102
103    template<class StrIt>
104    Operand eval_unary(StrIt &lo, StrIt hi) {
105        if (is_operand(*lo)) {
106            return eval_operand(*(lo++));
107        }
108        unary_op_map::iterator it = unary_ops.find(*lo);
109        if (it != unary_ops.end()) {
110            return (it->second)(eval_unary(++lo, hi));
111        }
112        if (*lo != "(") {
113            throw std::runtime_error("Expected \"(\" during eval.");
114        }
115        Operand res = eval_binary(++lo, hi, 0);
116        if (*lo != ")") {
117            throw std::runtime_error("Expected \")\"\\\" during eval.");
118        }

```

```

119     ++lo;
120     return res;
121 }
122
123 template<class StrIt>
124 Operand eval_binary(StrIt &lo, StrIt hi, Operand precedence) {
125     if (precedence > max_precedence) {
126         return eval_unary(lo, hi);
127     }
128     Operand v = eval_binary(lo, hi, precedence + 1);
129     while (lo != hi) {
130         binary_op_map::iterator it;
131         it = binary_ops.find(*lo);
132         if (it == binary_ops.end() || it->second.second != precedence) {
133             return v;
134         }
135         v = (it->second.first)(v, eval_binary(++lo, hi, precedence + 1));
136     }
137     return v;
138 }
139
140 static string strip(string s) {
141     s.erase(s.begin(), std::find_if(s.begin(), s.end(),
142         std::not1(std::ptr_fun<int, int>(std::isspace))));
143     s.erase(std::find_if(s.rbegin(), s.rend(),
144         std::not1(std::ptr_fun<int, int>(std::isspace))).base(), s.end());
145     return s;
146 }
147
148 public:
149     parser(const unary_op_map &unary_ops, const binary_op_map &binary_ops)
150         : unary_ops(unary_ops), binary_ops(binary_ops) {
151     for (unary_op_map::const_iterator it = unary_ops.begin();
152         it != unary_ops.end(); ++it) {
153         ops.insert(it->first);
154     }
155     max_precedence = 0;
156     for (binary_op_map::const_iterator it = binary_ops.begin();
157         it != binary_ops.end(); ++it) {
158         ops.insert(it->first);
159         max_precedence = std::max(max_precedence, it->second.second);
160     }
161 }
162
163     std::vector<string> split(const string &s) {
164         std::vector<string> res;
165         for (int i = 0; i < (int)s.size(); i++) {
166             if (s[i] == '') {
167                 continue;
168             }
169             int next_paren = s.size();
170             for (int j = i; j < (int)s.size(); j++) {
171                 if (s[j] == '(' || s[j] == ')') {
172                     next_paren = j;
173                     break;
174                 }
175             }
176             while (i < next_paren) {
177                 int found = next_paren;

```

```

178     string found_op;
179     for (int j = i; j < next_paren && found == next_paren; j++) {
180         for (std::set<string>::iterator it = ops.begin();
181             it != ops.end(); ++it) {
182             if (s.substr(j, it->size()) == *it) {
183                 found = j;
184                 found_op = *it;
185                 break;
186             }
187         }
188     }
189     string term = strip(s.substr(i, found - i));
190     if (!term.empty()) {
191         res.push_back(term);
192         if (!is_operand(term)) {
193             throw std::runtime_error("Failed to split term: \""
194                                     + term + "\".");
195         }
196     }
197     if (found < next_paren) {
198         res.push_back(found_op);
199         i = found + found_op.size();
200     } else {
201         i = next_paren;
202     }
203     if (next_paren < s.size()) {
204         res.push_back(string(1, s[next_paren]));
205     }
206 }
207 return res;
208 }
209
210 template<class StrIt>
211 Operand eval(StrIt lo, StrIt hi) {
212     Operand res = eval_binary(lo, hi, 0);
213     if (lo != hi) {
214         throw std::runtime_error("Eval failed at token " + *lo + ".");
215     }
216     return res;
217 }
218
219 Operand eval(const string &s) {
220     std::vector<string> tokens = split(s);
221     return eval(tokens.begin(), tokens.end());
222 }
223 };
224
225 /* Example Usage */
226
227 #include <cassert>
228 #include <cmath>
229 using namespace std;
230
231 #define EQ(a, b) (fabs((a) - (b)) < 1e-7)
232
233 double pos(double a) { return +a; }
234 double neg(double a) { return -a; }
235 double add(double a, double b) { return a + b; }
236 double sub(double a, double b) { return a - b; }

```

```

237 double mul(double a, double b) { return a * b; }
238 double div(double a, double b) { return a / b; }
239
240 int main() {
241     map<string, UnaryOp> unary_ops;
242     unary_ops["+"] = pos;
243     unary_ops["-"] = neg;
244
245     map<string, pair<BinaryOp, int> > binary_ops;
246     binary_ops["+"] = make_pair((BinaryOp)add, 0);
247     binary_ops["-"] = make_pair((BinaryOp)sub, 0);
248     binary_ops["*"] = make_pair((BinaryOp)mul, 1);
249     binary_ops["/] = make_pair((BinaryOp)div, 1);
250     binary_ops["^"] = make_pair((BinaryOp)pow, 2);
251
252     parser p(unary_ops, binary_ops);
253     assert(EQ(p.eval("-+((-(-+1)))"), -1));
254     assert(EQ(p.eval("5*(3+3)-2-2"), 26));
255     assert(EQ(p.eval("1+2*3*4+3*(+2)-100"), -69));
256     assert(EQ(p.eval("3*3*3*3*3-2*2*2*2*2*2*2"), 473));
257     assert(EQ(p.eval("3.14 + 3 * (7.7/9.8*32.9 )"), 3.14));
258     assert(EQ(p.eval("5*(3+2)/-1*-2+(-2-2-2+3)-3-(-2)+15/2/2/2+(-2)"), 45.875));
259     assert(EQ(p.eval("123456789./3/3/3*2*2*2+456/6-23/3"), 36579857.6666666667));
260     assert(EQ(p.eval("10/3+10/4+10/5+10/6+10/7+10/8+10/9+10/10+15*23456"),
261             351854.28968253968));
262     assert(EQ(p.eval("-(5-(5-(5-(5-2))))+(3-(3-(3-(3+3))))*"
263                 "(7-(7-(7-(7-7+4*5))))"), 117));
264
265     return 0;
266 }
```

3.3.3 Shunting Yard Parsing

```

1 /*
2
3 Evaluate an expression using a generalized parser class for custom-defined
4 operand types, prefix unary operators, binary operators, and precedences.
5 Typical parentheses behavior is supported, but multiplication by juxtaposition
6 is not. Evaluation is performed using the shunting yard algorithm.
7
8 An arbitrary operand type is supported, with its string representation defined
9 by a user-specified is_operand() and eval_operand() functions. For maximum
10 reliability, the string representation of operands should not use characters
11 shared by any operator. For instance, the best practice instead of accepting
12 "-1" as a valid operand (since the "-" sign may conflict with the identical
13 binary operator), is to specify non-negative number as operands alongside the
14 unary operator "-".
15
16 Operators may be non-empty strings of any length, but should not contain any
17 parentheses or shared characters with the string representations of operands.
18 Ideally, operators should not be prefixes or suffixes of one another, else the
19 tokenization process may be ambiguous. For example, if ++ and + are both
20 operators, then ++ may be split into either ["+", "+"] or ["++"] depending on
21 the lexicographical ordering of conflicting operators.
22
23 - parser(unary_op, binary_op) initializes a parser with operators specified by
```

```

24 maps unary_op (of operator to function pointer) and binary_op (of operator to
25 pair of function pointer and operator precedence). Operator precedences should
26 be numbered upwards starting at 0 (lowest precedence, evaluated last).
27 - split(s) returns a vector of tokens for the expression s, split on the given
28 operators during construction. Each parenthesis, operator, and operand
29 satisfying is_operand() will be split into a separate token. The algorithm is
30 naive, matching operators lazily in the case of overlapping operators as
31 mentioned above. Under these circumstances, the parse may not always succeed.
32 - eval(lo, hi) returns the evaluation of a range [lo, hi) of already split-up
33 expression tokens, where lo and hi must be random-access iterators.
34 - eval(s) returns the evaluation of expression s, after first calling split(s)
35 to obtain the tokens.
36
37 Time Complexity:
38 - O(m) per call to the constructor, where m is the total number of operators.
39 - O(nmk) per call to split(s), where n is the length of s, m is the total number
40 of operators defined for the parser instance, and k is the maximum length for
41 any operator representation.
42 - O(n log m) per call to eval(lo, hi), where n is the distance between lo and hi
43 and m is the total number of operators defined for the parser instance. In
44 C++11 and later, std::unordered_map may be used in place of std::map for
45 storing the unary_ops and binary_ops, which will eliminate the log m factor
46 for a time complexity of O(n) per call.
47 - O(nmk + n log m) per call to eval(s), where n is the distance between lo and
48 hi, and m and k are as defined previous.
49
50 Space Complexity:
51 - O(mk) for storage of the m operators, of maximum length k.
52 - O(n) auxiliary stack space for split(s), eval(lo, hi), and eval(s), where n is
53 the length of the argument.
54 */
55
56
57 #include <algorithm>
58 #include <cctype>
59 #include <functional>
60 #include <map>
61 #include <set>
62 #include <sstream>
63 #include <stack>
64 #include <stdexcept>
65 #include <string>
66 #include <utility>
67 #include <vector>
68 using std::string;
69
70 // Define the custom operand type and representation below.
71 typedef double Operand;
72 typedef Operand (*UnaryOp)(Operand a);
73 typedef Operand (*BinaryOp)(Operand a, Operand b);
74
75 bool is_operand(const string &s) {
76     int npoints = 0;
77     for (int i = 0; i < (int)s.size(); i++) {
78         if (s[i] == '.') {
79             if (++npoints > 1) {
80                 return false;
81             }
82         } else if (!isdigit(s[i])) {

```

```

83     return false;
84 }
85 }
86 return !s.empty();
87 }
88
89 Operand eval_operand(const string &s) {
90     Operand res;
91     std::stringstream ss(s);
92     ss >> res;
93     return res;
94 }
95
96 class parser {
97     typedef std::map<string, UnaryOp> unary_op_map;
98     typedef std::map<string, std::pair<BinaryOp, int> > binary_op_map;
99     unary_op_map unary_ops;
100    binary_op_map binary_ops;
101    std::set<string> ops;
102
103    static string strip(string s) {
104        s.erase(s.begin(), std::find_if(s.begin(), s.end(),
105            std::not1(std::ptr_fun<int, int>(std::isspace))));;
106        s.erase(std::find_if(s.rbegin(), s.rend(),
107            std::not1(std::ptr_fun<int, int>(std::isspace))).base(), s.end());
108        return s;
109    }
110
111 public:
112     parser(const unary_op_map &unary_ops, const binary_op_map &binary_ops)
113         : unary_ops(unary_ops), binary_ops(binary_ops) {
114         for (unary_op_map::const_iterator it = unary_ops.begin();
115             it != unary_ops.end(); ++it) {
116             ops.insert(it->first);
117         }
118         for (binary_op_map::const_iterator it = binary_ops.begin();
119             it != binary_ops.end(); ++it) {
120             ops.insert(it->first);
121         }
122     }
123
124     std::vector<string> split(const string &s) {
125         std::vector<string> res;
126         for (int i = 0; i < (int)s.size(); i++) {
127             if (s[i] == ',') {
128                 continue;
129             }
130             int next_paren = s.size();
131             for (int j = i; j < (int)s.size(); j++) {
132                 if (s[j] == '(' || s[j] == ')') {
133                     next_paren = j;
134                     break;
135                 }
136             }
137             while (i < next_paren) {
138                 int found = next_paren;
139                 string found_op;
140                 for (int j = i; j < next_paren && found == next_paren; j++) {
141                     for (std::set<string>::iterator it = ops.begin();

```

```

142         it != ops.end(); ++it) {
143             if (s.substr(j, it->size()) == *it) {
144                 found = j;
145                 found_op = *it;
146                 break;
147             }
148         }
149     }
150     string term = strip(s.substr(i, found - i));
151     if (!term.empty()) {
152         res.push_back(term);
153         if (!is_operand(term)) {
154             throw std::runtime_error("Failed to split term: " + term + "\\.");
155         }
156     }
157     if (found < next_paren) {
158         res.push_back(found_op);
159         i = found + found_op.size();
160     } else {
161         i = next_paren;
162     }
163 }
164 if (next_paren < s.size()) {
165     res.push_back(string(1, s[next_paren]));
166 }
167 }
168 return res;
169 }

170 template<class StrIt>
171 Operand eval(StrIt lo, StrIt hi) {
172     std::stack<Operand> vals;
173     std::stack<std::pair<string, bool>> ops;
174     ops.push(std::make_pair("(", false));
175     StrIt prev = hi;
176     do {
177         string curr = (lo == hi) ? ")" : *lo;
178         if (is_operand(curr)) {
179             vals.push(eval_operand(curr));
180         } else if (curr == "(") {
181             ops.push(std::make_pair(curr, false));
182         } else if (unary_ops.find(curr) != unary_ops.end() && (prev == hi ||
183             *prev == "(" || binary_ops.find(*prev) != binary_ops.end())) {
184             ops.push(std::make_pair(curr, true));
185         } else {
186             for (;;) {
187                 string op = ops.top().first;
188                 bool is_unary = ops.top().second;
189                 binary_op_map::iterator it1 = binary_ops.find(op);
190                 binary_op_map::iterator it2 = binary_ops.find(curr);
191                 if (!is_unary &&
192                     (it1 == binary_ops.end() ? -1 : it1->second.second) <
193                     (it2 == binary_ops.end() ? -1 : it2->second.second)) {
194                     break;
195                 }
196                 ops.pop();
197                 if (op == "(") {
198                     break;
199                 }
200             }

```

```

201     Operand b = vals.top();
202     vals.pop();
203     if (is_unary) {
204         unary_op_map::iterator it = unary_ops.find(op);
205         if (it == unary_ops.end()) {
206             throw std::runtime_error("Failed to eval unary op: " + op);
207         }
208         vals.push((it->second)(b));
209     } else {
210         Operand a = vals.top();
211         vals.pop();
212         if (it1 == binary_ops.end()) {
213             throw std::runtime_error("Failed to eval binary op: " + op);
214         }
215         vals.push((it1->second.first)(a, b));
216     }
217 }
218 if (curr != ")") {
219     ops.push(std::make_pair(*lo, false));
220 }
221 }
222 prev = lo;
223 } while (lo++ != hi);
224 return vals.top();
225 }
226
227 Operand eval(const string &s) {
228     std::vector<string> tokens = split(s);
229     return eval(tokens.begin(), tokens.end());
230 }
231 };
232
233 /** Example Usage **/
234
235 #include <cassert>
236 #include <cmath>
237 using namespace std;
238
239 #define EQ(a, b) (fabs((a) - (b)) < 1e-7)
240
241 double pos(double a) { return +a; }
242 double neg(double a) { return -a; }
243 double add(double a, double b) { return a + b; }
244 double sub(double a, double b) { return a - b; }
245 double mul(double a, double b) { return a * b; }
246 double div(double a, double b) { return a / b; }
247
248 int main() {
249     map<string, UnaryOp> unary_ops;
250     unary_ops["+"] = pos;
251     unary_ops["-"] = neg;
252
253     map<string, pair<BinaryOp, int> > binary_ops;
254     binary_ops["+"] = make_pair((BinaryOp)add, 0);
255     binary_ops["-"] = make_pair((BinaryOp)sub, 0);
256     binary_ops["*"] = make_pair((BinaryOp)mul, 1);
257     binary_ops["/] = make_pair((BinaryOp)div, 1);
258     binary_ops["^"] = make_pair((BinaryOp)pow, 2);
259 }
```

3.4 Dynamic Programming

3.4.1 Longest Common Substring

```
1  /*
2
3 Given two strings, determine their longest common substring (i.e. consecutive
4 subsequence) using dynamic programming.
5
6 Time Complexity:
7 - O(n*m) per call to longest_common_substring(s1, s2), where n and m are the
8 lengths of s1 and s2, respectively.
9
10 Space Complexity:
11 - O(min(n, m)) auxiliary heap space, where n and m are the lengths of s1 and
12 s2, respectively.
13
14 */
15
16 #include <string>
17 #include <vector>
18 using std::string;
19
20 string longest_common_substring(const string &s1, const string &s2) {
21     int n = s1.size(), m = s2.size();
22     if (n == 0 || m == 0) {
23         return "";
24     }
25     if (n < m) {
26         return longest_common_substring(s2, s1);
27     }
28     std::vector<int> curr(m), prev(m);
29     int pos = 0, len = 0;
30     for (int i = 0; i < n; i++) {
31         for (int j = 0; j < m; j++) {
32             if (s1[i] == s2[j]) {
33                 curr[j] = (i > 0 && j > 0) ? prev[j - 1] + 1 : 1;
34                 if (len < curr[j]) {
35                     len = curr[j];
36                     pos = i;
37                 }
38             }
39         }
40     }
41     return s1.substr(pos, len);
42 }
```

```

35         len = curr[j];
36         pos = i - curr[j] + 1;
37     }
38 } else {
39     curr[j] = 0;
40 }
41 }
42 curr.swap(prev);
43 }
44 return s1.substr(pos, len);
45 }

46 /**
47 *** Example Usage ***
48
49 #include <cassert>
50
51 int main() {
52     assert(longest_common_substring("bbbabca", "aababcd") == "babc");
53     return 0;
54 }
```

3.4.2 Longest Common Subsequence

```

1 /*
2
3 Given two strings, determine their longest common subsequence. A subsequence is
4 a string that can be derived from the original string by deleting some elements
5 without changing the order of the remaining elements (e.g. "ACE" is a
6 subsequence of "ABCDE", but "BAE" is not).
7
8 - longest_common_subsequence(s1, s2) returns the longest common subsequence of
9   strings s1 and s2 using a classic dynamic programming approach. This
10  implementation computes dp[i][j] (the length of the longest common subsequence
11  for the length i prefix of s1 and the length j prefix of s2) before following
12  the path backwards to construct the answer.
13 - hirschberg_lcs(s1, s2) returns the longest common subsequence of strings s1
14  and s2 using the more memory efficient Hirschberg's algorithm.
15
16 Time Complexity:
17 - O(n*m) per call to longest_common_subsequence(s1, s2) as well as
18  hirschberg_lcs(s1, s2), where n and m are the lengths of s1 and s2,
19  respectively.
20
21 Space Complexity:
22 - O(n*m) auxiliary heap space for longest_common_subsequence(s1, s2), where n
23  and m are the lengths of s1 and s2, respectively.
24 - O(log max(n, m)) auxiliary stack space and O(min(n, m)) auxiliary heap space
25  for hirschberg_lcs(s1, s2), where n and m are the lengths of s1 and s2,
26  respectively.
27
28 */
29
30 #include <algorithm>
31 #include <iterator>
32 #include <string>
```

```

33 #include <vector>
34 using std::string;
35
36 string longest_common_subsequence(const string &s1, const string &s2) {
37     int n = s1.size(), m = s2.size();
38     std::vector<std::vector<int>> dp(n + 1, std::vector<int>(m + 1, 0));
39     for (int i = 1; i <= n; i++) {
40         for (int j = 1; j <= m; j++) {
41             if (s1[i - 1] == s2[j - 1]) {
42                 dp[i][j] = dp[i - 1][j - 1] + 1;
43             } else {
44                 dp[i][j] = std::max(dp[i][j - 1], dp[i - 1][j]);
45             }
46         }
47     }
48     string res;
49     for (int i = n, j = m; i > 0 && j > 0; ) {
50         if (s1[i - 1] == s2[j - 1]) {
51             res += s1[i - 1];
52             i--;
53             j--;
54         } else if (dp[i - 1][j] >= dp[i][j - 1]) {
55             i--;
56         } else {
57             j--;
58         }
59     }
60     std::reverse(res.begin(), res.end());
61     return res;
62 }
63
64 template<class It>
65 std::vector<int> lcs_len(It lo1, It hi1, It lo2, It hi2) {
66     std::vector<int> res(std::distance(lo2, hi2) + 1, prev(res));
67     for (It it1 = lo1; it1 != hi1; ++it1) {
68         res.swap(prev);
69         int i = 0;
70         for (It it2 = lo2; it2 != hi2; ++it2) {
71             res[i + 1] = (*it1 == *it2) ? prev[i] + 1 : std::max(res[i], prev[i + 1]);
72             i++;
73         }
74     }
75     return res;
76 }
77
78 template<class It>
79 void hirschberg_rec(It lo1, It hi1, It lo2, It hi2, string *res) {
80     if (lo1 == hi1) {
81         return;
82     }
83     if (lo1 + 1 == hi1) {
84         if (std::find(lo2, hi2, *lo1) != hi2) {
85             *res += *lo1;
86         }
87         return;
88     }
89     It mid1 = lo1 + (hi1 - lo1)/2;
90     std::reverse_iterator<It> rlo1(hi1), rmid1(mid1), rlo2(hi2), rhi2(lo2);
91     std::vector<int> fwd = lcs_len(lo1, mid1, lo2, hi2);

```

```

92     std::vector<int> rev = lcs_len(rlo1, rmid1, rlo2, rhi2);
93     It mid2 = lo2;
94     int maxlen = -1;
95     for (int i = 0, j = (int)rev.size() - 1; i < (int)fwd.size(); i++, j--) {
96         if (fwd[i] + rev[j] > maxlen) {
97             maxlen = fwd[i] + rev[j];
98             mid2 = lo2 + i;
99         }
100    }
101    hirschberg_rec(lo1, mid1, lo2, mid2, res);
102    hirschberg_rec(mid1, hi1, mid2, hi2, res);
103 }
104
105 string hirschberg_lcs(const string &s1, const string &s2) {
106     if (s1.size() < s2.size()) {
107         return hirschberg_lcs(s2, s1);
108     }
109     string res;
110     hirschberg_rec(s1.begin(), s1.end(), s2.begin(), s2.end(), &res);
111     return res;
112 }
113
114 /** Example Usage ***/
115
116 #include <cassert>
117
118 int main() {
119     assert(longest_common_subsequence("xmjyauz", "mzjawxu") == "mjau");
120     assert(hirschberg_lcs("xmjyauz", "mzjawxu") == "mjau");
121     return 0;
122 }
```

3.4.3 Sequence Alignment

```

1  /*
2
3 Given two strings, determine their minimum-cost alignment. An alignment of two
4 strings is a transformation of both strings by inserting gap characters '_' in
5 some way to make the final lengths equal. The total cost of an alignment given
6 a gap_cost (insertion or deletion cost) and a sub_cost (substitution, i.e.
7 mismatch cost) is gap_cost*(the number of gaps inserted across both strings),
8 plus sub_cost*(the number of indices at which the two aligned strings differ).
9
10 - align_sequences(s1, s2, gap_cost, sub_cost) returns a pair of aligned strings
11   for strings s1 and s2, using a classic dynamic programming approach. This
12   implementation first computes dp[i][j] (the cost of aligning the length i
13   prefix of s1 with the length j prefix of s2) before following the path
14   backwards to construct the answer. For gap_cost = sub_cost = 1, dp[n][m] will
15   be the Levenshtein edit distance, where n and m are the lengths of s1 and
16   s2, respectively.
17 - hirschberg_align_sequences(s1, s2, gap_cost, sub_cost) returns the sequence
18   alignment of strings s1 and s2 using the more memory efficient Hirschberg's
19   algorithm.
20
21 Time Complexity:
```

```

22 - O(n*m) per call to align_sequences(s1, s2) as well as
23 hirschberg_align_sequences(s1, s2), where n and m are the lengths of s1 and
24 s2, respectively.
25
26 Space Complexity:
27 - O(n*m) auxiliary heap space for align_sequences(s1, s2), where n and m are the
28 lengths of s1 and s2, respectively.
29 - O(log max(n, m)) auxiliary stack space and O(min(n, m)) auxiliary heap space
30 for hirschberg_align_sequences(s1, s2), where n and m are the lengths of s1
31 and s2, respectively.
32
33 */
34
35 #include <algorithm>
36 #include <string>
37 #include <vector>
38 #include <utility>
39 using std::string;
40
41 std::pair<string, string> align_sequences(
42     const string &s1, const string &s2, int gap_cost = 1, int sub_cost = 1) {
43     int n = s1.size(), m = s2.size();
44     std::vector<std::vector<int>> dp(n + 1, std::vector<int>(m + 1, 0));
45     for (int i = 0; i <= n; i++) {
46         dp[i][0] = i * gap_cost;
47     }
48     for (int j = 0; j <= m; j++) {
49         dp[0][j] = j * gap_cost;
50     }
51     for (int i = 1; i <= n; i++) {
52         for (int j = 1; j <= m; j++) {
53             dp[i][j] = (s1[i - 1] == s2[j - 1]) ? dp[i - 1][j - 1] : std::min(
54                 dp[i - 1][j - 1] + sub_cost,
55                 std::min(dp[i - 1][j], dp[i][j - 1]) + gap_cost);
56         }
57     }
58     string res1, res2;
59     int i = n, j = m;
60     while (i > 0 && j > 0) {
61         if (s1[i - 1] == s2[j - 1] || dp[i][j] == dp[i - 1][j - 1] + sub_cost) {
62             res1 += s1[--i];
63             res2 += s2[--j];
64         } else if (dp[i][j] == dp[i - 1][j] + gap_cost) {
65             res1 += s1[--i];
66             res2 += '_';
67         } else if (dp[i][j] == dp[i][j - 1] + gap_cost) {
68             res1 += '_';
69             res2 += s2[--j];
70         }
71     }
72     while (i > 0 || j > 0) {
73         res1 += (i > 0) ? s1[--i] : '_';
74         res2 += (j > 0) ? s2[--j] : '_';
75     }
76     std::reverse(res1.begin(), res1.end());
77     std::reverse(res2.begin(), res2.end());
78     return std::make_pair(res1, res2);
79 }
80

```

```

81 template<class It>
82 std::vector<int> row_cost(It lo1, It hi1, It lo2, It hi2,
83                           int gap_cost, int sub_cost) {
84     std::vector<int> res(std::distance(lo2, hi2) + 1), prev(res);
85     for (It it1 = lo1; it1 != hi1; ++it1) {
86         res.swap(prev);
87         int i = 0;
88         for (It it2 = lo2; it2 != hi2; ++it2) {
89             res[i + 1] = (*it1 == *it2) ? prev[i] : std::min(prev[i] + sub_cost,
90                                                 res[i] + gap_cost);
91             i++;
92         }
93     }
94     return res;
95 }
96
97 template<class It>
98 void hirschberg_rec(It lo1, It hi1, It lo2, It hi2,
99                      string *res1, string *res2, int gap_cost, int sub_cost) {
100    if (lo1 == hi1) {
101        for (It it2 = lo2; it2 != hi2; ++it2) {
102            *res1 += '_';
103            *res2 += *it2;
104        }
105        return;
106    }
107    if (lo1 + 1 == hi1) {
108        It pos = std::find(lo2, hi2, *lo1);
109        bool insert = (pos == hi2) && (gap_cost*(hi2 - lo2 + 1) < sub_cost);
110        if (lo2 == hi2 || insert) {
111            *res1 += *lo1;
112            *res2 += '_';
113        }
114        for (It it2 = lo2; it2 != hi2; ++it2) {
115            *res1 += (pos == it2 || (!insert && it2 == lo2)) ? *lo1 : '_';
116            *res2 += *it2;
117        }
118        return;
119    }
120    It mid1 = lo1 + (hi1 - lo1)/2;
121    std::reverse_iterator<It> rlo1(hi1), rmid1(mid1), rlo2(hi2), rhi2(lo2);
122    std::vector<int> fwd = row_cost(lo1, mid1, lo2, hi2, gap_cost, sub_cost);
123    std::vector<int> rev = row_cost(rlo1, rmid1, rlo2, rhi2, gap_cost, sub_cost);
124    It mid2 = lo2;
125    int mincost = -1;
126    for (int i = 0, j = (int)rev.size() - 1; i < (int)fwd.size(); i++, j--) {
127        if (mincost < 0 || fwd[i] + rev[j] < mincost) {
128            mincost = fwd[i] + rev[j];
129            mid2 = lo2 + i;
130        }
131    }
132    hirschberg_rec(lo1, mid1, lo2, mid2, res1, res2, gap_cost, sub_cost);
133    hirschberg_rec(mid1, hi1, mid2, hi2, res1, res2, gap_cost, sub_cost);
134 }
135
136 std::pair<string, string> hirschberg_align_sequences(
137     const string &s1, const string &s2, int gap_cost = 1, int sub_cost = 1) {
138     if (s1.size() < s2.size()) {
139         return hirschberg_align_sequences(s2, s1, gap_cost, sub_cost);

```

```

140     }
141     string res1, res2;
142     hirschberg_rec(s1.begin(), s1.end(), s2.begin(), s2.end(), &res1, &res2,
143                     gap_cost, sub_cost);
144     return std::make_pair(res1, res2);
145 }
146
147 /*** Example Usage ***/
148
149 #include <cassert>
150
151 int main() {
152     assert(align_sequences("AGGGCT", "AGGCA", 2, 3) ==
153             make_pair(string("AGGGCT"), string("A_GGCA")));
154     assert(hirschberg_align_sequences("AGGGCT", "AGGCA", 2, 3) ==
155             make_pair(string("AGGGCT"), string("A_GGCA")));
156     return 0;
157 }
```

3.5 Suffix Array and LCP

3.5.1 Suffix Array and LCP (Manber-Myers)

```

1 /*
2
3 Given a string s, a suffix array is the array of the smallest starting positions
4 for the sorted suffixes of s. That is, the i-th position of the suffix array
5 stores the starting position of the i-th lexicographically smallest suffix of s.
6 For examples, s = "cab" has the suffixes "cab", "ab", and "b". When sorted, the
7 indices of the suffixes are "ab", "b", and "cab", so the suffix array (assuming
8 zero-based indices) is [1, 2, 0].
9
10 For a string s of length n the longest common prefix (LCP) array of length n - 1
11 stores the lengths of the longest common prefixes between all pairs of
12 lexicographically adjacent suffixes in s. For example, "baa" has the sorted
13 suffixes "a", "aa", and "baa", with an LCP array of [1, 0].
14
15 - suffix_array(s) constructs a suffix array from the given string s using the
16   original Manber-Myers gap partitioning algorithm with a comparison-based sort.
17 - get_sa() returns the constructed suffix array.
18 - get_lcp() returns the corresponding LCP array for the suffix array.
19 - find(needle) returns one position that needle occurs in s (not necessarily the
20   first), or std::string::npos if it cannot be found. For a needle of length m,
21   this implementation uses an O(m log n) binary search, but can be optimized to
22   O(m + log n) by first computing the LCP-LR array using the LCP array.
23
24 Time Complexity:
25 - O(n log^2 n) per call to the constructor, where n is the length of s.
26 - O(1) per call to get_sa().
27 - O(n) per call to get_lcp(), where n is the length of s.
28 - O(m log n) per call to find(needle), where m is the length of needle and n is
29   the length of s.
30
```

```

31 Space Complexity:
32 - O(n) auxiliary for storage of the suffix and LCP arrays, where n is the length
33   of s.
34 - O(n) auxiliary heap space for the constructor.
35 - O(1) auxiliary space for all other operations.
36
37 */
38
39 #include <algorithm>
40 #include <string>
41 #include <utility>
42 #include <vector>
43 using std::string;
44
45 class suffix_array {
46     struct comp {
47         const std::vector<std::pair<int, int> > &rank;
48
49         comp(const std::vector<std::pair<int, int> > &rank) : rank(rank) {}
50
51         bool operator()(int i, int j) {
52             return rank[i] < rank[j];
53         }
54     };
55
56     string s;
57     std::vector<int> sa, rank;
58
59 public:
60     suffix_array(const string &s) : s(s), sa(s.size()), rank(s.size()) {
61         int n = s.size();
62         for (int i = 0; i < n; i++) {
63             sa[i] = i;
64             rank[i] = (int)s[i];
65         }
66         std::vector<std::pair<int, int> > rank2(n);
67         for (int gap = 1; gap < n; gap *= 2) {
68             for (int i = 0; i < n; i++) {
69                 rank2[i] = std::make_pair(rank[i], i + gap < n ? rank[i + gap] + 1 : 0);
70             }
71             std::sort(sa.begin(), sa.end(), comp(rank2));
72             for (int i = 0; i < n; i++) {
73                 rank[sa[i]] = (i > 0 && rank2[sa[i - 1]] == rank2[sa[i]])
74                         ? rank[sa[i - 1]] : i;
75             }
76         }
77     }
78
79     std::vector<int> get_sa() {
80         return sa;
81     }
82
83     std::vector<int> get_lcp() {
84         int n = s.size();
85         std::vector<int> lcp(n - 1);
86         for (int i = 0, k = 0; i < n; i++) {
87             if (rank[i] < n - 1) {
88                 int j = sa[rank[i] + 1];
89                 while (std::max(i, j) + k < n && s[i + k] == s[j + k]) {

```

```

90         k++;
91     }
92     lcp[rank[i]] = k;
93     if (k > 0) {
94         k--;
95     }
96 }
97
98 return lcp;
99 }
100
101 size_t find(const string &needle) {
102     int lo = 0, hi = (int)s.size() - 1;
103     while (lo <= hi) {
104         int mid = lo + (hi - lo)/2;
105         int cmp = s.compare(sa[mid], needle.size(), needle);
106         if (cmp < 0) {
107             lo = mid + 1;
108         } else if (cmp > 0) {
109             hi = mid - 1;
110         } else {
111             return sa[mid];
112         }
113     }
114     return string::npos;
115 }
116 };
117
118 /** Example Usage ***/
119
120 #include <cassert>
121 using namespace std;
122
123 int main() {
124     suffix_array sa("banana");
125     vector<int> sarr = sa.get_sa(), lcp = sa.get_lcp();
126     int sarr_expected[] = {5, 3, 1, 0, 4, 2};
127     int lcp_expected[] = {1, 3, 0, 0, 2};
128     assert(equal(sarr.begin(), sarr.end(), sarr_expected));
129     assert(equal(lcp.begin(), lcp.end(), lcp_expected));
130     assert(sa.find("ana") == 1);
131     assert(sa.find("x") == string::npos);
132     return 0;
133 }
```

3.5.2 Suffix Array and LCP (Counting Sort)

```

1 /*
2
3 Given a string s, a suffix array is the array of the smallest starting positions
4 for the sorted suffixes of s. That is, the i-th position of the suffix array
5 stores the starting position of the i-th lexicographically smallest suffix of s.
6 For examples, s = "cab" has the suffixes "cab", "ab", and "b". When sorted, the
7 indices of the suffixes are "ab", "b", and "cab", so the suffix array (assuming
8 zero-based indices) is [1, 2, 0].
```

```

9
10 For a string s of length n the longest common prefix (LCP) array of length n - 1
11 stores the lengths of the longest common prefixes between all pairs of
12 lexicographically adjacent suffixes in s. For example, "baa" has the sorted
13 suffixes "a", "aa", and "baa", with an LCP array of [1, 0].
14
15 - suffix_array(s) constructs a suffix array from the given string s using the
16 original Manber-Myers gap partitioning algorithm with a counting sort instead
17 of a comparison-based sort to reduce the running time to O(n log n).
18 - get_sa() returns the constructed suffix array.
19 - get_lcp() returns the corresponding LCP array for the suffix array.
20 - find(needle) returns one position that needle occurs in s (not necessarily the
21 first), or std::string::npos if it cannot be found. For a needle of length m,
22 this implementation uses an O(m log n) binary search, but can be optimized to
23 O(m + log n) by first computing the LCP-LR array using the LCP array.
24
25 Time Complexity:
26 - O(n log n) per call to the constructor, where n is the length of s.
27 - O(1) per call to get_sa().
28 - O(n) per call to get_lcp(), where n is the length of s.
29 - O(m log n) per call to find(needle), where m is the length of needle and n is
30 the length of s.
31
32 Space Complexity:
33 - O(n) auxiliary for storage of the suffix and LCP arrays, where n is the length
34 of s.
35 - O(n) auxiliary heap space for the constructor.
36 - O(1) auxiliary space for all other operations.
37
38 */
39
40 #include <algorithm>
41 #include <string>
42 #include <utility>
43 #include <vector>
44 using std::string;
45
46 class suffix_array {
47     struct comp {
48         const string &s;
49
50         comp(const string &s) : s(s) {}
51
52         bool operator()(int i, int j) {
53             return s[i] < s[j];
54         }
55     };
56
57     string s;
58     std::vector<int> sa, rank;
59
60 public:
61     suffix_array(const string &s) : s(s), sa(s.size()), rank(s.size()) {
62         int n = s.size();
63         for (int i = 0; i < n; i++) {
64             sa[i] = n - 1 - i;
65             rank[i] = (int)s[i];
66         }
67         std::stable_sort(sa.begin(), sa.end(), comp(s));

```

```

68     for (int gap = 1; gap < n; gap *= 2) {
69         std::vector<int> prev_rank(rank), prev_sa(sa), cnt(n);
70         for (int i = 0; i < n; i++) {
71             cnt[i] = i;
72         }
73         for (int i = 0; i < n; i++) {
74             rank[sa[i]] = (i > 0 && prev_rank[sa[i - 1]] == prev_rank[sa[i]] &&
75                         sa[i - 1] + gap < n &&
76                         prev_rank[sa[i - 1] + gap/2] == prev_rank[sa[i] + gap/2])
77                         ? rank[sa[i - 1]] : i;
78         }
79         for (int i = 0; i < n; i++) {
80             int s1 = prev_sa[i] - gap;
81             if (s1 >= 0) {
82                 sa[cnt[rank[s1]]++] = s1;
83             }
84         }
85     }
86 }
87
88 std::vector<int> get_sa() {
89     return sa;
90 }
91
92 std::vector<int> get_lcp() {
93     int n = s.size();
94     std::vector<int> lcp(n - 1);
95     for (int i = 0, k = 0; i < n; i++) {
96         if (rank[i] < n - 1) {
97             int j = sa[rank[i] + 1];
98             while (std::max(i, j) + k < n && s[i + k] == s[j + k]) {
99                 k++;
100            }
101            lcp[rank[i]] = k;
102            if (k > 0) {
103                k--;
104            }
105        }
106    }
107    return lcp;
108 }
109
110 size_t find(const string &needle) {
111     int lo = 0, hi = (int)s.size() - 1;
112     while (lo <= hi) {
113         int mid = lo + (hi - lo)/2;
114         int cmp = s.compare(sa[mid], needle.size(), needle);
115         if (cmp < 0) {
116             lo = mid + 1;
117         } else if (cmp > 0) {
118             hi = mid - 1;
119         } else {
120             return sa[mid];
121         }
122     }
123     return string::npos;
124 }
125 };
126

```

```

127 /** Example Usage **/
128
129 #include <cassert>
130 using namespace std;
131
132 int main() {
133     suffix_array sa("banana");
134     vector<int> sarr = sa.get_sa(), lcp = sa.get_lcp();
135     int sarr_expected[] = {5, 3, 1, 0, 4, 2};
136     int lcp_expected[] = {1, 3, 0, 0, 2};
137     assert(equal(sarr.begin(), sarr.end(), sarr_expected));
138     assert(equal(lcp.begin(), lcp.end(), lcp_expected));
139     assert(sa.find("ana") == 1);
140     assert(sa.find("x") == string::npos);
141     return 0;
142 }
```

3.5.3 Suffix Array and LCP (Linear DC3)

```

1 /*
2
3 Given a string s, a suffix array is the array of the smallest starting positions
4 for the sorted suffices of s. That is, the i-th position of the suffix array
5 stores the starting position of the i-th lexicographically smallest suffix of s.
6 For examples, s = "cab" has the suffices "cab", "ab", and "b". When sorted, the
7 indices of the suffixes are "ab", "b", and "cab", so the suffix array (assuming
8 zero-based indices) is [1, 2, 0].
9
10 For a string s of length n the longest common prefix (LCP) array of length n - 1
11 stores the lengths of the longest common prefixes between all pairs of
12 lexicographically adjacent suffices in s. For example, "baa" has the sorted
13 suffices "a", "aa", and "baa", with an LCP array of [1, 0].
14
15 - suffix_array(s) constructs a suffix array from the given string s using the
16   linear time DC3/skew algorithm by Karkkainen & Sanders (2003) with radix sort.
17 - get_sa() returns the constructed suffix array.
18 - get_lcp() returns the corresponding LCP array for the suffix array.
19 - find(needle) returns one position that needle occurs in s (not necessarily the
20   first), or std::string::npos if it cannot be found. For a needle of length m,
21   this implementation uses an O(m log n) binary search, but can be optimized to
22   O(m + log n) by first computing the LCP-LR array using the LCP array.
23
24 Time Complexity:
25 - O(n) per call to the constructor, where n is the length of s.
26 - O(1) per call to get_sa().
27 - O(n) per call to get_lcp(), where n is the length of s.
28 - O(m log n) per call to find(needle), where m is the length of needle and n is
29   the length of s.
30
31 Space Complexity:
32 - O(n) auxiliary for storage of the suffix and LCP arrays, where n is the length
33   of s.
34 - O(n) auxiliary heap space for the constructor.
35 - O(1) auxiliary space for all other operations.
36
```

```

37  */
38
39 #include <algorithm>
40 #include <string>
41 #include <utility>
42 #include <vector>
43 using std::string;
44
45 class suffix_array {
46     static bool leq(int a1, int a2, int b1, int b2) {
47         return (a1 < b1) || (a1 == b1 && a2 <= b2);
48     }
49
50     static bool leq(int a1, int a2, int a3, int b1, int b2, int b3) {
51         return (a1 < b1) || (a1 == b1 && leq(a2, a3, b2, b3));
52     }
53
54     template<class It>
55     static void radix_pass(It a, It b, It r, int n, int K) {
56         std::vector<int> cnt(K + 1);
57         for (int i = 0; i < n; i++) {
58             cnt[r[a[i]]]++;
59         }
60         for (int i = 1; i <= K; i++) {
61             cnt[i] += cnt[i - 1];
62         }
63         for (int i = n - 1; i >= 0; i--) {
64             b[--cnt[r[a[i]]]] = a[i];
65         }
66     }
67
68     template<class It>
69     static void suffix_array_dc3(It s, It sa, int n, int K) {
70         int n0 = (n + 2)/3, n1 = (n + 1)/3, n2 = n/3, n02 = n0 + n2;
71         std::vector<int> s12(n02 + 3), sa12(n02 + 3), s0(n0), sa0(n0);
72         s12[n02] = s12[n02 + 1] = s12[n02 + 2] = 0;
73         sa12[n02] = sa12[n02 + 1] = sa12[n02 + 2] = 0;
74         for (int i = 0, j = 0; i < n + n0 - n1; i++) {
75             if (i % 3 != 0) {
76                 s12[j++] = i;
77             }
78         }
79         radix_pass(s12.begin(), sa12.begin(), s + 2, n02, K);
80         radix_pass(sa12.begin(), s12.begin(), s + 1, n02, K);
81         radix_pass(s12.begin(), sa12.begin(), s, n02, K);
82         int name = 0, c0 = -1, c1 = -1, c2 = -1;
83         for (int i = 0; i < n02; i++) {
84             if (s[sa12[i]] != c0 || s[sa12[i] + 1] != c1 || s[sa12[i] + 2] != c2) {
85                 name++;
86                 c0 = s[sa12[i]];
87                 c1 = s[sa12[i] + 1];
88                 c2 = s[sa12[i] + 2];
89             }
90             (sa12[i] % 3 == 1 ? s12[sa12[i]/3] : s12[sa12[i]/3 + n0]) = name;
91         }
92         if (name < n02) {
93             suffix_array_dc3(s12.begin(), sa12.begin(), n02, name);
94             for (int i = 0; i < n02; i++) {
95                 s12[sa12[i]] = i + 1;

```

```

96         }
97     } else {
98         for (int i = 0; i < n02; i++) {
99             sa12[s12[i] - 1] = i;
100        }
101    }
102    for (int i = 0, j = 0; i < n02; i++) {
103        if (sa12[i] < n0) {
104            s0[j++] = 3*sa12[i];
105        }
106    }
107    radix_pass(s0.begin(), sa0.begin(), s, n0, K);
108    for (int p = 0, t = n0 - n1, k = 0; k < n; k++) {
109        int i = (sa12[t] < n0) ? 3*sa12[t] + 1 : 3*(sa12[t] - n0) + 2, j = sa0[p];
110        if (sa12[t] < n0 ? leq(s[i], s12[sa12[t] + n0], s[j], s12[j/3])
111            : leq(s[i], s[i + 1], s12[sa12[t] - n0 + 1], s[j],
112                  s[j + 1], s12[j / 3 + n0])) {
113            sa[k] = i;
114            if (++t == n02) {
115                for (k++; p < n0; p++, k++) {
116                    sa[k] = sa0[p];
117                }
118            }
119        } else {
120            sa[k] = j;
121            if (++p == n0) {
122                for (k++; t < n02; t++, k++) {
123                    sa[k] = (sa12[t] < n0) ? 3*sa12[t] + 1 : 3*(sa12[t] - n0) + 2;
124                }
125            }
126        }
127    }
128 }
129
130 string s;
131 std::vector<int> sa;
132
133 public:
134     suffix_array(const string &s) : s(s), sa(s.size() + 1) {
135         int n = s.size();
136         std::vector<int> scopy(s.begin(), s.end());
137         scopy.resize(n + 3);
138         suffix_array_dc3(scopy.begin(), sa.begin(), n + 1, 255);
139         sa.erase(sa.begin());
140     }
141
142     std::vector<int> get_sa() {
143         return sa;
144     }
145
146     std::vector<int> get_lcp() {
147         int n = s.size();
148         std::vector<int> rank(n), lcp(n - 1);
149         for (int i = 0; i < n; i++) {
150             rank[sa[i]] = i;
151         }
152         for (int i = 0, k = 0; i < n; i++) {
153             if (rank[i] < n - 1) {
154                 int j = sa[rank[i] + 1];

```

```

155     while (std::max(i, j) + k < n && s[i + k] == s[j + k]) {
156         k++;
157     }
158     lcp[rank[i]] = k;
159     if (k > 0) {
160         k--;
161     }
162 }
163 }
164 return lcp;
165 }
166
167 size_t find(const string &needle) {
168     int lo = 0, hi = (int)s.size() - 1;
169     while (lo <= hi) {
170         int mid = lo + (hi - lo)/2;
171         int cmp = s.compare(sa[mid], needle.size(), needle);
172         if (cmp < 0) {
173             lo = mid + 1;
174         } else if (cmp > 0) {
175             hi = mid - 1;
176         } else {
177             return sa[mid];
178         }
179     }
180     return string::npos;
181 }
182 };
183
184 /** Example Usage ***/
185
186 #include <cassert>
187 using namespace std;
188
189 int main() {
190     suffix_array sa("banana");
191     vector<int> sarr = sa.get_sa(), lcp = sa.get_lcp();
192     int sarr_expected[] = {5, 3, 1, 0, 4, 2};
193     int lcp_expected[] = {1, 3, 0, 0, 2};
194     assert(equal(sarr.begin(), sarr.end(), sarr_expected));
195     assert(equal(lcp.begin(), lcp.end(), lcp_expected));
196     assert(sa.find("ana") == 1);
197     assert(sa.find("x") == string::npos);
198     return 0;
199 }
```

3.6 String Data Structures

3.6.1 Trie

```

1 /*
2
3  Maintain a map of strings to values using an ordered tree data structure. Each
```

```

4 node corresponds to a character, and each inserted string corresponds to a path
5 from the root to a node that is flagged as a terminal node.
6
7 - trie() constructs an empty map.
8 - size() returns the size of the map.
9 - empty() returns whether the map is empty.
10 - insert(s, v) adds an entry with string key s and value v to the map, returning
11   true if a new entry was added or false if the string already exists (in which
12   case the map is unchanged and the old value associated with the string key is
13   preserved).
14 - erase(s) removes the entry with string key s from the map, returning true if
15   the removal was successful or false if the string to be removed was not found.
16 - find(s) returns a pointer to a const value associated with string key s, or
17   NULL if the key was not found.
18 - walk(f) calls the function f(s, v) on each entry of the map, in
19   lexicographically ascending order of the string keys.
20
21 Time Complexity:
22 -  $O(n)$  per call to insert(s, v), erase(s), and find(s), where n is the length of
23   s. Note that there is a hidden factor of log(alphabet_size) which can be
24   considered constant, since char can only take on  $2^{CHAR\_BIT}$  values. The
25   implementation may be optimized by storing the children of nodes in an
26   std::unordered_map in C++11 and later, or an array if a smaller alphabet size
27   is guaranteed.
28 -  $O(1)$  per call to walk(), where l is the total length of string keys that are
29   currently in the map.
30 -  $O(1)$  per call to all other operations.
31
32 Space Complexity:
33 -  $O(1)$  for storage of the trie, where l is the total length of string keys that
34   are currently in the map.
35 -  $O(n)$  auxiliary stack space for construction, destruction, walk(), where n is
36   the maximum length of any string that has been inserted so far.
37 -  $O(n)$  auxiliary stack space for erase(s), where n is the length of s.
38 -  $O(1)$  auxiliary for all other operations.
39
40 */
41
42 #include <cstddef>
43 #include <map>
44 #include <string>
45 #include <utility>
46 using std::string;
47
48 template<class V>
49 class trie {
50     struct node_t {
51         V value;
52         bool is_terminal;
53         std::map<char, node_t*> children;
54
55         node_t() : is_terminal(false) {}
56     } *root;
57
58     typedef typename std::map<char, node_t*>::iterator cit;
59
60     static bool erase(node_t *n, const string &s, int i) {
61         if (i == (int)s.size()) {
62             if (!n->is_terminal) {

```

```

63         return false;
64     }
65     n->is_terminal = false;
66     return true;
67 }
68 cit it = n->children.find(s[i]);
69 if (it == n->children.end() || !erase(it->second, s, i + 1)) {
70     return false;
71 }
72 if (it->second->children.empty()) {
73     delete it->second;
74     n->children.erase(it);
75 }
76     return true;
77 }
78
79 template<class KVFunction>
80 static void walk(node_t *n, string &s, KVFunction f) {
81     if (n->is_terminal) {
82         f(s, n->value);
83     }
84     for (cit it = n->children.begin(); it != n->children.end(); ++it) {
85         s += it->first;
86         walk(it->second, s, f);
87         s.pop_back();
88     }
89 }
90
91 static void clean_up(node_t *n) {
92     for (cit it = n->children.begin(); it != n->children.end(); ++it) {
93         clean_up(it->second);
94     }
95     delete n;
96 }
97
98 int num_terminals;
99
100 public:
101     trie() : root(new node_t()), num_terminals(0) {}
102
103     ~trie() {
104         clean_up(root);
105     }
106
107     int size() const {
108         return num_terminals;
109     }
110
111     bool empty() const {
112         return num_terminals == 0;
113     }
114
115     bool insert(const string &s, const V &v) {
116         node_t *n = root;
117         for (int i = 0; i < (int)s.size(); i++) {
118             cit it = n->children.find(s[i]);
119             if (it == n->children.end()) {
120                 n->children[s[i]] = new node_t();
121             }

```

```

122     n = n->children[s[i]];
123 }
124 if (n->is_terminal) {
125     return false;
126 }
127 num_terminals++;
128 n->is_terminal = true;
129 n->value = v;
130 return true;
131 }
132
133 bool erase(const string &s) {
134     if (erase(root, s, 0)) {
135         num_terminals--;
136         return true;
137     }
138     return false;
139 }
140
141 const V* find(const string &s) const {
142     node_t *n = root;
143     for (int i = 0; i < (int)s.size(); i++) {
144         cit it = n->children.find(s[i]);
145         if (it == n->children.end()) {
146             return NULL;
147         }
148         n = it->second;
149     }
150     return n->is_terminal ? &(n->value) : NULL;
151 }
152
153 template<class KVFunction>
154 void walk(KVFunction f) const {
155     string s = "";
156     walk(root, s, f);
157 }
158 };
159
160 /**
161 ("", 0)
162 ("a", 1)
163 ("i", 6)
164 ("in", 7)
165 ("inn", 8)
166 ("tea", 3)
167 ("ted", 4)
168 ("ten", 5)
169 ("to", 2)
170
171 */
172
173 #include <cassert>
174 #include <iostream>
175 using namespace std;
176
177 void print_entry(const string &k, int v) {
178     cout << "(" << k << "\", " << v << ")" << endl;
179 }
180

```

```

181
182 int main() {
183     string s[9] = {"", "a", "to", "tea", "ted", "ten", "i", "in", "inn"};
184     trie<int> t;
185     assert(t.empty());
186     for (int i = 0; i < 9; i++) {
187         assert(t.insert(s[i], i));
188     }
189     t.walk(print_entry);
190     assert(!t.empty());
191     assert(t.size() == 9);
192     assert(!t.insert(s[0], 2));
193     assert(t.size() == 9);
194     assert(*t.find("") == 0);
195     assert(*t.find("ten") == 5);
196     assert(t.erase("tea"));
197     assert(t.size() == 8);
198     assert(t.find("tea") == NULL);
199     assert(t.erase(""));
200     assert(t.find("") == NULL);
201     return 0;
202 }
```

3.6.2 Radix Tree

```

1 /*
2
3 Maintain a map of strings to values using an ordered tree data structure. Each
4 node corresponds to a substring of an inserted string, and each inserted string
5 corresponds to a path from the root to a node that is flagged as a terminal
6 node. Contrary to a regular trie, a radix tree is more space efficient as it
7 combines chains of nodes with only a single child.
8
9 - radix_tree() constructs an empty map.
10 - size() returns the size of the map.
11 - empty() returns whether the map is empty.
12 - insert(s, v) adds an entry with string key s and value v to the map, returning
13   true if a new entry was added or false if the string already exists (in which
14   case the map is unchanged and the old value associated with the string key is
15   preserved).
16 - erase(s) removes the entry with string key s from the map, returning true if
17   the removal was successful or false if the string to be removed was not found.
18 - find(s) returns a pointer to a const value associated with string key s, or
19   NULL if the key was not found.
20 - walk(f) calls the function f(s, v) on each entry of the map, in
21   lexicographically ascending order of the string keys.
22
23 Time Complexity:
24 - O(n) per call to insert(s, v), erase(s), and find(s), where n is the length of
25   s. Note that there is a hidden factor of log(n) due to map lookups, which can
26   be considered constant amortized. The implementation may be optimized by
27   storing the children of nodes in an std::unordered_map in C++11 and later, or
28   an std::vector< pair<string, node_t*> >, since the only container operations
29   required are iteration over the (key, child) pairs and inserting a new pair.
30   Sticking with an (ordered) std::map, we can optimize all operations by using
```

```

31     map.lower_bound(), a binary tree search for a child with a shared prefix,
32     instead of iteration.
33 - O(1) per call to walk(), where l is the total length of string keys that are
34     currently in the map.
35 - O(1) per call to all other operations.
36
37 Space Complexity:
38 - O(l) for storage of the radix tree, where l is the total length of string keys
39     that are currently in the map.
40 - O(n) auxiliary stack space for construction, destruction, walk(), where n is
41     the maximum length of any string that has been inserted so far.
42 - O(n) auxiliary stack space for erase(s), where n is the length of s.
43 - O(1) auxiliary for all other operations.
44 */
45
46
47 #include <cstddef>
48 #include <map>
49 #include <string>
50 #include <utility>
51 using std::string;
52
53 template<class V>
54 class radix_tree {
55     struct node_t {
56         V value;
57         bool is_terminal;
58         std::map<string, node_t*> children;
59
60         node_t(const V &value = V(), bool is_terminal = false)
61             : value(value), is_terminal(is_terminal) {}
62     } *root;
63
64     typedef typename std::map<string, node_t*>::iterator cit;
65
66     static int lcp_len(const string &s1, const string &s2, int s2start) {
67         int i = 0;
68         for (int j = s2start; i < (int)s1.size() && j < (int)s2.size(); i++, j++) {
69             if (s1[i] != s2[j]) {
70                 break;
71             }
72         }
73         return i;
74     }
75
76     static bool insert(node_t *n, const string &s, int i, const V &v) {
77         if (i == (int)s.size()) {
78             if (n->is_terminal) {
79                 return false;
80             }
81             n->is_terminal = true;
82             return true;
83         }
84         for (cit it = n->children.begin(); it != n->children.end(); ++it) {
85             int len = lcp_len(it->first, s, i);
86             if (len == 0) {
87                 continue;
88             }
89             if (len == (int)it->first.size()) {

```

```

90         return insert(it->second, s, i + len, v);
91     }
92     string left = it->first.substr(0, len);
93     string right = it->first.substr(len);
94     node_t *tmp = new node_t();
95     tmp->children[right] = it->second;
96     n->children.erase(it);
97     n->children[left] = tmp;
98     if (len == (int)s.size() - i) {
99         tmp->value = v;
100        tmp->is_terminal = true;
101        return true;
102    }
103    return insert(tmp, s, i + len, v);
104}
105n->children[s.substr(i)] = new node_t(v, true);
106return true;
107}
108
109static bool erase(node_t *n, const string &s, int i) {
110    if (i == (int)s.size()) {
111        if (!n->is_terminal) {
112            return false;
113        }
114        n->is_terminal = false;
115        return true;
116    }
117    for (cit it = n->children.begin(); it != n->children.end(); ++it) {
118        int len = lcp_len(it->first, s, i);
119        if (len == 0) {
120            continue;
121        }
122        node_t *child = it->second;
123        if (!erase(child, s, i + len)) {
124            return false;
125        }
126        if (child->children.empty()) {
127            delete child;
128            n->children.erase(it);
129        } else if (child->children.size() == 1) {
130            node_t *grandchild = child->children.begin()->second;
131            if (!child->is_terminal) {
132                string merged_key(it->first + child->children.begin()->first);
133                child->value = grandchild->value;
134                child->is_terminal = grandchild->is_terminal;
135                child->children = grandchild->children;
136                delete grandchild;
137                n->children.erase(it);
138                n->children[merged_key] = child;
139            }
140        }
141        return true;
142    }
143    return false;
144}
145
146template<class KVFunction>
147static void walk(node_t *n, string &s, KVFunction f) {
148    if (n->is_terminal) {

```

```

149     f(s, n->value);
150 }
151 for (cit it = n->children.begin(); it != n->children.end(); ++it) {
152     s += it->first;
153     walk(it->second, s, f);
154     s.pop_back();
155 }
156 }
157
158 static void clean_up(node_t *n) {
159     for (cit it = n->children.begin(); it != n->children.end(); ++it) {
160         clean_up(it->second);
161     }
162     delete n;
163 }
164
165 int num_terminals;
166
167 public:
168     radix_tree() : root(new node_t()), num_terminals(0) {}
169
170     ~radix_tree() {
171         clean_up(root);
172     }
173
174     int size() const {
175         return num_terminals;
176     }
177
178     bool empty() const {
179         return num_terminals == 0;
180     }
181
182     bool insert(const string &s, const V &v) {
183         if (insert(root, s, 0, v)) {
184             num_terminals++;
185             return true;
186         }
187         return false;
188     }
189
190     bool erase(const string &s) {
191         if (erase(root, s, 0)) {
192             num_terminals--;
193             return true;
194         }
195         return false;
196     }
197
198     const V* find(const string &s) const {
199         node_t *n = root;
200         int i = 0;
201         while (i < (int)s.size()) {
202             bool found = false;;
203             for (cit it = n->children.begin(); it != n->children.end(); ++it) {
204                 if (it->first[0] == s[i]) {
205                     i += lcp_len(it->first, s, i);
206                     n = it->second;
207                     found = true;

```

```

208         break;
209     }
210   }
211   if (!found) {
212     return NULL;
213   }
214 }
215 return n->is_terminal ? &(n->value) : NULL;
216 }
217
218 template<class KVFunction>
219 void walk(KVFunction f) const {
220   string s = "";
221   walk(root, s, f);
222 }
223 };
224
225 /** Example Usage and Output:
226
227 ("", 0)
228 ("a", 1)
229 ("i", 6)
230 ("in", 7)
231 ("inn", 8)
232 ("tea", 3)
233 ("ted", 4)
234 ("ten", 5)
235 ("to", 2)
236
237 */
238
239 #include <cassert>
240 #include <iostream>
241 using namespace std;
242
243 void print_entry(const string &k, int v) {
244   cout << "(" << k << "\", " << v << ")" << endl;
245 }
246
247 int main() {
248   string s[9] = {"", "a", "to", "tea", "ted", "ten", "i", "in", "inn"};
249   radix_tree<int> t;
250   assert(t.empty());
251   for (int i = 0; i < 9; i++) {
252     assert(t.insert(s[i], i));
253   }
254   t.walk(print_entry);
255   assert(!t.empty());
256   assert(t.size() == 9);
257   assert(!t.insert(s[0], 2));
258   assert(t.size() == 9);
259   assert(t.find("") && *t.find("") == 0);
260   assert(*t.find("ten") == 5);
261   assert(t.erase("tea"));
262   assert(t.size() == 8);
263   assert(t.find("tea") == NULL);
264   assert(t.erase(""));
265   assert(t.find("") == NULL);
266   return 0;

```

267 }

Chapter 4

Graphs

4.1 Depth-First Search

4.1.1 Graph Class and Depth-First Search

```
1  /*
2
3  A graph consists of a set of objects (a.k.a vertices, or nodes) and a set of
4  connections (a.k.a. edges) between pairs of said objects. A graph may be stored
5  as an adjacency list, which is a space efficient representation that is also
6  time-efficient for traversals.
7
8  The following class implements a simple graph using adjacency lists, along with
9  depth-first search and a few other applications. The constructor takes a Boolean
10 argument which specifies whether the instance is a directed or undirected graph.
11 The nodes of the graph are identified by integers indices numbered consecutively
12 starting from 0. The total number of nodes will automatically increase based on
13 the maximum node index passed to add_edge() so far.
14
15 Time Complexity:
16 - O(1) amortized per call to add_edge(), or O(max(n, m)) for n calls where the
17   maximum node index passed as an argument is m.
18 - O(max(n, m)) per call for dfs(), has_cycle(), is_tree(), or is_dag(), where n
19   is the number of nodes and m is the number of edges.
20 - O(1) per call to all other public member functions.
21
22 Space Complexity:
23 - O(max(n, m)) for storage of the graph, where n is the number of nodes and m
24   is the number of edges.
25 - O(n) auxiliary stack space for dfs(), has_cycle(), is_tree(), and is_dag().
26 - O(1) auxiliary for all other public member functions.
27
28 */
29
30 #include <algorithm>
31 #include <vector>
32
33 class graph {
34     std::vector<std::vector<int>> adj;
```

```

35     bool directed;
36
37     template<class ReportFunction>
38     void dfs(int n, std::vector<bool> &visit, ReportFunction f) const {
39         f(n);
40         visit[n] = true;
41         std::vector<int>::const_iterator it;
42         for (it = adj[n].begin(); it != adj[n].end(); ++it) {
43             if (!visit[*it]) {
44                 dfs(*it, visit, f);
45             }
46         }
47     }
48
49     bool has_cycle(int n, int prev, std::vector<bool> &visit,
50                     std::vector<bool> &onstack) const {
51         visit[n] = true;
52         onstack[n] = true;
53         std::vector<int>::const_iterator it;
54         for (it = adj[n].begin(); it != adj[n].end(); ++it) {
55             if (directed && onstack[*it]) {
56                 return true;
57             }
58             if (!directed && visit[*it] && *it != prev) {
59                 return true;
60             }
61             if (!visit[*it] && has_cycle(*it, n, visit, onstack)) {
62                 return true;
63             }
64         }
65         onstack[n] = false;
66         return false;
67     }
68
69     public:
70     graph(bool directed = true) : directed(directed) {}
71
72     int nodes() const {
73         return (int)adj.size();
74     }
75
76     std::vector<int>& operator[](int n) {
77         return adj[n];
78     }
79
80     void add_edge(int u, int v) {
81         int n = adj.size();
82         if (u >= n || v >= n) {
83             adj.resize(std::max(u, v) + 1);
84         }
85         adj[u].push_back(v);
86         if (!directed) {
87             adj[v].push_back(u);
88         }
89     }
90
91     bool is_directed() const {
92         return directed;
93     }

```

```
94     bool has_cycle() const {
95         int n = adj.size();
96         std::vector<bool> visit(n, false), onstack(n, false);
97         for (int i = 0; i < n; i++) {
98             if (!visit[i] && has_cycle(i, -1, visit, onstack)) {
99                 return true;
100            }
101        }
102    }
103    return false;
104 }
105
106    bool is_tree() const {
107        return !directed && !has_cycle();
108    }
109
110    bool is_dag() const {
111        return directed && !has_cycle();
112    }
113
114    template<class ReportFunction>
115    void dfs(int start, ReportFunction f) const {
116        std::vector<bool> visit(adj.size(), false);
117        dfs(start, visit, f);
118    }
119 };
120
121 /** Example Usage and Output:
122 DFS order: 0 1 2 3 4 5 6 7 8 9 10 11
123
124 */
125
126
127 #include <cassert>
128 #include <iostream>
129 using namespace std;
130
131 void print(int n) {
132     cout << n << " ";
133 }
134
135 int main() {
136     {
137         graph g;
138         g.add_edge(0, 1);
139         g.add_edge(0, 6);
140         g.add_edge(0, 7);
141         g.add_edge(1, 2);
142         g.add_edge(1, 5);
143         g.add_edge(2, 3);
144         g.add_edge(2, 4);
145         g.add_edge(7, 8);
146         g.add_edge(7, 11);
147         g.add_edge(8, 9);
148         g.add_edge(8, 10);
149         cout << "DFS order: ";
150         g.dfs(0, print);
151         cout << endl;
152         assert(g[0].size() == 3);
```

```

153     assert(g.is_dag());
154     assert(!g.has_cycle());
155 }
156 {
157     graph tree(false);
158     tree.add_edge(0, 1);
159     tree.add_edge(0, 2);
160     tree.add_edge(1, 3);
161     tree.add_edge(1, 4);
162     assert(tree.is_tree());
163     assert(!tree.is_dag());
164     tree.add_edge(2, 3);
165     assert(!tree.is_tree());
166 }
167     return 0;
168 }
```

4.1.2 Topological Sorting (DFS)

```

1  /*
2
3 Given a directed acyclic graph, find one of possibly many orderings of the nodes
4 such that for every edge from node u to v, u comes before v in the ordering.
5 Depth-first search is used to traverse all nodes in post-order.
6
7 toposort(nodes) takes a directed graph stored as a global adjacency list with
8 nodes indexed from 0 to (nodes - 1) and assigns a valid topological ordering to
9 the global result vector. An error is thrown if the graph contains a cycle.
10
11 Time Complexity:
12 - O(max(n, m)) per call to toposort(), where n is the number of nodes and m is
13   the number of edges.
14
15 Space Complexity:
16 - O(max(n, m)) for storage of the graph, where n is the number of nodes and m
17   is the number of edges.
18 - O(n) auxiliary stack space for toposort().
19 */
20
21 #include <algorithm>
22 #include <stdexcept>
23 #include <vector>
24
25
26 const int MAXN = 100;
27 std::vector<int> adj[MAXN], res;
28 std::vector<bool> visit(MAXN), done(MAXN);
29
30 void dfs(int u) {
31     if (visit[u]) {
32         throw std::runtime_error("Not a directed acyclic graph.");
33     }
34     if (done[u]) {
35         return;
36     }
```

```

37     visit[u] = true;
38     for (int j = 0; j < (int)adj[u].size(); j++) {
39         dfs(adj[u][j]);
40     }
41     visit[u] = false;
42     done[u] = true;
43     res.push_back(u);
44 }
45
46 void toposort(int nodes) {
47     fill(visit.begin(), visit.end(), false);
48     fill(done.begin(), done.end(), false);
49     res.clear();
50     for (int i = 0; i < nodes; i++) {
51         if (!done[i]) {
52             dfs(i);
53         }
54     }
55     std::reverse(res.begin(), res.end());
56 }
57
58 /** Example Usage and Output:
59
60 The topological order: 2 1 0 4 3 7 6 5
61 */
62
63
64 #include <iostream>
65 using namespace std;
66
67 int main() {
68     adj[0].push_back(3);
69     adj[0].push_back(4);
70     adj[1].push_back(3);
71     adj[2].push_back(4);
72     adj[2].push_back(7);
73     adj[3].push_back(5);
74     adj[3].push_back(6);
75     adj[3].push_back(7);
76     adj[4].push_back(6);
77     toposort(8);
78     cout << "The topological order:";
79     for (int i = 0; i < (int)res.size(); i++) {
80         cout << " " << res[i];
81     }
82     cout << endl;
83     return 0;
84 }
```

4.1.3 Eulerian Cycles (DFS)

```

1 /*
2
3 A Eulerian trail is a path in a graph which contains every edge exactly once. An
4 Eulerian cycle or circuit is an Eulerian trail which begins and ends on the same
```

```

5 node. A directed graph has an Eulerian cycle if and only if every node has an
6 in-degree equal to its out-degree, and all of its nodes with nonzero degree
7 belong to a single strongly connected component. An undirected graph has an
8 Eulerian cycle if and only if every node has even degree, and all of its nodes
9 with nonzero degree belong to a single connected component.
10
11 Given a graph as an adjacency list along with the starting node of the cycle,
12 both functions below return a vector containing all nodes reachable from the
13 starting node in an order which forms an Eulerian cycle. The first node of the
14 cycle will be repeated as the last element of the vector. All nodes of input
15 adjacency lists to both functions must be between 0 and MAXN - 1, inclusive.
16 In addition, euler_cycle_undirected() requires that for every node v which is
17 found in adj[u], node u must also be found in adj[v].
18
19 Time Complexity:
20 - O(max(n, m)) per call to either function, where n and m are the numbers of
21 nodes and edges respectively.
22
23 Space Complexity:
24 - O(n) auxiliary heap space for euler_cycle_directed(), where n is the number of
25 nodes.
26 - O(n^2) auxiliary heap space for euler_cycle_undirected(), where n is the
27 number of nodes. This can be reduced to O(m) auxiliary heap space on the
28 number of edges if the used[][] bit matrix is replaced with an
29 std::unordered_set<std::pair<int, int>>.
30
31 */
32
33 #include <algorithm>
34 #include <bitset>
35 #include <vector>
36
37 const int MAXN = 100;
38
39 std::vector<int> euler_cycle_directed(std::vector<int> adj[], int u) {
40     std::vector<int> stack, curr_edge(MAXN), res;
41     stack.push_back(u);
42     while (!stack.empty()) {
43         u = stack.back();
44         stack.pop_back();
45         while (curr_edge[u] < (int)adj[u].size()) {
46             stack.push_back(u);
47             u = adj[u][curr_edge[u]++;
48         }
49         res.push_back(u);
50     }
51     std::reverse(res.begin(), res.end());
52     return res;
53 }
54
55 std::vector<int> euler_cycle_undirected(std::vector<int> adj[], int u) {
56     std::bitset<MAXN> used[MAXN];
57     std::vector<int> stack, curr_edge(MAXN), res;
58     stack.push_back(u);
59     while (!stack.empty()) {
60         u = stack.back();
61         stack.pop_back();
62         while (curr_edge[u] < (int)adj[u].size()) {
63             int v = adj[u][curr_edge[u]++;

```

```
64     int mn = std::min(u, v), mx = std::max(u, v);
65     if (!used[mn] [mx]) {
66         used[mn] [mx] = true;
67         stack.push_back(u);
68         u = v;
69     }
70 }
71 res.push_back(u);
72 }
73 std::reverse(res.begin(), res.end());
74 return res;
75 }
76
77 /** Example Usage and Output:
78
79 Eulerian cycle from 0 (directed): 0 1 3 4 1 2 0
80 Eulerian cycle from 2 (undirected): 2 1 3 4 1 0 2
81
82 */
83
84 #include <iostream>
85 using namespace std;
86
87 int main() {
88 {
89     vector<int> g[5], cycle;
90     g[0].push_back(1);
91     g[1].push_back(2);
92     g[2].push_back(0);
93     g[1].push_back(3);
94     g[3].push_back(4);
95     g[4].push_back(1);
96     cycle = euler_cycle_directed(g, 0);
97     cout << "Eulerian cycle from 0 (directed):";
98     for (int i = 0; i < (int)cycle.size(); i++) {
99         cout << " " << cycle[i];
100    }
101    cout << endl;
102 }
103 {
104     vector<int> g[5], cycle;
105     g[0].push_back(1);
106     g[1].push_back(0);
107     g[1].push_back(2);
108     g[2].push_back(1);
109     g[2].push_back(0);
110     g[0].push_back(2);
111     g[1].push_back(3);
112     g[3].push_back(1);
113     g[3].push_back(4);
114     g[4].push_back(3);
115     g[4].push_back(1);
116     g[1].push_back(4);
117     cycle = euler_cycle_undirected(g, 2);
118     cout << "Eulerian cycle from 2 (undirected):";
119     for (int i = 0; i < (int)cycle.size(); i++) {
120         cout << " " << cycle[i];
121    }
122    cout << endl;
```

```

123     }
124     return 0;
125 }
```

4.1.4 Unweighted Tree Centers (DFS)

```

1  /*
2
3  An unweighted tree possesses a center, centroid, and diameter. The following
4  functions apply to a global, pre-populated adjacency list adj[] which satisfies
5  the precondition that for every node v in adj[u], node u also exists in adj[v].
6  Nodes in adj[] must be numbered with integers between 0 (inclusive) and the
7  total number of nodes (exclusive), as passed in the function arguments.
8
9  - find_centers() returns a vector of either one or two tree Jordan centers. The
10   Jordan center of a tree is the set of all nodes with minimum eccentricity,
11   that is, the set of all nodes where the maximum distance to all other nodes in
12   the tree is minimal.
13 - find_centroid() returns the node where all of its subtrees have a size less
14   than or equal to n/2, where n is the number of nodes in the tree.
15 - diameter() returns the maximum distance between any two nodes in the tree,
16   using a well-known double depth-first search technique.
17
18 Time Complexity:
19 - O(max(n, m)) per call to find_centers(), find_centroid(), and diameter(),
20   where n is the number of nodes and m is the number of edges.
21
22 Space Complexity:
23 - O(n) auxiliary stack space for find_centers(), find_centroid(), and
24   diameter(), where n is the number of nodes.
25
26 */
27
28 #include <utility>
29 #include <vector>
30
31 const int MAXN = 100;
32 std::vector<int> adj[MAXN];
33
34 std::vector<int> find_centers(int nodes) {
35     std::vector<int> leaves, degree(nodes);
36     for (int i = 0; i < nodes; i++) {
37         degree[i] = adj[i].size();
38         if (degree[i] <= 1) {
39             leaves.push_back(i);
40         }
41     }
42     int removed = leaves.size();
43     while (removed < nodes) {
44         std::vector<int> nleaves;
45         for (int i = 0; i < (int)leaves.size(); i++) {
46             int u = leaves[i];
47             for (int j = 0; j < (int)adj[u].size(); j++) {
48                 int v = adj[u][j];
49                 if (--degree[v] == 1) {
```

```

50         nleaves.push_back(v);
51     }
52   }
53 }
54 leaves = nleaves;
55 removed += leaves.size();
56 }
57 return leaves;
58 }

59 int find_centroid(int nodes, int u = 0, int p = -1) {
60   int count = 1;
61   bool good_center = true;
62   for (int j = 0; j < (int)adj[u].size(); j++) {
63     int v = adj[u][j];
64     if (v == p) {
65       continue;
66     }
67     int res = find_centroid(nodes, v, u);
68     if (res >= 0) {
69       return res;
70     }
71     int size = -res;
72     good_center &= (size <= nodes / 2);
73     count += size;
74   }
75   good_center &= (nodes - count <= nodes / 2);
76   return good_center ? u : -count;
77 }
78 }

79 std::pair<int, int> dfs(int u, int p, int depth) {
80   std::pair<int, int> res = std::make_pair(depth, u);
81   for (int j = 0; j < (int)adj[u].size(); j++) {
82     if (adj[u][j] != p) {
83       res = max(res, dfs(adj[u][j], u, depth + 1));
84     }
85   }
86   return res;
87 }

88 }

89 int diameter() {
90   int furthest_node = dfs(0, -1, 0).second;
91   return dfs(furthest_node, -1, 0).first;
92 }

93 }

94 /**
95  * Example Usage */
96

97 #include <cassert>
98 using namespace std;
99

100 int main() {
101   int nodes = 6;
102   adj[0].push_back(1);
103   adj[1].push_back(0);
104   adj[1].push_back(2);
105   adj[2].push_back(1);
106   adj[1].push_back(4);
107   adj[4].push_back(1);
108   adj[3].push_back(4);

```

```

109     adj[4].push_back(3);
110     adj[4].push_back(5);
111     adj[5].push_back(4);
112     vector<int> centers = find_centers(nodes);
113     assert(centers.size() == 2 && centers[0] == 1 && centers[1] == 4);
114     assert(find_centroid(nodes) == 4);
115     assert(diameter() == 3);
116     return 0;
117 }

```

4.2 Shortest Path

4.2.1 Shortest Path (BFS)

```

1  /*
2
3 Given a starting node in an unweighted, directed graph, visit every connected
4 node and determine the minimum distance to each such node. Optionally, output
5 the shortest path to a specific destination node using the shortest-path tree
6 from the predecessor array pred[]. bfs() applies to a global, pre-populated
7 adjacency list adj[] which consists of only nodes numbered with integers between
8 0 (inclusive) and the total number of nodes (exclusive), as passed in the
9 function argument.
10
11 Time Complexity:
12 - O(n) per call to bfs(), where n is the number of nodes.
13
14 Space Complexity:
15 - O(max(n, m)) for storage of the graph, where n is the number of nodes and m
16   is the number of edges.
17 - O(n) auxiliary heap space for bfs().
18
19 */
20
21 #include <queue>
22 #include <utility>
23 #include <vector>
24
25 const int MAXN = 100, INF = 0x3f3f3f3f;
26 std::vector<int> adj[MAXN];
27 int dist[MAXN], pred[MAXN];
28
29 void bfs(int nodes, int start) {
30     std::vector<bool> visit(nodes, false);
31     for (int i = 0; i < nodes; i++) {
32         dist[i] = INF;
33         pred[i] = -1;
34     }
35     std::queue<std::pair<int, int> > q;
36     q.push(std::make_pair(start, 0));
37     while (!q.empty()) {
38         int u = q.front().first;
39         int d = q.front().second;

```

```

40     q.pop();
41     visit[u] = true;
42     for (int j = 0; j < (int)adj[u].size(); j++) {
43         int v = adj[u][j];
44         if (visit[v]) {
45             continue;
46         }
47         dist[v] = d + 1;
48         pred[v] = u;
49         q.push(std::make_pair(v, d + 1));
50     }
51 }
52 }
53
54 /** Example Usage and Output:
55
56 The shortest distance from 0 to 3 is 2.
57 Take the path: 0->1->3.
58
59 */
60
61 #include <iostream>
62 using namespace std;
63
64 void print_path(int dest) {
65     vector<int> path;
66     for (int j = dest; pred[j] != -1; j = pred[j]) {
67         path.push_back(pred[j]);
68     }
69     cout << "Take the path: ";
70     while (!path.empty()) {
71         cout << path.back() << "->";
72         path.pop_back();
73     }
74     cout << dest << "." << endl;
75 }
76
77 int main() {
78     int start = 0, dest = 3;
79     adj[0].push_back(1);
80     adj[0].push_back(3);
81     adj[1].push_back(2);
82     adj[1].push_back(3);
83     adj[2].push_back(3);
84     adj[0].push_back(3);
85     bfs(4, start);
86     cout << "The shortest distance from " << start << " to " << dest << " is "
87         << dist[dest] << "." << endl;
88     print_path(dest);
89     return 0;
90 }

```

4.2.2 Shortest Path (Dijkstra)

```

1  /*

```

```

2
3 Given a starting node in a weighted, directed graph with nonnegative weights
4 only, visit every connected node and determine the minimum distance to each such
5 node. Optionally, output the shortest path to a specific destination node using
6 the shortest-path tree from the predecessor array pred[]. dijkstra() applies to
7 a global, pre-populated adjacency list adj[] which must only consist of nodes
8 numbered with integers between 0 (inclusive) and the total number of nodes
9 (exclusive), as passed in the function argument.
10
11 Since std::priority_queue is by default a max-heap, we simulate a min-heap by
12 negating node distances before pushing them and negating them again after
13 popping them. Alternatively, the container can be declared with the following
14 template arguments (#include <functional> to access std::greater):
15     priority_queue<pair<int, int>, vector<pair<int, int> >,
16             greater<pair<int, int> > pq;
17
18 Dijkstra's algorithm may be modified to support negative edge weights by
19 allowing nodes to be re-visited (removing the visited array check in the inner
20 for-loop). This is known as the Shortest Path Faster Algorithm (SPFA), which has
21 a larger running time of  $O(n*m)$  on the number of nodes and edges respectively.
22 While it is as slow in the worst case as the Bellman-Ford algorithm, the SPFA
23 still tends to outperform in the average case.
24
25 Time Complexity:
26 -  $O(m \log n)$  for dijkstra(), where m is the number of edges and n is the number
27 of nodes.
28
29 Space Complexity:
30 -  $O(\max(n, m))$  for storage of the graph, where n is the number of nodes and m
31 is the number of edges.
32 -  $O(n)$  auxiliary heap space for dijkstra().
33
34 */
35
36 #include <queue>
37 #include <utility>
38 #include <vector>
39
40 const int MAXN = 100, INF = 0x3f3f3f3f;
41 std::vector<std::pair<int, int> > adj[MAXN];
42 int dist[MAXN], pred[MAXN];
43
44 void dijkstra(int nodes, int start) {
45     std::vector<bool> visit(nodes, false);
46     for (int i = 0; i < nodes; i++) {
47         dist[i] = INF;
48         pred[i] = -1;
49     }
50     dist[start] = 0;
51     std::priority_queue<std::pair<int, int> > pq;
52     pq.push(std::make_pair(0, start));
53     while (!pq.empty()) {
54         int u = pq.top().second;
55         pq.pop();
56         visit[u] = true;
57         for (int j = 0; j < (int)adj[u].size(); j++) {
58             int v = adj[u][j].first;
59             if (visit[v]) {
60                 continue;
61             }
62             if (dist[v] > dist[u] + adj[u][j].second) {
63                 dist[v] = dist[u] + adj[u][j].second;
64                 pred[v] = u;
65                 pq.push(std::make_pair(dist[v], v));
66             }
67         }
68     }
69 }
```

```

61     }
62     if (dist[v] > dist[u] + adj[u][j].second) {
63         dist[v] = dist[u] + adj[u][j].second;
64         pred[v] = u;
65         pq.push(std::make_pair(-dist[v], v));
66     }
67 }
68 }
69 }

70 /**
71 *** Example Usage and Output:
72
73 The shortest distance from 0 to 3 is 5.
74 Take the path: 0->1->2->3.
75
76 ***
77
78 #include <iostream>
79 using namespace std;
80
81 void print_path(int dest) {
82     vector<int> path;
83     for (int j = dest; pred[j] != -1; j = pred[j]) {
84         path.push_back(pred[j]);
85     }
86     cout << "Take the path: ";
87     while (!path.empty()) {
88         cout << path.back() << "->";
89         path.pop_back();
90     }
91     cout << dest << "." << endl;
92 }
93
94 int main() {
95     int start = 0, dest = 3;
96     adj[0].push_back(make_pair(1, 2));
97     adj[0].push_back(make_pair(3, 8));
98     adj[1].push_back(make_pair(2, 2));
99     adj[1].push_back(make_pair(3, 4));
100    adj[2].push_back(make_pair(3, 1));
101    dijkstra(4, start);
102    cout << "The shortest distance from " << start << " to " << dest << " is "
103        << dist[dest] << "." << endl;
104    print_path(dest);
105    return 0;
106 }
```

4.2.3 Shortest Path (Bellman-Ford)

```

1 /*
2
3 Given a starting node in a weighted, directed graph with possibly negative
4 weights, visit every connected node and determine the minimum distance to each
5 such node. Optionally, output the shortest path to a specific destination node
6 using the shortest-path tree from the predecessor array pred[]. bellman_ford()
```

```

7 applies to a global, pre-populated edge list which must only consist of nodes
8 numbered with integers between 0 (inclusive) and the total number of nodes
9 (exclusive), as passed in the function argument.
10
11 This function will also detect whether the graph contains negative-weighted
12 cycles, in which case there is no shortest path and an error will be thrown.
13
14 Time Complexity:
15 -  $O(n*m)$  per call to bellman_ford(), where n is the number of nodes and m is the
16 number of edges.
17
18 Space Complexity:
19 -  $O(\max(n, m))$  for storage of the graph, where n is the number of nodes and m is
20 the number of edges.
21 -  $O(n)$  auxiliary heap space for bellman_ford(), where n is the number of nodes.
22
23 */
24
25 #include <stdexcept>
26 #include <vector>
27
28 struct edge { int u, v, w; }; // Edge from u to v with weight w.
29
30 const int MAXN = 100, INF = 0x3f3f3f3f;
31 std::vector<edge> e;
32 int dist[MAXN], pred[MAXN];
33
34 void bellman_ford(int nodes, int start) {
35     for (int i = 0; i < nodes; i++) {
36         dist[i] = INF;
37         pred[i] = -1;
38     }
39     dist[start] = 0;
40     for (int i = 0; i < nodes; i++) {
41         for (int j = 0; j < (int)e.size(); j++) {
42             if (dist[e[j].v] > dist[e[j].u] + e[j].w) {
43                 dist[e[j].v] = dist[e[j].u] + e[j].w;
44                 pred[e[j].v] = e[j].u;
45             }
46         }
47     }
48     // Optional: Report negative-weighted cycles.
49     for (int i = 0; i < (int)e.size(); i++) {
50         if (dist[e[i].v] > dist[e[i].u] + e[i].w) {
51             throw std::runtime_error("Negative-weight cycle found.");
52         }
53     }
54 }
55
56 /** Example Usage and Output:
57
58 The shortest distance from 0 to 2 is 3.
59 Take the path: 0->1->2.
60
61 ***/
62
63 #include <iostream>
64 using namespace std;
65

```

```

66 void print_path(int dest) {
67     vector<int> path;
68     for (int j = dest; pred[j] != -1; j = pred[j]) {
69         path.push_back(pred[j]);
70     }
71     cout << "Take the path: ";
72     while (!path.empty()) {
73         cout << path.back() << "->";
74         path.pop_back();
75     }
76     cout << dest << "." << endl;
77 }
78
79 int main() {
80     int start = 0, dest = 2;
81     e.push_back((edge){0, 1, 1});
82     e.push_back((edge){1, 2, 2});
83     e.push_back((edge){0, 2, 5});
84     bellman_ford(3, start);
85     cout << "The shortest distance from " << start << " to " << dest << " is "
86         << dist[dest] << "." << endl;
87     print_path(dest);
88     return 0;
89 }
```

4.2.4 Shortest Path (Floyd-Warshall)

```

1 /*
2
3 Given a weighted, directed graph with possibly negative weights, determine the
4 minimum distance between all pairs of start and destination nodes in the graph.
5 Optionally, output the shortest path between two nodes using the shortest-path
6 tree precomputed into the parent[][] array. floyd_marshall() applies to a global
7 adjacency matrix dist[][][], which must be initialized using initialize() and
8 subsequently populated with weights. After the function call, dist[u][v] will
9 have been modified to contain the shortest path from u to v, for all pairs of
10 valid nodes u and v.
11
12 This function will also detect whether the graph contains negative-weighted
13 cycles, in which case there is no shortest path and an error will be thrown.
14
15 Time Complexity:
16 - O(n^2) per call to initialize(), where n is the number of nodes.
17 - O(n^3) per call to floyd_marshall().
18
19 Space Complexity:
20 - O(n^2) for storage of the graph, where n is the number of nodes.
21 - O(n^2) auxiliary heap space for initialize() and floyd_marshall().
22
23 */
24
25 #include <stdexcept>
26
27 const int MAXN = 100, INF = 0x3f3f3f3f;
28 int dist[MAXN][MAXN], parent[MAXN][MAXN];
```

```

29
30 void initialize(int nodes) {
31     for (int i = 0; i < nodes; i++) {
32         for (int j = 0; j < nodes; j++) {
33             dist[i][j] = (i == j) ? 0 : INF;
34             parent[i][j] = j;
35         }
36     }
37 }
38
39 void floyd_marshall(int nodes) {
40     for (int k = 0; k < nodes; k++) {
41         for (int i = 0; i < nodes; i++) {
42             for (int j = 0; j < nodes; j++) {
43                 if (dist[i][j] > dist[i][k] + dist[k][j]) {
44                     dist[i][j] = dist[i][k] + dist[k][j];
45                     parent[i][j] = parent[i][k];
46                 }
47             }
48         }
49     }
50     // Optional: Report negative-weighted cycles.
51     for (int i = 0; i < nodes; i++) {
52         if (dist[i][i] < 0) {
53             throw std::runtime_error("Negative-weight cycle found.");
54         }
55     }
56 }
57
58 /**
59 * Example Usage and Output:
60 The shortest distance from 0 to 2 is 3.
61 Take the path: 0->1->2.
62
63 */
64
65 #include <iostream>
66 using namespace std;
67
68 void print_path(int u, int v) {
69     cout << "Take the path " << u;
70     while (u != v) {
71         u = parent[u][v];
72         cout << "->" << u;
73     }
74     cout << "." << endl;
75 }
76
77 int main() {
78     initialize(3);
79     int start = 0, dest = 2;
80     dist[0][1] = 1;
81     dist[1][2] = 2;
82     dist[0][2] = 5;
83     floyd_marshall(3);
84     cout << "The shortest distance from " << start << " to " << dest << " is "
85         << dist[start][dest] << "." << endl;
86     print_path(start, dest);
87     return 0;

```

88 }

4.3 Connectivity

4.3.1 Strongly Connected Components (Kosaraju)

```

1  /*
2
3 Given a directed graph, determine the strongly connected components, that is,
4 the set of all strongly (maximally) connected subgraphs. A subgraph is strongly
5 connected if there is a path between each pair of nodes. Condensing the strongly
6 connected components of a graph into single nodes will result in a directed
7 acyclic graph. kosaraju() applies to a global, pre-populated adjacency list
8 adj[] which must only consist of nodes numbered with integers between 0
9 (inclusive) and the total number of nodes (exclusive), as passed in the function
10 argument.
11
12 Time Complexity:
13 - O(max(n, m)) per call to kosaraju(), where n is the number of nodes and m is
14 the number of edges.
15
16 Space Complexity:
17 - O(max(n, m)) auxiliary heap space for storage of the graph, where n the number
18 of nodes and m is the number of edges.
19 - O(n) auxiliary stack space for kosaraju().
20
21 */
22
23 #include <algorithm>
24 #include <vector>
25
26 const int MAXN = 100;
27 std::vector<int> adj[MAXN], rev[MAXN];
28 std::vector<bool> visit(MAXN);
29 std::vector<std::vector<int>> scc;
30
31 void dfs(std::vector<int> g[], std::vector<int> &res, int u) {
32     visit[u] = true;
33     for (int j = 0; j < (int)g[u].size(); j++) {
34         if (!visit[g[u][j]]) {
35             dfs(g, res, g[u][j]);
36         }
37     }
38     res.push_back(u);
39 }
40
41 void kosaraju(int nodes) {
42     std::fill(visit.begin(), visit.end(), false);
43     std::vector<int> order;
44     for (int i = 0; i < nodes; i++) {
45         rev[i].clear();
46         if (!visit[i]) {
47             dfs(adj, order, i);

```

```

48      }
49  }
50  std::reverse(order.begin(), order.end());
51  std::fill(visit.begin(), visit.end(), false);
52  for (int i = 0; i < nodes; i++) {
53    for (int j = 0; j < (int)adj[i].size(); j++) {
54      rev[adj[i][j]].push_back(i);
55    }
56  }
57  scc.clear();
58  for (int i = 0; i < (int)order.size(); i++) {
59    if (visit[order[i]]) {
60      continue;
61    }
62    std::vector<int> component;
63    dfs(rev, component, order[i]);
64    scc.push_back(component);
65  }
66 }
67
68 /** Example Usage and Output:
69
70 1 4 0
71 7 3 2
72 5 6
73
74 */
75
76 #include <iostream>
77 using namespace std;
78
79 int main() {
80  adj[0].push_back(1);
81  adj[1].push_back(2);
82  adj[1].push_back(4);
83  adj[1].push_back(5);
84  adj[2].push_back(3);
85  adj[2].push_back(6);
86  adj[3].push_back(2);
87  adj[3].push_back(7);
88  adj[4].push_back(0);
89  adj[4].push_back(5);
90  adj[5].push_back(6);
91  adj[6].push_back(5);
92  adj[7].push_back(3);
93  adj[7].push_back(6);
94  kosaraju(8);
95  cout << "Components:" << endl;
96  for (int i = 0; i < (int)scc.size(); i++) {
97    for (int j = 0; j < (int)scc[i].size(); j++) {
98      cout << scc[i][j] << " ";
99    }
100   cout << endl;
101 }
102 return 0;
103 }
```

4.3.2 Strongly Connected Components (Tarjan)

```

1  /*
2
3 Given a directed graph, determine the strongly connected components. The
4 strongly connected components of a graph is the set of all strongly (maximally)
5 connected subgraphs. A subgraph is strongly connected if there is a path between
6 each pair of nodes. Condensing the strongly connected components of a graph into
7 single nodes will result in a directed acyclic graph. tarjan() applies to a
8 global, pre-populated adjacency list adj[] which must only consist of nodes
9 numbered with integers between 0 (inclusive) and the total number of nodes
10 (exclusive), as passed in the function argument.
11
12 Time Complexity:
13 - O(max(n, m)) per call to tarjan(), where n is the number of nodes and m is the
14 number of edges.
15
16 Space Complexity:
17 - O(max(n, m)) for storage of the graph, where n the number of nodes and m is
18 the number of edges.
19 - O(n) auxiliary stack space for tarjan().
20
21 */
22
23 #include <algorithm>
24 #include <vector>
25
26 const int MAXN = 100, INF = 0x3f3f3f3f;
27 std::vector<int> adj[MAXN], stack;
28 int timer, lowlink[MAXN];
29 std::vector<bool> visit(MAXN);
30 std::vector<std::vector<int>> scc;
31
32 void dfs(int u) {
33     lowlink[u] = timer++;
34     visit[u] = true;
35     stack.push_back(u);
36     bool is_component_root = true;
37     int v;
38     for (int j = 0; j < (int)adj[u].size(); j++) {
39         v = adj[u][j];
40         if (!visit[v]) {
41             dfs(v);
42         }
43         if (lowlink[u] > lowlink[v]) {
44             lowlink[u] = lowlink[v];
45             is_component_root = false;
46         }
47     }
48     if (!is_component_root) {
49         return;
50     }
51     std::vector<int> component;
52     do {
53         v = stack.back();
54         visit[v] = true;
55         stack.pop_back();

```

```

56     lowlink[v] = INF;
57     component.push_back(v);
58 } while (u != v);
59 scc.push_back(component);
60 }
61
62 void tarjan(int nodes) {
63     scc.clear();
64     stack.clear();
65     std::fill(lowlink, lowlink + nodes, 0);
66     std::fill(visit.begin(), visit.end(), false);
67     timer = 0;
68     for (int i = 0; i < nodes; i++) {
69         if (!visit[i]) {
70             dfs(i);
71         }
72     }
73 }
74
75 /** Example Usage and Output:
76
77 Components:
78 5 6
79 7 3 2
80 4 1 0
81
82 */
83
84 #include <iostream>
85 using namespace std;
86
87 int main() {
88     adj[0].push_back(1);
89     adj[1].push_back(2);
90     adj[1].push_back(4);
91     adj[1].push_back(5);
92     adj[2].push_back(3);
93     adj[2].push_back(6);
94     adj[3].push_back(2);
95     adj[3].push_back(7);
96     adj[4].push_back(0);
97     adj[4].push_back(5);
98     adj[5].push_back(6);
99     adj[6].push_back(5);
100    adj[7].push_back(3);
101    adj[7].push_back(6);
102    tarjan(8);
103    cout << "Components:" << endl;
104    for (int i = 0; i < (int)scc.size(); i++) {
105        for (int j = 0; j < (int)scc[i].size(); j++) {
106            cout << scc[i][j] << " ";
107        }
108        cout << endl;
109    }
110    return 0;
111 }
```

4.3.3 Bridges, Cut-points, and Biconnectivity

```

1  /*
2
3 Given an undirected graph, compute the following properties of the graph using
4 Tarjan's algorithm. tarjan() applies to a global, pre-populated adjacency list
5 adj[] which satisfies the precondition that for every node v in adj[u], node u
6 also exists in adj[v]. Nodes in adj[] must be numbered with integers between 0
7 (inclusive) and the total number of nodes (exclusive), as passed in the function
8 arguments. get_block_forest() applies to the global vector of biconnected
9 components bcc[] which must have already been precomputed by a call to tarjan().
10
11 A bridge is an edge such that when deleted, the number of connected components
12 in the graph is increased. An edge is a bridge if and only if it is not part of
13 any cycle.
14
15 A cut-point (i.e. cut-node, or articulation point) is any node whose removal
16 increases the number of connected components in the graph.
17
18 A biconnected component of a graph is a maximally biconnected subgraph. A
19 biconnected graph is a connected and "non-separable" graph, meaning that if any
20 node were to be removed, the graph will remain connected. Thus, a biconnected
21 graph has no articulation points.
22
23 Any connected graph decomposes into a tree of biconnected components called the
24 "block tree" of the graph. An unconnected graph will thus decompose into a
25 "block forest."
26
27 Time Complexity:
28 - O(max(n, m)) per call to tarjan() and get_block_forest(), where n is the
29   number of nodes and m is the number of edges.
30
31 Space Complexity:
32 - O(max(n, m)) for storage of the graph, where n the number of nodes and m is
33   the number of edges
34 - O(n) auxiliary stack space for tarjan().
35 - O(1) auxiliary stack space for get_block_forest().
36
37 */
38
39 #include <algorithm>
40 #include <vector>
41
42 const int MAXN = 100;
43 int timer, lowlink[MAXN], tin[MAXN], comp[MAXN];
44 std::vector<bool> visit(MAXN);
45 std::vector<int> adj[MAXN], bcc_forest[MAXN];
46 std::vector<int> stack, cutpoints;
47 std::vector<std::vector<int>> bcc;
48 std::vector<std::pair<int, int>> bridges;
49
50 void dfs(int u, int p) {
51     visit[u] = true;
52     lowlink[u] = tin[u] = timer++;
53     stack.push_back(u);
54     int v, children = 0;
55     bool cutpoint = false;

```

```

56     for (int j = 0; j < (int)adj[u].size(); j++) {
57         v = adj[u][j];
58         if (v == p) {
59             continue;
60         }
61         if (visit[v]) {
62             lowlink[u] = std::min(lowlink[u], tin[v]);
63         } else {
64             dfs(v, u);
65             lowlink[u] = std::min(lowlink[u], lowlink[v]);
66             cutpoint |= (lowlink[v] >= tin[u]);
67             if (lowlink[v] > tin[u]) {
68                 bridges.push_back(std::make_pair(u, v));
69             }
70             children++;
71         }
72     }
73     if (p == -1) {
74         cutpoint = (children >= 2);
75     }
76     if (cutpoint) {
77         cutpoints.push_back(u);
78     }
79     if (lowlink[u] == tin[u]) {
80         std::vector<int> component;
81         do {
82             v = stack.back();
83             stack.pop_back();
84             component.push_back(v);
85         } while (u != v);
86         bcc.push_back(component);
87     }
88 }
89
90 void tarjan(int nodes) {
91     bcc.clear();
92     bridges.clear();
93     cutpoints.clear();
94     stack.clear();
95     std::fill(lowlink, lowlink + nodes, 0);
96     std::fill(tin, tin + nodes, 0);
97     std::fill(visit.begin(), visit.end(), false);
98     timer = 0;
99     for (int i = 0; i < nodes; i++) {
100         if (!visit[i]) {
101             dfs(i, -1);
102         }
103     }
104 }
105
106 void get_block_forest(int nodes) {
107     std::fill(comp, comp + nodes, 0);
108     for (int i = 0; i < nodes; i++) {
109         bcc_forest[i].clear();
110     }
111     for (int i = 0; i < (int)bcc.size(); i++) {
112         for (int j = 0; j < (int)bcc[i].size(); j++) {
113             comp[bcc[i][j]] = i;
114         }

```

```
115     }
116     for (int i = 0; i < nodes; i++) {
117         for (int j = 0; j < (int)adj[i].size(); j++) {
118             if (comp[i] != comp[adj[i][j]]) {
119                 bcc_forest[comp[i]].push_back(comp[adj[i][j]]);
120             }
121         }
122     }
123 }
124
125 /** Example Usage and Output:
126
127 Cut-points: 5 1
128 Bridges:
129 1 2
130 5 4
131 3 7
132 Edge-Biconnected Components:
133 2
134 4
135 5 1 0
136 7
137 3
138 6
139 Adjacency List for Block Forest:
140 0 => 2
141 1 => 2
142 2 => 0 1
143 3 => 4
144 4 => 3
145 5 =>
146
147 ***/
148
149 #include <iostream>
150 using namespace std;
151
152 void add_edge(int u, int v) {
153     adj[u].push_back(v);
154     adj[v].push_back(u);
155 }
156
157 int main() {
158     add_edge(0, 1);
159     add_edge(0, 5);
160     add_edge(1, 2);
161     add_edge(1, 5);
162     add_edge(3, 7);
163     add_edge(4, 5);
164     tarjan(8);
165     get_block_forest(8);
166     cout << "Cut-points:";
167     for (int i = 0; i < (int)cutpoints.size(); i++) {
168         cout << " " << cutpoints[i];
169     }
170     cout << endl << "Bridges:" << endl;
171     for (int i = 0; i < (int)bridges.size(); i++) {
172         cout << bridges[i].first << " " << bridges[i].second << endl;
173 }
```

```

174     cout << "Edge-Biconnected Components:" << endl;
175     for (int i = 0; i < (int)bcc.size(); i++) {
176         for (int j = 0; j < (int)bcc[i].size(); j++) {
177             cout << bcc[i][j] << " ";
178         }
179         cout << endl;
180     }
181     cout << "Adjacency List for Block Forest:" << endl;
182     for (int i = 0; i < (int)bcc.size(); i++) {
183         cout << i << " =>";
184         for (int j = 0; j < (int)bcc_forest[i].size(); j++) {
185             cout << " " << bcc_forest[i][j];
186         }
187         cout << endl;
188     }
189     return 0;
190 }
```

4.4 Minimum Spanning Tree

4.4.1 Minimum Spanning Tree (Prim)

```

1  /*
2
3 Given a connected, undirected, weighted graph with possibly negative weights,
4 its minimum spanning tree is a subgraph which is a tree that connects all nodes
5 with a subset of its edges such that their total weight is minimized. prim()
6 applies to a global, pre-populated adjacency list adj[] which must only consist
7 of nodes numbered with integers between 0 (inclusive) and the total number of
8 nodes (exclusive), as passed in the function argument. If the input graph is not
9 connected, then this implementation will find the minimum spanning forest.
10
11 Since std::priority_queue is by default a max-heap, we simulate a min-heap by
12 negating node distances before pushing them and negating them again after
13 popping them. To modify this implementation to find the maximum spanning tree,
14 the two negation steps can be skipped to prioritize the max edges.
15
16 Time Complexity:
17 - O(m log n) per call to prim(), where m is the number of edges and n is the
18   number of nodes.
19
20 Space Complexity:
21 - O(max(n, m)) for storage of the graph, where n the number of nodes and m is
22   the number of edges.
23 - O(n) auxiliary heap space for prim().
24
25 */
26
27 #include <queue>
28 #include <utility>
29 #include <vector>
30
31 const int MAXN = 100;
```

```

32 std::vector<std::pair<int, int> > adj[MAXN], mst;
33
34 int prim(int nodes) {
35     mst.clear();
36     std::vector<bool> visit(nodes);
37     int total_dist = 0;
38     for (int i = 0; i < nodes; i++) {
39         if (visit[i]) {
40             continue;
41         }
42         visit[i] = true;
43         std::priority_queue<std::pair<int, std::pair<int, int> > > pq;
44         for (int j = 0; j < (int)adj[i].size(); j++) {
45             pq.push(std::make_pair(-adj[i][j].second,
46                                    std::make_pair(i, adj[i][j].first)));
47         }
48         while (!pq.empty()) {
49             int u = pq.top().second.first;
50             int v = pq.top().second.second;
51             int w = -pq.top().first;
52             pq.pop();
53             if (visit[u] && !visit[v]) {
54                 visit[v] = true;
55                 if (v != i) {
56                     mst.push_back(std::make_pair(u, v));
57                     total_dist += w;
58                 }
59                 for (int j = 0; j < (int)adj[v].size(); j++) {
60                     pq.push(std::make_pair(-adj[v][j].second,
61                                            std::make_pair(v, adj[v][j].first)));
62                 }
63             }
64         }
65     }
66     return total_dist;
67 }
68
69 /** Example Usage and Output:
70
71 Total distance: 13
72 0 <-> 2
73 0 <-> 1
74 3 <-> 4
75 4 <-> 5
76 5 <-> 6
77
78 */
79
80 #include <iostream>
81 using namespace std;
82
83 void add_edge(int u, int v, int w) {
84     adj[u].push_back(make_pair(v, w));
85     adj[v].push_back(make_pair(u, w));
86 }
87
88 int main() {
89     add_edge(0, 1, 4);
90     add_edge(1, 2, 6);

```

```

91     add_edge(2, 0, 3);
92     add_edge(3, 4, 1);
93     add_edge(4, 5, 2);
94     add_edge(5, 6, 3);
95     add_edge(6, 4, 4);
96     cout << "Total distance: " << prim(7) << endl;
97     for (int i = 0; i < (int)mst.size(); i++) {
98         cout << mst[i].first << " -> " << mst[i].second << endl;
99     }
100    return 0;
101 }
```

4.4.2 Minimum Spanning Tree (Kruskal)

```

1  /*
2
3 Given a connected, undirected, weighted graph with possibly negative weights,
4 its minimum spanning tree is a subgraph which is a tree that connects all nodes
5 with a subset of its edges such that their total weight is minimized. kruskal()
6 applies to a global, pre-populated adjacency list adj[] which must only consist
7 of nodes numbered with integers between 0 (inclusive) and the total number of
8 nodes (exclusive), as passed in the function argument. If the input graph is not
9 connected, then this implementation will find the minimum spanning forest.
10
11 Time Complexity:
12 - O(m log n) per call to kruskal(), where m is the number of edges and n is the
13   number of nodes.
14
15 Space Complexity:
16 - O(max(n, m)) for storage of the graph, where n the number of nodes and m is
17   the number of edges
18 - O(n) auxiliary stack space for kruskal().
19 */
20
21
22 #include <algorithm>
23 #include <utility>
24 #include <vector>
25
26 const int MAXN = 100;
27 std::vector<std::pair<int, std::pair<int, int> > > edges;
28 int root[MAXN];
29 std::vector<std::pair<int, int> > mst;
30
31 int find_root(int x) {
32     if (root[x] != x) {
33         root[x] = find_root(root[x]);
34     }
35     return root[x];
36 }
37
38 int kruskal(int nodes) {
39     mst.clear();
40     std::sort(edges.begin(), edges.end());
41     int total_dist = 0;
```

```

42     for (int i = 0; i < nodes; i++) {
43         root[i] = i;
44     }
45     for (int i = 0; i < (int)edges.size(); i++) {
46         int u = find_root(edges[i].second.first);
47         int v = find_root(edges[i].second.second);
48         if (u != v) {
49             root[u] = root[v];
50             mst.push_back(edges[i].second);
51             total_dist += edges[i].first;
52         }
53     }
54     return total_dist;
55 }
56
57 /** Example Usage and Output:
58
59 Total distance: 13
60 3 <-> 4
61 4 <-> 5
62 2 <-> 0
63 5 <-> 6
64 0 <-> 1
65
66 */
67
68 #include <iostream>
69 using namespace std;
70
71 void add_edge(int u, int v, int w) {
72     edges.push_back(make_pair(w, make_pair(u, v)));
73 }
74
75 int main() {
76     add_edge(0, 1, 4);
77     add_edge(1, 2, 6);
78     add_edge(2, 0, 3);
79     add_edge(3, 4, 1);
80     add_edge(4, 5, 2);
81     add_edge(5, 6, 3);
82     add_edge(6, 4, 4);
83     cout << "Total distance: " << kruskal(7) << endl;
84     for (int i = 0; i < (int)mst.size(); i++) {
85         cout << mst[i].first << " <-> " << mst[i].second << endl;
86     }
87     return 0;
88 }
```

4.5 Maximum Flow

4.5.1 Maximum Flow (Ford-Fulkerson)

```

2
3 Given a flow network with integer capacities, find the maximum flow from a given
4 source node to a given sink node. The flow of a given edge u -> v is defined as
5 the minimum of its capacity and the sum of the flows of all incoming edges of u.
6 ford_fulkerson() applies to global variables nodes, source, sink, and cap[][][]
7 which is an adjacency matrix that will be modified by the function call.
8
9 The Ford-Fulkerson algorithm is only optimal on graphs with integer capacities,
10 as there exists certain real-valued flow inputs for which the algorithm never
11 terminates. The Edmonds-Karp algorithm is an improvement using breadth-first
12 search, addressing this problem.
13
14 Time Complexity:
15 - O(n^2*f) per call to ford_fulkerson(), where n is the number of nodes and f
16   is the maximum flow.
17
18 Space Complexity:
19 - O(n^2) for storage of the flow network, where n is the number of nodes.
20 - O(n) auxiliary stack space for ford_fulkerson().
21
22 */
23
24 #include <algorithm>
25 #include <vector>
26
27 const int MAXN = 100, INF = 0x3f3f3f3f;
28 int nodes, source, sink, cap[MAXN][MAXN];
29 std::vector<bool> visit(MAXN);
30
31 int dfs(int u, int f) {
32     if (u == sink) {
33         return f;
34     }
35     visit[u] = true;
36     for (int v = 0; v < nodes; v++) {
37         if (!visit[v] && cap[u][v] > 0) {
38             int flow = dfs(v, std::min(f, cap[u][v]));
39             if (flow > 0) {
40                 cap[u][v] -= flow;
41                 cap[v][u] += flow;
42                 return flow;
43             }
44         }
45     }
46     return 0;
47 }
48
49 int ford_fulkerson() {
50     int max_flow = 0;
51     for (;;) {
52         std::fill(visit.begin(), visit.end(), false);
53         int flow = dfs(source, INF);
54         if (flow == 0) {
55             break;
56         }
57         max_flow += flow;
58     }
59     return max_flow;
60 }
```

```

61
62 /** Example Usage **/
63
64 #include <cassert>
65
66 int main() {
67     nodes = 6;
68     source = 0;
69     sink = 5;
70     cap[0][1] = 3;
71     cap[0][2] = 3;
72     cap[1][2] = 2;
73     cap[1][3] = 3;
74     cap[2][4] = 2;
75     cap[3][4] = 1;
76     cap[3][5] = 2;
77     cap[4][5] = 3;
78     assert(ford_fulkerson() == 5);
79     return 0;
80 }
```

4.5.2 Maximum Flow (Edmonds-Karp)

```

1 /*
2
3 Given a flow network with integer capacities, find the maximum flow from a given
4 source node to a given sink node. The flow of a given edge  $u \rightarrow v$  is defined as
5 the minimum of its capacity and the sum of the flows of all incoming edges of  $u$ .
6 edmonds_karp() applies to a global adjacency list adj[] that will be modified by
7 the function call.
8
9 The Edmonds-Karp algorithm will also support real-valued flow capacities. As
10 such, this implementation will work as intended upon changing the appropriate
11 variables to doubles.
12
13 Time Complexity:
14 -  $O(\min(n*m^2, m*f))$  per call to edmonds_karp(), where  $n$  is the number of nodes,
15    $m$  is the number of edges, and  $f$  is the maximum flow.
16
17 Space Complexity:
18 -  $O(\max(n, m))$  for storage of the flow network, where  $n$  is the number of nodes
19   and  $m$  is the number of edges.
20 */
21
22
23 #include <algorithm>
24 #include <queue>
25 #include <vector>
26
27 struct edge { int u, v, rev, cap, f; };
28
29 const int MAXN = 100, INF = 0x3f3f3f3f;
30 std::vector<edge> adj[MAXN];
31
32 void add_edge(int u, int v, int cap) {
```

```

33     adj[u].push_back((edge){u, v, (int)adj[v].size(), cap, 0});
34     adj[v].push_back((edge){v, u, (int)adj[u].size() - 1, 0, 0});
35 }
36
37 int edmonds_karp(int nodes, int source, int sink) {
38     int max_flow = 0;
39     for (;;) {
40         std::vector<edge*> pred(nodes, (edge*)0);
41         std::queue<int> q;
42         q.push(source);
43         while (!q.empty() && !pred[sink]) {
44             int u = q.front();
45             q.pop();
46             for (int j = 0; j < (int)adj[u].size(); j++) {
47                 edge &e = adj[u][j];
48                 if (!pred[e.v] && e.cap > e.f) {
49                     pred[e.v] = &e;
50                     q.push(e.v);
51                 }
52             }
53         }
54         if (!pred[sink]) {
55             break;
56         }
57         int flow = INF;
58         for (int u = sink; u != source; u = pred[u]->u) {
59             flow = std::min(flow, pred[u]->cap - pred[u]->f);
60         }
61         for (int u = sink; u != source; u = pred[u]->u) {
62             pred[u]->f += flow;
63             adj[pred[u]->v][pred[u]->rev].f -= flow;
64         }
65         max_flow += flow;
66     }
67     return max_flow;
68 }
69
70 /** Example Usage ***/
71
72 #include <cassert>
73
74 int main() {
75     add_edge(0, 1, 3);
76     add_edge(0, 2, 3);
77     add_edge(1, 2, 2);
78     add_edge(1, 3, 3);
79     add_edge(2, 4, 2);
80     add_edge(3, 4, 1);
81     add_edge(3, 5, 2);
82     add_edge(4, 5, 3);
83     assert(edmonds_karp(6, 0, 5) == 5);
84     return 0;
85 }
```

4.5.3 Maximum Flow (Dinic)

```

1  /*
2
3 Given a flow network with integer capacities, find the maximum flow from a given
4 source node to a given sink node. The flow of a given edge  $u \rightarrow v$  is defined as
5 the minimum of its capacity and the sum of the flows of all incoming edges of  $u$ .
6 dinic() applies to a global adjacency list adj[] that will be modified by the
7 function call.
8
9 Dinic's algorithm will also support real-valued flow capacities. As such, this
10 implementation will work as intended upon changing the appropriate variables to
11 doubles.
12
13 Time Complexity:
14 -  $O(n^2m)$  per call to dinic(), where  $n$  is the number of nodes and  $m$  is the
15   number of edges.
16
17 Space Complexity:
18 -  $O(\max(n, m))$  for storage of the flow network, where  $n$  is the number of nodes
19   and  $m$  is the number of edges.
20 -  $O(n)$  auxiliary stack and heap space for dinic().
21
22 */
23
24 #include <algorithm>
25 #include <queue>
26 #include <vector>
27
28 struct edge { int v, rev, cap, f; };
29
30 const int MAXN = 100, INF = 0x3f3f3f3f;
31 std::vector<edge> adj[MAXN];
32 int dist[MAXN], ptr[MAXN];
33
34 void add_edge(int u, int v, int cap) {
35     adj[u].push_back((edge){v, (int)adj[v].size(), cap, 0});
36     adj[v].push_back((edge){u, (int)adj[u].size() - 1, 0, 0});
37 }
38
39 bool dinic_bfs(int nodes, int source, int sink) {
40     std::fill(dist, dist + nodes, -1);
41     dist[source] = 0;
42     std::queue<int> q;
43     q.push(source);
44     while (!q.empty()) {
45         int u = q.front();
46         q.pop();
47         for (int j = 0; j < (int)adj[u].size(); j++) {
48             edge &e = adj[u][j];
49             if (dist[e.v] < 0 && e.f < e.cap) {
50                 dist[e.v] = dist[u] + 1;
51                 q.push(e.v);
52             }
53         }
54     }
55     return dist[sink] >= 0;
56 }
57
58 int dinic_dfs(int u, int f, int sink) {
59     if (u == sink) {

```

```

60     return f;
61 }
62 for ( ; ptr[u] < (int)adj[u].size(); ptr[u]++) {
63     edge &e = adj[u][ptr[u]];
64     if (dist[e.v] == dist[u] + 1 && e.f < e.cap) {
65         int flow = dinic_dfs(e.v, std::min(f, e.cap - e.f), sink);
66         if (flow > 0) {
67             e.f += flow;
68             adj[e.v][e.rev].f -= flow;
69             return flow;
70         }
71     }
72 }
73 return 0;
74 }

75
76 int dinic(int nodes, int source, int sink) {
77     int flow, max_flow = 0;
78     while (dinic_bfs(nodes, source, sink)) {
79         std::fill(ptr, ptr + nodes, 0);
80         while ((flow = dinic_dfs(source, INF, sink)) != 0) {
81             max_flow += flow;
82         }
83     }
84     return max_flow;
85 }

86 /**
87  * Example Usage */
88
89 #include <cassert>
90
91 int main() {
92     add_edge(0, 1, 3);
93     add_edge(0, 2, 3);
94     add_edge(1, 2, 2);
95     add_edge(1, 3, 3);
96     add_edge(2, 4, 2);
97     add_edge(3, 4, 1);
98     add_edge(3, 5, 2);
99     add_edge(4, 5, 3);
100    assert(dinic(6, 0, 5) == 5);
101    return 0;
102 }
```

4.5.4 Maximum Flow (Push-Relabel)

```

1 /*
2
3 Given a flow network with integer capacities, find the maximum flow from a given
4 source node to a given sink node. The flow of a given edge  $u \rightarrow v$  is defined as
5 the minimum of its capacity and the sum of the flows of all incoming edges of  $u$ .
6 push_relabel() applies to a global adjacency matrix cap[][] and returns the
7 maximum flow.
8
9 Although the push-relabel algorithm is considered one of the most efficient
```

```

10 maximum flow algorithms, it cannot take advantage of the magnitude of the
11 maximum flow being less than  $n^3$  (in which case the Ford-Fulkerson or
12 Edmonds-Karp algorithms may be more efficient).
13
14 Time Complexity:
15 -  $O(n^3)$  per call to push_relabel(), where  $n$  is the number of nodes.
16
17 Space Complexity:
18 -  $O(n^2)$  for storage of the flow network, where  $n$  is the number of nodes.
19 -  $O(n)$  auxiliary heap space for push_relabel().
20
21 */
22
23 #include <algorithm>
24 #include <vector>
25
26 const int MAXN = 100, INF = 0x3f3f3f3f;
27 int cap[MAXN][MAXN], f[MAXN][MAXN];
28
29 int push_relabel(int nodes, int source, int sink) {
30     std::vector<int> e(nodes, 0), h(nodes, 0), maxh(nodes, 0);
31     for (int i = 0; i < nodes; i++) {
32         std::fill(f[i], f[i] + nodes, 0);
33     }
34     h[source] = nodes - 1;
35     for (int i = 0; i < nodes; i++) {
36         f[source][i] = cap[source][i];
37         f[i][source] = -f[source][i];
38         e[i] = cap[source][i];
39     }
40     int size = 0;
41     for (;;) {
42         if (size == 0) {
43             for (int i = 0; i < nodes; i++) {
44                 if (i != source && i != sink && e[i] > 0) {
45                     if (size != 0 && h[i] > h[maxh[0]]) {
46                         size = 0;
47                     }
48                     maxh[size++] = i;
49                 }
50             }
51         }
52         if (size == 0) {
53             break;
54         }
55         while (size != 0) {
56             int i = maxh[size - 1];
57             bool pushed = false;
58             for (int j = 0; j < nodes && e[i] != 0; j++) {
59                 if (h[i] == h[j] + 1 && cap[i][j] - f[i][j] > 0) {
60                     int df = std::min(cap[i][j] - f[i][j], e[i]);
61                     f[i][j] += df;
62                     f[j][i] -= df;
63                     e[i] -= df;
64                     e[j] += df;
65                     if (e[i] == 0) {
66                         size--;
67                     }
68                     pushed = true;
69                 }
70             }
71         }
72     }
73 }
```

```

69         }
70     }
71     if (pushed) {
72         continue;
73     }
74     h[i] = INF;
75     for (int j = 0; j < nodes; j++) {
76         if (h[i] > h[j] + 1 && cap[i][j] - f[i][j] > 0) {
77             h[i] = h[j] + 1;
78         }
79     }
80     if (h[i] > h[maxh[0]]) {
81         size = 0;
82         break;
83     }
84 }
85 }
86 int max_flow = 0;
87 for (int i = 0; i < nodes; i++) {
88     max_flow += f[source][i];
89 }
90 return max_flow;
91 }
92
93 /** Example Usage ***/
94
95 #include <cassert>
96
97 int main() {
98     cap[0][1] = 3;
99     cap[0][2] = 3;
100    cap[1][2] = 2;
101    cap[1][3] = 3;
102    cap[2][4] = 2;
103    cap[3][4] = 1;
104    cap[3][5] = 2;
105    cap[4][5] = 3;
106    assert(push_relabel(6, 0, 5) == 5);
107    return 0;
108 }
```

4.6 Maximum Matching

4.6.1 Maximum Bipartite Matching (Kuhn)

```

1 /*
2
3 Given two sets of nodes A = {0, 1, ..., n1} and B = {0, 1, ..., n2} such that
4 n1 < n2, as well as a set of edges E mapping nodes from set A to set B, find the
5 largest possible subset of E containing no edges that share the same node.
6 kuhn() applies to a global, pre-populated adjacency list adj[] which must only
7 consist of nodes numbered with integers between 0 (inclusive) and the total
8 number of nodes (exclusive), as passed in the function argument.
```

```

9
10 Time Complexity:
11 - O(m*(n1 + n2)) per call to kuhn(), where m is the number of edges.
12
13 Space Complexity:
14 - O(n1 + n2) auxiliary stack space for kuhn().
15
16 */
17
18 #include <algorithm>
19 #include <vector>
20
21 const int MAXN = 100;
22 int match[MAXN];
23 std::vector<bool> visit(MAXN);
24 std::vector<int> adj[MAXN];
25
26 bool dfs(int u) {
27     visit[u] = true;
28     for (int j = 0; j < (int)adj[u].size(); j++) {
29         int v = match[adj[u][j]];
30         if (v == -1 || (!visit[v] && dfs(v))) {
31             match[adj[u][j]] = u;
32             return true;
33         }
34     }
35     return false;
36 }
37
38 int kuhn(int n1, int n2) {
39     std::fill(visit.begin(), visit.end(), false);
40     std::fill(match, match + n2, -1);
41     int matches = 0;
42     for (int i = 0; i < n1; i++) {
43         std::fill(visit.begin(), visit.begin() + n1, false);
44         if (dfs(i)) {
45             matches++;
46         }
47     }
48     return matches;
49 }
50
51 /** Example Usage and Output:
52
53 Matched 3 pair(s):
54 1 0
55 0 1
56 2 2
57
58 ***/
59
60 #include <iostream>
61 using namespace std;
62
63 int main() {
64     int n1 = 3, n2 = 4;
65     adj[0].push_back(1);
66     adj[1].push_back(0);
67     adj[1].push_back(1);

```

```

68     adj[1].push_back(2);
69     adj[2].push_back(2);
70     adj[2].push_back(3);
71     cout << "Matched " << kuhn(n1, n2) << " pair(s):" << endl;
72     for (int i = 0; i < n2; i++) {
73         if (match[i] != -1) {
74             cout << match[i] << " " << i << endl;
75         }
76     }
77     return 0;
78 }
```

4.6.2 Maximum Bipartite Matching (Hopcroft-Karp)

```

1  /*
2
3 Given two sets of nodes A = {0, 1, ..., n1} and B = {0, 1, ..., n2} such that
4 n1 < n2, as well as a set of edges E mapping nodes from set A to set B, find the
5 largest possible subset of E containing no edges that share the same node.
6 hopcroft_karp() applies to a global, pre-populated adjacency list adj[] which
7 must only consist of nodes numbered with integers between 0 (inclusive) and the
8 total number of nodes (exclusive), as passed in the function argument.
9
10 Time Complexity:
11 - O(m*sqrt(n1 + n2)) per call to hopcroft_karp(), where m is the number of
12   edges.
13
14 Space Complexity:
15 - O(max(n, m)) for storage of the graph, where n the number of nodes and m is
16   the number of edges.
17 - O(n1 + n2) auxiliary stack and heap space for hopcroft_karp().
18
19 */
20
21 #include <algorithm>
22 #include <queue>
23 #include <vector>
24
25 const int MAXN = 100;
26 std::vector<int> adj[MAXN];
27 std::vector<bool> used(MAXN), visit(MAXN);
28 int match[MAXN], dist[MAXN];
29
30 void bfs(int n1, int n2) {
31     std::fill(dist, dist + n1, -1);
32     std::queue<int> q;
33     for (int u = 0; u < n1; u++) {
34         if (!used[u]) {
35             q.push(u);
36             dist[u] = 0;
37         }
38     }
39     while (!q.empty()) {
40         int u = q.front();
41         q.pop();
```

```

42     for (int j = 0; j < (int)adj[u].size(); j++) {
43         int v = match[adj[u][j]];
44         if (v >= 0 && dist[v] < 0) {
45             dist[v] = dist[u] + 1;
46             q.push(v);
47         }
48     }
49 }
50 }
51
52 bool dfs(int u) {
53     visit[u] = true;
54     for (int j = 0; j < (int)adj[u].size(); j++) {
55         int v = match[adj[u][j]];
56         if (v < 0 || (!visit[v] && dist[v] == dist[u] + 1 && dfs(v))) {
57             match[adj[u][j]] = u;
58             used[u] = true;
59             return true;
60         }
61     }
62     return false;
63 }
64
65 int hopcroft_karp(int n1, int n2) {
66     std::fill(match, match + n2, -1);
67     std::fill(used.begin(), used.end(), false);
68     int res = 0;
69     for (;;) {
70         bfs(n1, n2);
71         std::fill(visit.begin(), visit.end(), false);
72         int f = 0;
73         for (int u = 0; u < n1; u++) {
74             if (!used[u] && dfs(u)) {
75                 f++;
76             }
77         }
78         if (f == 0) {
79             return res;
80         }
81         res += f;
82     }
83     return res;
84 }
85
86 /**
87 Matched 3 pair(s):
88 1 0
89 0 1
90 2 2
91 */
92
93 /**
94 #include <iostream>
95 using namespace std;
96
97 int main() {
98     int n1 = 3, n2 = 4;
99     adj[0].push_back(1);
100    adj[1].push_back(0);
101    adj[2].push_back(0);
102    adj[2].push_back(1);
103    adj[2].push_back(2);
104    adj[3].push_back(1);
105    adj[3].push_back(2);
106    adj[3].push_back(3);
107
108    bfs(0, 3);
109
110    for (int i = 0; i < n1; i++) {
111        for (int j = 0; j < n2; j++) {
112            cout << match[i][j] << " ";
113        }
114        cout << endl;
115    }
116
117    cout << "Number of pairs matched: " << res << endl;
118
119    return 0;
120 }
```

```

101    adj[1].push_back(0);
102    adj[1].push_back(1);
103    adj[1].push_back(2);
104    adj[2].push_back(2);
105    adj[2].push_back(3);
106    cout << "Matched " << hopcroft_karp(n1, n2) << " pair(s):" << endl;
107    for (int i = 0; i < n2; i++) {
108        if (match[i] != -1) {
109            cout << match[i] << " " << i << endl;
110        }
111    }
112    return 0;
113 }
```

4.6.3 Maximum Graph Matching (Edmonds)

```

1  /*
2
3 Given a directed graph, determine a maximal subset of its edges such that no
4 node is shared between different edges in the resulting subset. edmonds()
5 applies to a global, pre-populated adjacency list adj[] which must only consist
6 of nodes numbered with integers between 0 (inclusive) and the total number of
7 nodes (exclusive), as passed in the function argument.
8
9 Time Complexity:
10 - O(n^3) per call to edmonds(), where n is the number of nodes.
11
12 Space Complexity:
13 - O(max(n, m)) for storage of the graph, where n the number of nodes and m is
14   the number of edges.
15 - O(n) auxiliary heap space for edmonds(), where n is the number of nodes.
16
17 */
18
19 #include <queue>
20 #include <vector>
21
22 const int MAXN = 100;
23 std::vector<int> adj[MAXN];
24 int p[MAXN], base[MAXN], match[MAXN];
25
26 int lca(int nodes, int u, int v) {
27     std::vector<bool> used(nodes);
28     for (;;) {
29         u = base[u];
30         used[u] = true;
31         if (match[u] == -1) {
32             break;
33         }
34         u = p[match[u]];
35     }
36     for (;;) {
37         v = base[v];
38         if (used[v]) {
39             return v;
```

```

40     }
41     v = p[match[v]];
42   }
43 }
44
45 void mark_path(std::vector<bool> &blossom, int u, int b, int child) {
46   for (; base[u] != b; u = p[match[u]]) {
47     blossom[base[u]] = true;
48     blossom[base[match[u]]] = true;
49     p[u] = child;
50     child = match[u];
51   }
52 }
53
54 int find_path(int nodes, int root) {
55   std::vector<bool> used(nodes);
56   for (int i = 0; i < nodes; ++i) {
57     p[i] = -1;
58     base[i] = i;
59   }
60   used[root] = true;
61   std::queue<int> q;
62   q.push(root);
63   while (!q.empty()) {
64     int u = q.front();
65     q.pop();
66     for (int j = 0; j < (int)adj[u].size(); j++) {
67       int v = adj[u][j];
68       if (base[u] == base[v] || match[u] == v) {
69         continue;
70       }
71       if (v == root || (match[v] != -1 && p[match[v]] != -1)) {
72         int curr_base = lca(nodes, u, v);
73         std::vector<bool> blossom(nodes);
74         mark_path(blossom, u, curr_base, v);
75         mark_path(blossom, v, curr_base, u);
76         for (int i = 0; i < nodes; i++) {
77           if (blossom[base[i]]) {
78             base[i] = curr_base;
79             if (!used[i]) {
80               used[i] = true;
81               q.push(i);
82             }
83           }
84         }
85       } else if (p[v] == -1) {
86         p[v] = u;
87         if (match[v] == -1) {
88           return v;
89         }
90         v = match[v];
91         used[v] = true;
92         q.push(v);
93       }
94     }
95   }
96   return -1;
97 }
98

```

```

99 int edmonds(int nodes) {
100    for (int i = 0; i < nodes; i++) {
101        match[i] = -1;
102    }
103    for (int i = 0; i < nodes; i++) {
104        if (match[i] == -1) {
105            int u, pu, ppu;
106            for (u = find_path(nodes, i); u != -1; u = ppu) {
107                pu = p[u];
108                ppu = match[pu];
109                match[u] = pu;
110                match[pu] = u;
111            }
112        }
113    }
114    int matches = 0;
115    for (int i = 0; i < nodes; i++) {
116        if (match[i] != -1) {
117            matches++;
118        }
119    }
120    return matches/2;
121 }
122
123 /** Example Usage and Output:
124
125 Matched 2 pair(s):
126 0 1
127 2 3
128
129 */
130
131 #include <iostream>
132 using namespace std;
133
134 int main() {
135     int nodes = 4;
136     adj[0].push_back(1);
137     adj[1].push_back(0);
138     adj[1].push_back(2);
139     adj[2].push_back(1);
140     adj[2].push_back(3);
141     adj[3].push_back(2);
142     adj[3].push_back(0);
143     adj[0].push_back(3);
144     cout << "Matched " << edmonds(nodes) << " pair(s):" << endl;
145     for (int i = 0; i < nodes; i++) {
146         if (match[i] != -1 && i < match[i]) {
147             cout << i << " " << match[i] << endl;
148         }
149     }
150     return 0;
151 }
```

4.7 Hard Problems

4.7.1 Maximum Clique (Bron-Kerbosch)

```

1  /*
2
3 Given an undirected graph, max_clique() returns the size of the maximum clique,
4 that is, the largest subset of nodes such that all pairs of nodes in the subset
5 are connected by an edge. max_clique_weighted() additionally uses a global array
6 w[] specifying a weight value for each node, returning the clique in the graph
7 that has maximum total weight.
8
9 Both functions apply to a global, pre-populated adjacency matrix adj[] which
10 must satisfy the condition that adj[u][v] is true if and only if adj[v][u] is
11 true, for all pairs of nodes u and v respectively between 0 (inclusive) and the
12 total number of nodes (exclusive) as passed in the function argument. Note that
13 max_clique_weighted() is an efficient implementation using bitmasks of unsigned
14 64-bit integers, thus requiring the number of nodes to be less than 64.
15
16 Time Complexity:
17 - O(3^(n/3)) per call to max_clique() and max_clique_weighted(), where n
18 is the number of nodes.
19
20 Space Complexity:
21 - O(n^2) for storage of the graph, where n is the number of nodes.
22 - O(n) auxiliary stack space for max_clique() and max_clique_weighted().
23
24 */
25
26 #include <algorithm>
27 #include <bitset>
28 #include <vector>
29
30 const int MAXN = 35;
31 typedef std::bitset<MAXN> bits;
32 typedef unsigned long long uint64;
33
34 bool adj[MAXN][MAXN];
35 int w[MAXN];
36
37 int rec(int nodes, bits &curr, bits &pool, bits &excl) {
38     if (pool.none() && excl.none()) {
39         return curr.count();
40     }
41     int ans = 0, u = 0;
42     for (int v = 0; v < nodes; v++) {
43         if (pool[v] || excl[v]) {
44             u = v;
45         }
46     }
47     for (int v = 0; v < nodes; v++) {
48         if (!pool[v] || adj[u][v]) {
49             continue;
50         }
51         bits ncurr, npool, nexcl;
52         for (int i = 0; i < nodes; i++) {

```

```

53     ncurr[i] = curr[i];
54 }
55 ncurr[v] = true;
56 for (int j = 0; j < nodes; j++) {
57     npool[j] = pool[j] && adj[v][j];
58     nexcl[j] = excl[j] && adj[v][j];
59 }
60 ans = std::max(ans, rec(nodes, ncurr, npool, nexcl));
61 pool[v] = false;
62 excl[v] = true;
63 }
64 return ans;
65 }
66
67 int max_clique(int nodes) {
68     bits curr, excl, pool;
69     pool.flip();
70     return rec(nodes, curr, pool, excl);
71 }
72
73 int rec(const std::vector<uint64> &g, uint64 curr, uint64 pool, uint64 excl) {
74     if (pool == 0 && excl == 0) {
75         int res = 0, u = __builtin_ctzll(curr);
76         while (u < (int)g.size()) {
77             res += w[u];
78             u += __builtin_ctzll(curr >> (u + 1)) + 1;
79         }
80         return res;
81     }
82     if (pool == 0) {
83         return -1;
84     }
85     int res = -1, pivot = __builtin_ctzll(pool | excl);
86     uint64 z = pool & ~g[pivot];
87     int u = __builtin_ctzll(z);
88     while (u < (int)g.size()) {
89         res = std::max(res, rec(g, curr | (1LL << u), pool & g[u], excl & g[u]));
90         pool ^= 1LL << u;
91         excl |= 1LL << u;
92         u += __builtin_ctzll(z >> (u + 1)) + 1;
93     }
94     return res;
95 }
96
97 int max_clique_weighted(int nodes) {
98     std::vector<uint64> g(nodes, 0);
99     for (int i = 0; i < nodes; i++) {
100         for (int j = 0; j < nodes; j++) {
101             if (adj[i][j]) {
102                 g[i] |= 1LL << j;
103             }
104         }
105     }
106     return rec(g, 0, (1LL << nodes) - 1, 0);
107 }
108
109 /** Example Usage ***/
110
111 #include <cassert>

```

```

112
113 void add_edge(int u, int v) {
114     adj[u][v] = true;
115     adj[v][u] = true;
116 }
117
118 int main() {
119     add_edge(0, 1);
120     add_edge(0, 2);
121     add_edge(0, 3);
122     add_edge(1, 2);
123     add_edge(1, 3);
124     add_edge(2, 3);
125     add_edge(3, 4);
126     add_edge(4, 2);
127     w[0] = 10;
128     w[1] = 20;
129     w[2] = 30;
130     w[3] = 40;
131     w[4] = 50;
132     assert(max_clique(5) == 4);
133     assert(max_clique_weighted(5) == 120);
134     return 0;
135 }
```

4.7.2 Graph Coloring

```

1 /*
2
3 Given an undirected graph, assign a color to every node such that no pair of
4 adjacent nodes have the same color, and that the total number of colors used is
5 minimized. color_graph() applies to a global, pre-populated adjacency matrix
6 adj[][] which must satisfy the condition that adj[u][v] is true if and only if
7 adj[v][u] is true, for all pairs of nodes u and v respectively between 0
8 (inclusive) and the total number of nodes (exclusive) as passed in the function
9 argument.
10
11 Time Complexity:
12 - Exponential on the number of nodes per call to color_graph().
13
14 Space Complexity:
15 - O(n^2) for storage of the graph, where n is the number of nodes.
16 - O(n) auxiliary stack and heap space for color_graph().
17
18 */
19
20 #include <algorithm>
21 #include <vector>
22
23 const int MAXN = 30;
24 int adj[MAXN][MAXN], min_colors, color[MAXN];
25 int curr[MAXN], id[MAXN + 1], degree[MAXN + 1];
26
27 void rec(int lo, int hi, int n, int used_colors) {
28     if (used_colors >= min_colors) {
```

```

29     return;
30 }
31 if (n == hi) {
32     for (int i = lo; i < hi; i++) {
33         color[id[i]] = curr[i];
34     }
35     min_colors = used_colors;
36     return;
37 }
38 std::vector<bool> used(used_colors + 1);
39 for (int i = 0; i < n; i++) {
40     if (adj[id[n]][id[i]]) {
41         used[curr[i]] = true;
42     }
43 }
44 for (int i = 0; i <= used_colors; i++) {
45     if (!used[i]) {
46         int tmp = curr[n];
47         curr[n] = i;
48         rec(lo, hi, n + 1, std::max(used_colors, i + 1));
49         curr[n] = tmp;
50     }
51 }
52 }
53
54 int color_graph(int nodes) {
55     for (int i = 0; i <= nodes; i++) {
56         id[i] = i;
57         degree[i] = 0;
58     }
59     int res = 1, lo = 0;
60     for (int hi = 1; hi <= nodes; hi++) {
61         int best = hi;
62         for (int i = hi; i < nodes; i++) {
63             if (adj[id[hi - 1]][id[i]]) {
64                 degree[id[i]]++;
65             }
66             if (degree[id[best]] < degree[id[i]]) {
67                 best = i;
68             }
69         }
70         std::swap(id[hi], id[best]);
71         if (degree[id[hi]] == 0) {
72             min_colors = nodes + 1;
73             std::fill(curr, curr + nodes, 0);
74             rec(lo, hi, lo, 0);
75             lo = hi;
76             res = std::max(res, min_colors);
77         }
78     }
79     return res;
80 }
81
82 /**
83  * Example Usage and Output:
84  * Colored using 3 color(s):
85  * Color 1: 0 3
86  * Color 2: 1 2
87  * Color 3: 4

```

```

88
89 ***/
90
91 #include <cassert>
92 #include <iostream>
93 using namespace std;
94
95 void add_edge(int u, int v) {
96     adj[u][v] = true;
97     adj[v][u] = true;
98 }
99
100 int main() {
101     add_edge(0, 1);
102     add_edge(0, 4);
103     add_edge(1, 3);
104     add_edge(1, 4);
105     add_edge(2, 3);
106     add_edge(2, 4);
107     add_edge(3, 4);
108     int colors = color_graph(5);
109     cout << "Colored using " << colors << " color(s):" << endl;
110     for (int i = 0; i < colors; i++) {
111         cout << "Color " << i + 1 << ":";
112         for (int j = 0; j < 5; j++) {
113             if (color[j] == i) {
114                 cout << " " << j;
115             }
116         }
117         cout << endl;
118     }
119     return 0;
120 }
```

4.7.3 Shortest Hamiltonian Cycle (TSP)

```

1 /*
2
3 Given a weighted graph, determine a cycle of minimum total distance which visits
4 each node exactly once and returns to the starting node. This is known as the
5 traveling salesman problem (TSP). Since this implementation uses bitmasks with
6 32-bit integers, the maximum number of nodes must be less than 32.
7 shortest_hamiltonian_cycle() applies to a global adjacency matrix adj[][][], which
8 must be populated with add_edge() before the function call.
9
10 Time Complexity:
11 - O(2^n * n^2) per call to shortest_hamiltonian_cycle(), where n is the number
12   of nodes.
13
14 Space Complexity:
15 - O(n^2) for storage of the graph, where n is the number of nodes.
16 - O(2^n * n) auxiliary heap space for shortest_hamiltonian_cycle().
17
18 */
19
```

```

20 #include <algorithm>
21
22 const int MAXN = 20, INF = 0x3f3f3f3f;
23 int adj[MAXN][MAXN], dp[1 << MAXN][MAXN], order[MAXN];
24
25 void add_edge(int u, int v, int w) {
26     adj[u][v] = w;
27     adj[v][u] = w; // Remove this line if the graph is directed.
28 }
29
30 int shortest_hamiltonian_cycle(int nodes) {
31     int max_mask = (1 << nodes) - 1;
32     for (int i = 0; i <= max_mask; i++) {
33         std::fill(dp[i], dp[i] + nodes, INF);
34     }
35     dp[1][0] = 0;
36     for (int mask = 1; mask <= max_mask; mask += 2) {
37         for (int i = 1; i < nodes; i++) {
38             if ((mask & 1 << i) != 0) {
39                 for (int j = 0; j < nodes; j++) {
40                     if ((mask & 1 << j) != 0) {
41                         dp[mask][i] = std::min(dp[mask][i],
42                                     dp[mask ^ (1 << i)][j] + adj[j][i]);
43                     }
44                 }
45             }
46         }
47     }
48     int res = INF + INF;
49     for (int i = 1; i < nodes; i++) {
50         res = std::min(res, dp[max_mask][i] + adj[i][0]);
51     }
52     int mask = max_mask, old = 0;
53     for (int i = nodes - 1; i >= 1; i--) {
54         int bj = -1;
55         for (int j = 1; j < nodes; j++) {
56             if ((mask & 1 << j) != 0 && (bj == -1 ||
57                 dp[mask][bj] + adj[bj][old] > dp[mask][j] + adj[j][old])) {
58                 bj = j;
59             }
60         }
61         order[i] = bj;
62         mask ^= 1 << bj;
63         old = bj;
64     }
65     return res;
66 }
67
68 /** Example Usage and Output:
69
70 The shortest hamiltonian cycle has length 5.
71 Take the path: 0->3->2->4->1->0.
72
73 */
74
75 #include <iostream>
76 using namespace std;
77
78 int main() {

```

```

79     int nodes = 5;
80     add_edge(0, 1, 1);
81     add_edge(0, 2, 10);
82     add_edge(0, 3, 1);
83     add_edge(0, 4, 10);
84     add_edge(1, 2, 10);
85     add_edge(1, 3, 10);
86     add_edge(1, 4, 1);
87     add_edge(2, 3, 1);
88     add_edge(2, 4, 1);
89     add_edge(3, 4, 10);
90     cout << "The shortest hamiltonian cycle has length "
91         << shortest_hamiltonian_cycle(nodes) << "."
92         << endl;
93     << "Take the path: ";
94     for (int i = 0; i < nodes; i++) {
95         cout << order[i] << "->";
96     }
97     cout << order[0] << "."
98     return 0;
99 }
```

4.7.4 Shortest Hamiltonian Path

```

1  /*
2
3 Given a weighted, directed graph, determine a path of minimum total distance
4 which visits each node exactly once. Unlike the traveling salesman problem, we
5 do not have to return to the starting vertex. Since this implementation uses
6 bitmasks with 32-bit integers, the maximum number of nodes must be less than 32.
7 shortest_hamiltonian_path() applies to a global adjacency matrix adj[][] which
8 must be populated before the function call.
9
10 Time Complexity:
11 - O(2^n * n^2) per call to shortest_hamiltonian_path(), where n is the number
12   of nodes.
13
14 Space Complexity:
15 - O(n^2) for storage of the graph, where n is the number of nodes.
16 - O(2^n * n^2) auxiliary heap space for shortest_hamiltonian_path().
17 */
18
19 #include <algorithm>
20
21
22 const int MAXN = 20, INF = 0x3f3f3f3f;
23 int adj[MAXN][MAXN], dp[1 << MAXN][MAXN], order[MAXN];
24
25 int shortest_hamiltonian_path(int nodes) {
26     int max_mask = (1 << nodes) - 1;
27     for (int i = 0; i <= max_mask; i++) {
28         std::fill(dp[i], dp[i] + nodes, INF);
29     }
30     for (int i = 0; i < nodes; i++) {
31         dp[1 << i][i] = 0;
32     }
33 }
```

```

33     for (int mask = 1; mask <= max_mask; mask += 2) {
34         for (int i = 0; i < nodes; i++) {
35             if ((mask & 1 << i) != 0) {
36                 for (int j = 0; j < nodes; j++) {
37                     if ((mask & 1 << j) != 0)
38                         dp[mask][i] = std::min(dp[mask][i],
39                                     dp[mask ^ (1 << i)][j] + adj[j][i]);
40                 }
41             }
42         }
43     }
44     int res = INF + INF;
45     for (int i = 1; i < nodes; i++) {
46         res = std::min(res, dp[max_mask][i]);
47     }
48     int mask = max_mask, old = -1;
49     for (int i = nodes - 1; i >= 0; i--) {
50         int bj = -1;
51         for (int j = 0; j < nodes; j++) {
52             if ((mask & 1 << j) != 0 &&
53                 (bj == -1 || dp[mask][bj] + (old == -1 ? 0 : adj[bj][old]) >
54                  dp[mask][j] + (old == -1 ? 0 : adj[j][old]))) {
55                 bj = j;
56             }
57         }
58         order[i] = bj;
59         mask ^= (1 << bj);
60         old = bj;
61     }
62     return res;
63 }
64
65 /** Example Usage and Output:
66
67 The shortest hamiltonian path has length 3.
68 Take the path: 0->1->2.
69
70 */
71
72 #include <iostream>
73 using namespace std;
74
75 int main() {
76     int nodes = 3;
77     adj[0][1] = 1;
78     adj[0][2] = 1;
79     adj[1][0] = 7;
80     adj[1][2] = 2;
81     adj[2][0] = 3;
82     adj[2][1] = 5;
83     cout << "The shortest hamiltonian path has length "
84         << shortest_hamiltonian_path(nodes) << "."
85         << endl
86         << "Take the path: " << order[0];
87     for (int i = 1; i < nodes; i++) {
88         cout << "->" << order[i];
89     }
90     cout << "."
91     return 0;
92 }
```

Chapter 5

Mathematics

5.1 Math Utilities

```
1  /*
2
3 Common mathematic constants and functions, many of which are substitutes for
4 features which are not available in standard C++, or may not be available on
5 compilers that do not support C++11 and later.
6
7 Time Complexity:
8 - O(1) for all operations.
9
10 Space Complexity:
11 - O(1) auxiliary for all operations.
12 */
13
14 #include <algorithm>
15 #include <cfloat>
16 #include <climits>
17 #include <cmath>
18 #include <cstdlib>
19 #include <limits>
20 #include <string>
21 #include <vector>
22
23
24 #ifndef M_PI
25     const double M_PI = acos(-1.0);
26 #endif
27 #ifndef M_E
28     const double M_E = exp(1.0);
29 #endif
30 const double M_PHI = (1.0 + sqrt(5.0))/2.0;
31 const double M_INF = std::numeric_limits<double>::infinity();
32 const double M_NAN = std::numeric_limits<double>::quiet_NaN();
33
34 #ifndef isnan
35     #define isnan(x) ((x) != (x))
36 #endif
```

```

37 /*
38  * 
39  * Epsilon Comparisons
40  *
41  * EQ(), NE(), LT(), GT(), LE(), and GE() relationally compares two values x and y
42  * accounting for absolute error. For any x, the range of values considered equal
43  * barring absolute error is [x - EPS, x + EPS]. Values outside of this range are
44  * considered not equal (strictly less or strictly greater).
45  *
46  * rEQ() returns whether x and y are equal barring relative error. For any x, the
47  * range of values considered equal is [x*(1 - EPS), x*(1 + EPS)].
48  *
49  */
50 
51 const double EPS = 1e-9;
52 
53 #define EQ(x, y) (fabs((x) - (y)) <= EPS)
54 #define NE(x, y) (fabs((x) - (y)) > EPS)
55 #define LT(x, y) ((x) < (y) - EPS)
56 #define GT(x, y) ((x) > (y) + EPS)
57 #define LE(x, y) ((x) <= (y) + EPS)
58 #define GE(x, y) ((x) >= (y) - EPS)
59 #define rEQ(x, y) (fabs((x) - (y)) <= EPS*fabs(x))
60 
61 /*
62  * 
63  * Sign Functions
64  *
65  * - sgn(x) returns -1 (if x < 0), 0 (if x == 0), or 1 (if x > 0). Unlike signbit()
66  * or copysign(), this does not handle the sign of NaN.
67  * - signbit_(x) is analogous to std::signbit() in C++11 and later, returning
68  * whether the sign bit of the floating point number is set to true. If so, then
69  * x is considered "negative." Note that this works as expected on +0.0, -0.0,
70  * Inf, -Inf, NaN, as well as -NaN. Warning: This assumes that the sign bit is
71  * the leading (most significant) bit in the internal representation of the IEEE
72  * floating point value.
73  * - copysign_(x, y) is analogous to std::copysign() in C++11 and later, returning
74  * a number with the magnitude of x but the sign of y.
75  *
76  */
77 
78 template<class T>
79 int sgn(const T &x) {
80     return (T(0) < x) - (x < T(0));
81 }
82 
83 template<class Double>
84 bool signbit_(Double x) {
85     return (((unsigned char *)&x)[sizeof(x) - 1] >> (CHAR_BIT - 1)) & 1;
86 }
87 
88 template<class Double>
89 Double copysign_(Double x, Double y) {
90     return signbit_(y) ? -fabs(x) : fabs(x);
91 }
92 
93 /*
94  */
95 
```

```
96 Rounding Functions
97
98 - floor0(x) returns x rounded down, symmetrically towards zero. This function is
99   analogous to trunc() in C++11 and later.
100 - ceil0(x) returns x rounded up, symmetrically away from zero. This function is
101   analogous to round() in C++11 and later.
102 - round_half_up(x) returns x rounded half up, towards positive infinity.
103 - round_half_down(x) returns x rounded half down, towards negative infinity.
104 - round_half_to0(x) returns x rounded half down, symmetrically towards zero.
105 - round_half_from0(x) returns x rounded half up, symmetrically away from zero.
106 - round_half_even(x) returns x rounded half to even, using banker's rounding.
107 - round_half_alternate(x) returns x rounded, where ties are broken by
108   alternating rounds towards positive and negative infinity.
109 - round_half_alternate0(x) returns x rounded, where ties are broken by
110   alternating symmetric rounds towards and away from zero.
111 - round_half_random(x) returns x rounded, where ties are broken randomly.
112 - round_n_places(x, n, f) returns x rounded to n digits after the decimal, using
113   the specified rounding function f(x).
114 */
116
117 template<class Double>
118 Double floor0(const Double &x) {
119     Double res = floor(fabs(x));
120     return (x < 0.0) ? -res : res;
121 }
122
123 template<class Double>
124 Double ceil0(const Double &x) {
125     Double res = ceil(fabs(x));
126     return (x < 0.0) ? -res : res;
127 }
128
129 template<class Double>
130 Double round_half_up(const Double &x) {
131     return floor(x + 0.5);
132 }
133
134 template<class Double>
135 Double round_half_down(const Double &x) {
136     return ceil(x - 0.5);
137 }
138
139 template<class Double>
140 Double round_half_to0(const Double &x) {
141     Double res = round_half_down(fabs(x));
142     return (x < 0.0) ? -res : res;
143 }
144
145 template<class Double>
146 Double round_half_from0(const Double &x) {
147     Double res = round_half_up(fabs(x));
148     return (x < 0.0) ? -res : res;
149 }
150
151 template<class Double>
152 Double round_half_even(const Double &x, const Double &eps = 1e-9) {
153     if (x < 0.0) {
154         return -round_half_even(-x, eps);
```

```

155     }
156     Double ipart;
157     modf(x, &ipart);
158     if (x - (ipart + 0.5) < eps) {
159         return (fmod(ipart, 2.0) < eps) ? ipart : ceil0(ipart + 0.5);
160     }
161     return round_half_from0(x);
162 }
163
164 template<class Double>
165 Double round_half_alternate(const Double &x) {
166     static bool up = true;
167     return (up = !up) ? round_half_up(x) : round_half_down(x);
168 }
169
170 template<class Double>
171 Double round_half_alternate0(const Double &x) {
172     static bool up = true;
173     return (up = !up) ? round_half_from0(x) : round_half_to0(x);
174 }
175
176 template<class Double>
177 Double round_half_random(const Double &x) {
178     return (rand() % 2 == 0) ? round_half_from0(x) : round_half_to0(x);
179 }
180
181 template<class Double, class RoundingFunction>
182 Double round_n_places(const Double &x, unsigned int n, RoundingFunction f) {
183     return f(x*pow(10, n)) / pow(10, n);
184 }
185
186 /*
187
188 Error Function
189
190 - erf_(x) returns the error encountered in integrating the normal distribution.
191 Its value is  $2/\sqrt{\pi} \cdot (\text{integral of } e^{-t^2} dt \text{ from 0 to } x)$ . This function
192 is analogous to erf(x) in C++11 and later.
193 - erfc_(x) returns the error function complement, that is,  $1 - \text{erf}_-(x)$ . This
194 function is analogous to erfc(x) in C++11 and later.
195
196 */
197
198 #define ERF_EPS 1e-14
199
200 double erfc_(double x);
201
202 double erf_(double x) {
203     if (signbit_(x)) {
204         return -erf_(-x);
205     }
206     if (fabs(x) > 2.2) {
207         return 1.0 - erfc_(x);
208     }
209     double sum = x, term = x, xx = x*x;
210     int j = 1;
211     do {
212         term *= xx / j;
213         sum -= term/(2*(j++) + 1);

```

```
214     term *= xx / j;
215     sum += term/(2*(j++ ) + 1);
216 } while (fabs(term) > sum*ERF_EPS);
217 return 2/sqrt(M_PI) * sum;
218 }
219
220 double erfc_(double x) {
221 if (fabs(x) < 2.2) {
222     return 1.0 - erf_(x);
223 }
224 if (signbit_(x)) {
225     return 2.0 - erfc_(-x);
226 }
227 double a = 1, b = x, c = x, d = x*x + 0.5, q1, q2 = 0, n = 1.0, t;
228 do {
229     t = a*n + b*x;
230     a = b;
231     b = t;
232     t = c*n + d*x;
233     c = d;
234     d = t;
235     n += 0.5;
236     q1 = q2;
237     q2 = b / d;
238 } while (fabs(q1 - q2) > q2*ERF_EPS);
239 return 1/sqrt(M_PI) * exp(-x*x) * q2;
240 }
241 #undef ERF_EPS
242 /*
243
244 Gamma Functions
245
246 - tgamma_(x) returns the gamma function of x. Unlike the tgamma() function in
247 C++11 and later, this version only supports positive x, returning NaN if x is
248 less than or equal to 0.
249 - lgamma_(x) returns the natural logarithm of the absolute value of the gamma
250 function of x. Unlike the lgamma() function in C++11 and later, this version
251 only supports positive x, returning NaN if x is less than or equal to 0.
252
253 */
254
255 double lgamma_(double x);
256
257 double tgamma_(double x) {
258 if (x <= 0) {
259     return M_NAN;
260 }
261 if (x < 1e-3) {
262     return 1.0 / (x*(1.0 + 0.57721566490153286060651209*x));
263 }
264 if (x < 12) {
265     double y = x;
266     int n = 0;
267     bool arg_was_less_than_one = (y < 1);
268     if (arg_was_less_than_one) {
269         y += 1;
270     } else {
```

```

273     n = (int)floor(y) - 1;
274     y -= n;
275 }
276 static const double p[] = {
277     -1.71618513886549492533811e+0, 2.47656508055759199108314e+1,
278     -3.79804256470945635097577e+2, 6.29331155312818442661052e+2,
279     8.66966202790413211295064e+2, -3.14512729688483675254357e+4,
280     -3.61444134186911729807069e+4, 6.64561438202405440627855e+4};
281 static const double q[] = {
282     -3.08402300119738975254353e+1, 3.15350626979604161529144e+2,
283     -1.01515636749021914166146e+3, -3.10777167157231109440444e+3,
284     2.25381184209801510330112e+4, 4.75584627752788110767815e+3,
285     -1.34659959864969306392456e+5, -1.15132259675553483497211e+5};
286 double num = 0, den = 1, z = y - 1;
287 for (int i = 0; i < 8; i++) {
288     num = (num + p[i])*z;
289     den = den*z + q[i];
290 }
291 double result = num/den + 1;
292 if (arg_was_less_than_one) {
293     result /= (y - 1);
294 } else {
295     for (int i = 0; i < n; i++) {
296         result *= y++;
297     }
298 }
299 return result;
300 }
301 return (x > 171.624) ? 2*DBL_MAX : exp(lgamma(x));
302 }
303
304 double lgamma_(double x) {
305     if (x <= 0) {
306         return M_NAN;
307     }
308     if (x < 12) {
309         return log(fabs(tgamma_(x)));
310     }
311     static const double c[8] = {
312         1.0/12, -1.0/360, 1.0/1260, -1.0/1680, 1.0/1188, -691.0/360360, 1.0/156,
313         -3617.0/122400
314     };
315     double z = 1.0/(x*x), sum = c[7];
316     for (int i = 6; i >= 0; i--) {
317         sum = sum*z + c[i];
318     }
319     return (x - 0.5)*log(x) - x + 0.91893853320467274178032973640562 + sum/x;
320 }
321 /*
322 Base Conversion
323
324 - Given an integer in base a as a vector d of digits (where d[0] is the least
325   significant digit), convert_base(d, a, b) returns a vector of the integer's
326   digits when converted base b (again with index 0 storing the least significant
327   digit). The actual value of the entire integer to be converted must be able to
328   fit within an unsigned 64-bit integer for intermediate storage.
329 - to_base(x, b) returns the digits of the unsigned integer x in base b, where

```

```

332     index 0 of the result stores the least significant digit.
333 - to_roman(x) returns the Roman numeral representation of the unsigned integer x
334     as a C++ string.
335
336 */
337
338 std::vector<int> convert_base(const std::vector<int> &d, int a, int b) {
339     unsigned long long x = 0, power = 1;
340     for (int i = 0; i < (int)d.size(); i++) {
341         x += d[i]*power;
342         power *= a;
343     }
344     int n = ceil(log(x + 1)/log(b));
345     std::vector<int> res;
346     for (int i = 0; i < n; i++) {
347         res.push_back(x % b);
348         x /= b;
349     }
350     return res;
351 }
352
353 std::vector<int> to_base(unsigned int x, int b = 10) {
354     std::vector<int> res;
355     while (x != 0) {
356         res.push_back(x % b);
357         x /= b;
358     }
359     return res;
360 }
361
362 std::string to_roman(unsigned int x) {
363     static const std::string h[] =
364         {"", "C", "CC", "CCC", "CD", "DC", "DCC", "DCC", "CM"};
365     static const std::string t[] =
366         {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
367     static const std::string o[] =
368         {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
369     std::string prefix(x / 1000, 'M');
370     x %= 1000;
371     return prefix + h[x/100] + t[x/10 % 10] + o[x % 10];
372 }
373
374 /** Example Usage ***/
375
376 #include <cassert>
377 #include <iostream>
378
379 int main() {
380     assert(EQ(M_PI, 3.14159265359));
381     assert(EQ(M_E, 2.718281828459));
382     assert(EQ(M_PHI, 1.61803398875));
383
384     double x = -12345.6789;
385     assert((-M_INF < x) && (x < M_INF));
386     assert((M_INF + x == M_INF) && (M_INF - x == M_INF));
387     assert((M_INF + M_INF == M_INF) && (-M_INF - M_INF == -M_INF));
388     assert((M_NAN != x) && (M_NAN != M_INF) && (M_NAN != M_NAN));
389     assert(!(M_NAN < x) && !(M_NAN > x) && !(M_NAN <= x) && !(M_NAN >= x));
390     assert(isnan(0.0*M_INF) && isnan(0.0*-M_INF) && isnan(M_INF/-M_INF));

```

```

391 assert(isnan(M_NAN) && isnan(-M_NAN) && isnan(M_INF - M_INF));
392
393 assert(sgn(x) == -1 && sgn(0.0) == 0 && sgn(5678) == 1);
394 assert(signbit_(x) && !signbit_(0.0) && signbit_(-0.0));
395 assert(!signbit_(M_INF) && signbit_(-M_INF));
396 assert(!signbit_(M_NAN) && signbit_(-M_NAN));
397 assert(copysign(1.0, +2.0) == +1.0 && copysign(M_INF, -2.0) == -M_INF);
398 assert(copysign(1.0, -2.0) == -1.0 && std::signbit(copysign(M_NAN, -2.0)));
399
400 assert(EQ(floor0(1.5), 1.0) && EQ(ceil0(1.5), 2.0));
401 assert(EQ(floor0(-1.5), -1.0) && EQ(ceil0(-1.5), -2.0));
402 assert(EQ(round_half_up(+1.5), +2) && EQ(round_half_down(+1.5), +1));
403 assert(EQ(round_half_up(-1.5), -1) && EQ(round_half_down(-1.5), -2));
404 assert(EQ(round_half_to0(+1.5), +1) && EQ(round_half_from0(+1.5), +2));
405 assert(EQ(round_half_to0(-1.5), -1) && EQ(round_half_from0(-1.5), -2));
406 assert(EQ(round_half_even(+1.5), +2) && EQ(round_half_even(-1.5), -2));
407 assert(NE(round_half_alternate(+1.5), round_half_alternate(+1.5)));
408 assert(NE(round_half_alternate0(-1.5), round_half_alternate0(-1.5)));
409 assert(EQ(round_n_places(-1.23456, 3, round_half_to0<double>), -1.235));
410
411 assert(EQ(erf_(1.0), 0.8427007929) && EQ(erf_(-1.0), -0.8427007929));
412 assert(EQ(tgamma_(0.5), 1.7724538509) && EQ(tgamma_(1.0), 1.0));
413 assert(EQ(lgamma_(0.5), 0.5723649429) && EQ(lgamma_(1.0), 0.0));
414
415 int digits[] = {6, 5, 4, 3, 2, 1};
416 std::vector<int> base20 = to_base(123456, 20);
417 assert(convert_base(base20, 20, 10) == std::vector<int>(digits, digits + 6));
418 assert(to_roman(1234) == "MCCXXXIV");
419 assert(to_roman(5678) == "MMMMMDCLXXVIII");
420 return 0;
421 }

```

5.2 Combinatorics

5.2.1 Combinatorial Calculations

```

1 /*
2
3 The following functions implement common operations in combinatorics. All input
4 arguments must be non-negative. All return values and table entries are computed
5 as 64-bit integers modulo an input argument m or p.
6
7 - factorial(n, m) returns n! mod m.
8 - factorialp(n, p) returns n! mod p, where p is prime.
9 - binomial_table(n, m) returns rows 0 to n of Pascal's triangle as a table t
10 such that t[i][j] is equal to (i choose j) mod m.
11 - permute(n, k, m) returns (n permute k) mod m.
12 - choose(n, k, p) returns (n choose k) mod p, where p is prime.
13 - multichoose(n, k, p) returns (n multi-choose k) mod p, where p is prime.
14 - catalan(n, p) returns the nth Catalan number mod p, where p is prime.
15 - partitions(n, m) returns the number of partitions of n, mod m.
16 - partitions(n, k, m) returns the number of partitions of n into k parts, mod m.
17 - stirling1(n, k, m) returns the (n, k) unsigned Stirling number of the 1st kind

```

```
18     mod m.
19 - stirling2(n, k, m) returns the (n, k) Stirling number of the 2nd kind mod m.
20 - eulerian1(n, k, m) returns the (n, k) Eulerian number of the 1st kind mod m,
21   where n > k.
22 - eulerian2(n, k, m) returns the (n, k) Eulerian number of the 2nd kind mod m,
23   where n > k.
24
25 Time Complexity:
26 - O(n) for factorial(n, m).
27 - O(p log n) for factorialp(n, p).
28 - O(n^2) for binomial_table(n, m).
29 - O(k) for permute(n, k, p).
30 - O(min(k, n - k)) for choose(n, k, p).
31 - O(k) for multichoose(n, k, p).
32 - O(n) for catalan(n, p).
33 - O(n^2) for partitions(n, m).
34 - O(n*k) for partitions(n, k, m), stirling1(n, k, m), stirling2(n, k, m),
35   eulerian1(n, k, m), and eulerian2(n, k, m).
36
37 Space Complexity:
38 - O(n^2) auxiliary heap space for binomial_table(n, m).
39 - O(n*k) auxiliary heap space for partitions(n, k, m), stirling1(n, k, m),
40   stirling2(n, k, m), eulerian1(n, k, m), and eulerian2(n, k, m).
41 - O(1) auxiliary for all other operations.
42 */
43
44
45 #include <vector>
46
47 typedef long long int64;
48 typedef std::vector<std::vector<int64>> table;
49
50 int64 factorial(int n, int m = 1000000007) {
51     int64 res = 1;
52     for (int i = 2; i <= n; i++) {
53         res = (res*i) % m;
54     }
55     return res % m;
56 }
57
58 int64 factorialp(int64 n, int64 p = 1000000007) {
59     int64 res = 1;
60     while (n > 1) {
61         if (n / p % 2 == 1) {
62             res = res*(p - 1) % p;
63         }
64         int max = n % p;
65         for (int i = 2; i <= max; i++) {
66             res = (res*i) % p;
67         }
68         n /= p;
69     }
70     return res % p;
71 }
72
73 table binomial_table(int n, int64 m = 1000000007) {
74     table t(n + 1);
75     for (int i = 0; i <= n; i++) {
76         for (int j = 0; j <= i; j++) {
```

```

77     if (i < 2 || j == 0 || i == j) {
78         t[i].push_back(1);
79     } else {
80         t[i].push_back((t[i - 1][j - 1] + t[i - 1][j]) % m);
81     }
82 }
83 }
84 return t;
85 }

86 int64 permute(int n, int k, int64 m = 1000000007) {
87     if (n < k) {
88         return 0;
89     }
90     int64 res = 1;
91     for (int i = 0; i < k; i++) {
92         res = res*(n - i) % m;
93     }
94     return res % m;
95 }
96 }

97 int64 mulmod(int64 x, int64 n, int64 m) {
98     int64 a = 0, b = x % m;
99     for (; n > 0; n >>= 1) {
100         if (n & 1) {
101             a = (a + b) % m;
102         }
103         b = (b << 1) % m;
104     }
105     return a % m;
106 }
107 }

108 int64 powmod(int64 x, int64 n, int64 m) {
109     int64 a = 1, b = x;
110     for (; n > 0; n >>= 1) {
111         if (n & 1) {
112             a = mulmod(a, b, m);
113         }
114         b = mulmod(b, b, m);
115     }
116     return a % m;
117 }
118 }

119 int64 choose(int n, int k, int64 p = 1000000007) {
120     if (n < k) {
121         return 0;
122     }
123     if (k > n - k) {
124         k = n - k;
125     }
126     int64 num = 1, den = 1;
127     for (int i = 0; i < k; i++) {
128         num = num*(n - i) % p;
129     }
130     for (int i = 1; i <= k; i++) {
131         den = den*i % p;
132     }
133     return num*powmod(den, p - 2, p) % p;
134 }
135 }
```

```

136
137 int64 multichoose(int n, int k, int64 p = 1000000007) {
138     return choose(n + k - 1, k, p);
139 }
140
141 int64 catalan(int n, int64 p = 1000000007) {
142     return choose(2*n, n, p)*powmod(n + 1, p - 2, p) % p;
143 }
144
145 int64 partitions(int n, int64 m = 1000000007) {
146     std::vector<int64> t(n + 1, 0);
147     t[0] = 1;
148     for (int i = 1; i <= n; i++) {
149         for (int j = i; j <= n; j++) {
150             t[j] = (t[j] + t[j - i]) % m;
151         }
152     }
153     return t[n] % m;
154 }
155
156 int64 partitions(int n, int k, int64 m = 1000000007) {
157     table t(n + 1, std::vector<int64>(k + 1, 0));
158     t[0][1] = 1;
159     for (int i = 1; i <= n; i++) {
160         for (int j = 1, h = k < i ? k : i; j <= h; j++) {
161             t[i][j] = (t[i - 1][j - 1] + t[i - j][j]) % m;
162         }
163     }
164     return t[n][k] % m;
165 }
166
167 int64 stirling1(int n, int k, int64 m = 1000000007) {
168     table t(n + 1, std::vector<int64>(k + 1, 0));
169     t[0][0] = 1;
170     for (int i = 1; i <= n; i++) {
171         for (int j = 1; j <= k; j++) {
172             t[i][j] = (i - 1)*t[i - 1][j] % m;
173             t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
174         }
175     }
176     return t[n][k] % m;
177 }
178
179 int64 stirling2(int n, int k, int64 m = 1000000007) {
180     table t(n + 1, std::vector<int64>(k + 1, 0));
181     t[0][0] = 1;
182     for (int i = 1; i <= n; i++) {
183         for (int j = 1; j <= k; j++) {
184             t[i][j] = j*t[i - 1][j] % m;
185             t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
186         }
187     }
188     return t[n][k] % m;
189 }
190
191 int64 eulerian1(int n, int k, int64 m = 1000000007) {
192     if (k > n - 1 - k) {
193         k = n - 1 - k;
194     }

```

```

195     table t(n + 1, std::vector<int64>(k + 1, 1));
196     for (int j = 1; j <= k; j++) {
197         t[0][j] = 0;
198     }
199     for (int i = 1; i <= n; i++) {
200         for (int j = 1; j <= k; j++) {
201             t[i][j] = (i - j)*t[i - 1][j - 1] % m;
202             t[i][j] = (t[i][j] + ((j + 1)*t[i - 1][j] % m)) % m;
203         }
204     }
205     return t[n][k] % m;
206 }
207
208 int64 eulerian2(int n, int k, int64 m = 1000000007) {
209     table t(n + 1, std::vector<int64>(k + 1, 1));
210     for (int i = 1; i <= n; i++) {
211         for (int j = 1; j <= k; j++) {
212             if (i == j) {
213                 t[i][j] = 0;
214             } else {
215                 t[i][j] = (j + 1)*t[i - 1][j] % m;
216                 t[i][j] = ((2*i - 1 - j)*t[i - 1][j - 1] % m + t[i][j]) % m;
217             }
218         }
219     }
220     return t[n][k] % m;
221 }
222
223 /** Example Usage **/
224
225 #include <cassert>
226
227 int main() {
228     table t = binomial_table(10);
229     for (int i = 0; i < (int)t.size(); i++) {
230         for (int j = 0; j < (int)t[i].size(); j++) {
231             assert(t[i][j] == choose(i, j));
232         }
233     }
234     assert(factorial(10) == 3628800);
235     assert(factorialp(123456) == 639390503);
236     assert(permute(10, 4) == 5040);
237     assert(choose(20, 7) == 77520);
238     assert(multichoose(20, 7) == 657800);
239     assert(catalan(10) == 16796);
240     assert(partitions(4) == 5);
241     assert(partitions(100, 5) == 38225);
242     assert(stirling1(4, 2) == 11);
243     assert(stirling2(4, 3) == 6);
244     assert(eulerian1(9, 5) == 88234);
245     assert(eulerian2(8, 3) == 195800);
246     return 0;
247 }
```

5.2.2 Enumerating Arrangements

```

1  /*
2
3 For the purposes of this section, we define a "size k arrangement of n" to be a
4 permutation of a size k subset of the integers from 0 to n - 1, for 0 <= k <= n.
5 There are n permute k possible arrangements, but n^k possible arrangements if
6 repeated values are allowed.
7
8 - next_arrangement(n, k, a) tries to rearrange a[] to the next lexicographically
9 greater arrangement, returning true if such an arrangement exists or false if
10 the array is already in descending order (in which case a[] is unchanged). The
11 input a[] must consist of exactly k distinct integers in the range [0, n).
12 - arrangement_by_rank(n, k, r) returns the size k arrangement of n which is
13 lexicographically ranked r out of all size k arrangements of n, where r is
14 a zero-based rank in the range [0, n permute k).
15 - rank_by_arrangement(n, k, a) returns an integer representing the zero-based
16 rank of arrangement a[], which must consist of exactly k distinct integers in
17 the range [0, n).
18 - next_arrangement_with_repeats(n, k, a) tries to rearrange a[] to the next
19 lexicographically greater arrangement with repeats, returning true if such an
20 arrangement exists or false if the array is already in descending order (in
21 which case a[] is unchanged). The input a[] must consist of exactly k (not
22 necessarily distinct) integers in the range [0, n). If a[] were interpreted as
23 a k digit integer in base n, this function could be thought of as incrementing
24 the integer.
25
26 Time Complexity:
27 - O(n*k) for next_arrangement(), arrangement_by_rank(), and
28 rank_by_arrangement().
29 - O(k) for next_arrangement_with_repeats().
30
31 Space Complexity:
32 - O(n) auxiliary heap space for next_arrangement(), arrangement_by_rank(), and
33 rank_by_arrangement().
34 - O(1) auxiliary for next_arrangement_with_repeats().
35
36 */
37
38 #include <algorithm>
39 #include <vector>
40
41 bool next_arrangement(int n, int k, int a[]) {
42     std::vector<bool> used(n);
43     for (int i = 0; i < k; i++) {
44         used[a[i]] = true;
45     }
46     for (int i = k - 1; i >= 0; i--) {
47         used[a[i]] = false;
48         for (int j = a[i] + 1; j < n; j++) {
49             if (!used[j]) {
50                 a[i + 1] = j;
51                 used[j] = true;
52                 for (int x = 0; i < k; x++) {
53                     if (!used[x]) {
54                         a[i + 1] = x;
55                     }
56                 }
57                 return true;
58             }
59         }
60     }
61 }
```

```

60     }
61     return false;
62 }
63
64 long long n_permute_k(int n, int k) {
65     long long res = 1;
66     for (int i = 0; i < k; i++) {
67         res *= n - i;
68     }
69     return res;
70 }
71
72 std::vector<int> arrangement_by_rank(int n, int k, long long r) {
73     std::vector<int> values(n), res(k);
74     for (int i = 0; i < n; i++) {
75         values[i] = i;
76     }
77     for (int i = 0; i < k; i++) {
78         long long count = n_permute_k(n - 1 - i, k - 1 - i);
79         int pos = r / count;
80         res[i] = values[pos];
81         std::copy(values.begin() + pos + 1, values.end(), values.begin() + pos);
82         r %= count;
83     }
84     return res;
85 }
86
87 long long rank_by_arrangement(int n, int k, int a[]) {
88     long long res = 0;
89     std::vector<bool> used(n);
90     for (int i = 0; i < k; i++) {
91         int count = 0;
92         for (int j = 0; j < a[i]; j++) {
93             if (!used[j]) {
94                 count++;
95             }
96         }
97         res += count * n_permute_k(n - i - 1, k - i - 1);
98         used[a[i]] = true;
99     }
100    return res;
101 }
102
103 bool next_arrangement_with_repeats(int n, int k, int a[]) {
104     for (int i = k - 1; i >= 0; i--) {
105         if (a[i] < n - 1) {
106             a[i]++;
107             std::fill(a + i + 1, a + k, 0);
108             return true;
109         }
110     }
111     return false;
112 }
113
114 /** Example Usage and Output:
115
116 4 permute 3 arrangements:
117 {0,1,2} {0,1,3} {0,2,1} {0,2,3} {0,3,1} {0,3,2} {1,0,2} {1,0,3} {1,2,0} {1,2,3}
118 {1,3,0} {1,3,2} {2,0,1} {2,0,3} {2,1,0} {2,1,3} {2,3,0} {2,3,1} {3,0,1} {3,0,2}

```

```

119 {3,1,0} {3,1,2} {3,2,0} {3,2,1}
120
121 4^2 arrangements with repeats:
122 {0,0} {0,1} {0,2} {0,3} {1,0} {1,1} {1,2} {1,3} {2,0} {2,1} {2,2} {2,3} {3,0}
123 {3,1} {3,2} {3,3}
124
125 */
126
127 #include <cassert>
128 #include <iostream>
129 using namespace std;
130
131 template<class It>
132 void print_range(It lo, It hi) {
133     cout << "{";
134     for (; lo != hi; ++lo) {
135         cout << *lo << (lo == hi - 1 ? "" : ",");
136     }
137     cout << "}";
138 }
139
140 int main() {
141 {
142     int n = 4, k = 3, a[] = {0, 1, 2};
143     cout << n << " permute " << k << " arrangements:" << endl;
144     int count = 0;
145     do {
146         print_range(a, a + k);
147         vector<int> b = arrangement_by_rank(n, k, count);
148         assert(equal(a, a + k, b.begin()));
149         assert(rank_by_arrangement(n, k, a) == count);
150         count++;
151     } while (next_arrangement(n, k, a));
152     cout << endl;
153 }
154 {
155     int n = 4, k = 2, a[] = {0, 0};
156     cout << endl << n << "^" << k << " arrangements with repeats:" << endl;
157     do {
158         print_range(a, a + k);
159     } while (next_arrangement_with_repeats(n, k, a));
160     cout << endl;
161 }
162     return 0;
163 }
```

5.2.3 Enumerating Permutations

```

1 /*
2
3 A permutation is an ordered list consisting of n (not necessarily distinct)
4 elements.
5
6 - next_permutation_(lo, hi) is analogous to std::next_permutation(lo, hi),
7   taking two BidirectionalIterators lo and hi as a range [lo, hi) for which the
```

```

8  function tries to rearrange to the next lexicographically greater permutation.
9  The function returns true if such a permutation exists, or false if the range
10 is already in descending order (in which case the values are unchanged). This
11 implementation requires an ordering on the set of possible elements defined by
12 the < operator on the iterator's value type.
13 - next_permutation(n, a) is analogous to next_permutation(), except that it
14 takes an array a[] of size n instead of a range.
15 - next_permutation(x) returns the next lexicographically greater permutation of
16 the binary digits of the integer x, that is, the lowest integer greater than
17 x with the same number of 1-bits. This can be used to generate combinations of
18 a set of n items by treating each 1 bit as whether to "take" the item at the
19 corresponding position.
20 - permutation_by_rank(n, r) returns the permutation of the integers in the range
21 [0, n) which is lexicographically ranked r, where r is a zero-based rank in
22 the range [0, n!).
23 - rank_by_permutation(n, a) returns an integer representing the zero-based
24 rank of permutation a[], which must be a permutation of the integers [0, n].
25 - permutation_cycles(n, a) returns the decomposition of the permutation a[] into
26 cycles. A permutation cycle is a subset of a permutation whose elements are
27 consecutively swapped, relative to a sorted set. For example, {3, 1, 0, 2}
28 decomposes to {0, 3, 2} and {1}, meaning that starting from the sorted order
29 {0, 1, 2, 3}, the 0th value is replaced by the 3rd, the 3rd by the 2nd, and
30 the 2nd by the 0th (0 -> 3 -> 2 -> 0).

31
32 Time Complexity:
33 - O(n^2) per call to next_permutation_(lo, hi), where n is the distance between
34 lo and hi.
35 - O(n^2) per call to next_permutation(n, a), permutation_by_rank(n, r), and
36 rank_by_permutation(n, a).
37 - O(1) per call to next_permutation(x).
38 - O(n) per call to permutation_cycles().

39
40 Space Complexity:
41 - O(1) auxiliary for next_permutation_() and next_permutation().
42 - O(n) auxiliary heap space for permutation_by_rank(), rank_by_permutation(),
43 and permutation_cycles().

44 */
45
46
47 #include <algorithm>
48 #include <vector>
49
50 template<class It>
51 bool next_permutation_(It lo, It hi) {
52     if (lo == hi) {
53         return false;
54     }
55     It i = lo;
56     if (++i == hi) {
57         return false;
58     }
59     i = hi;
60     --i;
61     for (;;) {
62         It j = i;
63         if (*--i < *j) {
64             It k = hi;
65             while (!(*i < *--k)) {}
66             std::iter_swap(i, k);

```

```

67     std::reverse(j, hi);
68     return true;
69 }
70 if (i == lo) {
71     std::reverse(lo, hi);
72     return false;
73 }
74 }
75 }

76 template<class T>
77 bool next_permutation(int n, T a[]) {
78     for (int i = n - 2; i >= 0; i--) {
79         if (a[i] < a[i + 1]) {
80             for (int j = n - 1; ; j--) {
81                 if (a[i] < a[j]) {
82                     std::swap(a[i+], a[j]);
83                     for (j = n - 1; i < j; i++, j--) {
84                         std::swap(a[i], a[j]);
85                     }
86                 }
87                 return true;
88             }
89         }
90     }
91     return false;
92 }
93 }

94 long long next_permutation(long long x) {
95     long long s = x & -x, r = x + s;
96     return r | ((x ^ r) >> 2)/s;
97 }

98 std::vector<int> permutation_by_rank(int n, long long x) {
99     std::vector<long long> factorial(n);
100    std::vector<int> values(n), res(n);
101    factorial[0] = 1;
102    for (int i = 1; i < n; i++) {
103        factorial[i] = i*factorial[i - 1];
104    }
105    for (int i = 0; i < n; i++) {
106        values[i] = i;
107    }
108    for (int i = 0; i < n; i++) {
109        int pos = x/factorial[n - 1 - i];
110        res[i] = values[pos];
111        std::copy(values.begin() + pos + 1, values.end(), values.begin() + pos);
112        x %= factorial[n - 1 - i];
113    }
114 }
115 return res;
116 }
117 }

118 long long rank_by_permutation(int n, int a[]) {
119     std::vector<long long> factorial(n);
120     factorial[0] = 1;
121     for (int i = 1; i < n; i++) {
122         factorial[i] = i*factorial[i - 1];
123     }
124     long long res = 0;
125 }
```

```

126     for (int i = 0; i < n; i++) {
127         int v = a[i];
128         for (int j = 0; j < i; j++) {
129             if (a[j] < a[i]) {
130                 v--;
131             }
132         }
133         res += v*factorial[n - 1 - i];
134     }
135     return res;
136 }
137
138 typedef std::vector<std::vector<int>> cycles;
139
140 cycles permutation_cycles(int n, int a[]) {
141     std::vector<bool> visit(n);
142     cycles res;
143     for (int i = 0; i < n; i++) {
144         if (!visit[i]) {
145             int j = i;
146             std::vector<int> curr;
147             do {
148                 curr.push_back(j);
149                 visit[j] = true;
150                 j = a[j];
151             } while (j != i);
152             res.push_back(curr);
153         }
154     }
155     return res;
156 }
157
158 /** Example Usage and Output:
159
160 Permutations of [0, 4]:
161 {0,1,2,3} {0,1,3,2} {0,2,1,3} {0,2,3,1} {0,3,1,2} {0,3,2,1} {1,0,2,3} {1,0,3,2}
162 {1,2,0,3} {1,2,3,0} {1,3,0,2} {1,3,2,0} {2,0,1,3} {2,0,3,1} {2,1,0,3} {2,1,3,0}
163 {2,3,0,1} {2,3,1,0} {3,0,1,2} {3,0,2,1} {3,1,0,2} {3,1,2,0} {3,2,0,1} {3,2,1,0}
164
165 Permutations of 2 zeros and 3 ones:
166 00111 01011 01101 01110 10011 10101 10110 11001 11010 11100
167
168 Decomposition of {3,1,0,2} into cycles:
169 {0,3,2} {1}
170
171 */
172
173 #include <bitset>
174 #include <cassert>
175 #include <iostream>
176 using namespace std;
177
178 template<class It>
179 void print_range(It lo, It hi) {
180     cout << "{";
181     for (; lo != hi; ++lo) {
182         cout << *lo << (lo == hi - 1 ? "" : ",");
183     }
184     cout << "}";

```

```

185 }
186
187 int main() {
188 {
189     const int n = 4;
190     int a[] = {0, 1, 2, 3}, b[n], c[n];
191     for (int i = 0; i < n; i++) {
192         b[i] = c[i] = a[i];
193     }
194     cout << "Permutations of [0, " << n << "):" << endl;
195     int count = 0;
196     do {
197         print_range(a, a + n);
198         assert(equal(b, b + n, a));
199         assert(equal(c, c + n, a));
200         vector<int> d = permutation_by_rank(n, count);
201         assert(equal(d.begin(), d.end(), a));
202         assert(rank_by_permutation(n, a) == count);
203         count++;
204         std::next_permutation(b, b + n);
205         next_permutation(c, c + n);
206     } while (next_permutation(n, a));
207     cout << endl;
208 }
209 // Permutations of binary digits.
210 const int n = 5;
211 cout << "\nPermutations of 2 zeros and 3 ones:" << endl;
212 int lo = bitset<5>(string("00111")).to_ulong();
213 int hi = bitset<6>(string("100011")).to_ulong();
214 do {
215     cout << bitset<n>(lo).to_string() << " ";
216 } while ((lo = next_permutation(lo)) != hi);
217 cout << endl;
218 }
219 // Decomposition into cycles.
220 const int n = 4;
221 int a[] = {3, 1, 0, 2};
222 cout << "\nDecomposition of {3,1,0,2} into cycles:" << endl;
223 cycles c = permutation_cycles(n, a);
224 for (int i = 0; i < (int)c.size(); i++) {
225     print_range(c[i].begin(), c[i].end());
226 }
227 cout << endl;
228 }
229 return 0;
230 }
```

5.2.4 Enumerating Combinations

```

1 /*
2
3 A combination is a subset of size k chosen from a total of n (not necessarily
4 distinct) elements, where order does not matter.
5
6 - next_combination(lo, mid, hi) takes random-access iterators lo, mid, and hi
```

```

7  as a range [lo, hi) of n elements for which the function will rearrange such
8  that the k elements in [lo, mid) becomes the next lexicographically greater
9  combination. The function returns true if such a combination exists, or false
10 if [lo, mid) already consists of the lexicographically greatest combination
11 of the elements in [lo, hi) (in which case the values are unchanged). This
12 implementation requires an ordering on the set of possible elements defined by
13 the < operator on the iterator's value type.
14 - next_combination(n, k, a) rearranges a[] to become the next lexicographically
15 greater combination of k distinct integers in the range [0, n). The array a[]
16 must consist of k distinct integers in the range [0, n].
17 - next_combination_mask(x) interprets the bits of an integer x as a mask with
18 1-bits specifying the chosen items for a combination and returns the mask of
19 the next lexicographically greater combination (that is, the lowest integer
20 greater than x with the same number of 1 bits). Note that this does not
21 generate combinations in the same order as next_combination(), nor does it
22 work if the corresponding n items are not distinct (in that case, duplicate
23 combinations will be generated).
24 - combination_by_rank(n, k, r) returns the combination of k distinct integers in
25 the range [0, n) that is lexicographically ranked r, where r is a zero-based
26 rank in the range [0, n choose k].
27 - rank_by_combination(n, k, a) returns an integer representing the zero-based
28 rank of combination a[], which must consist of k distinct integers in [0, n].
29 - next_combination_with_repeats(n, k, a) rearranges a[] to become the next
30 lexicographically greater combination of k (not necessarily distinct) integers
31 in the range [0, n). The array a[] must consist of k integers in the range
32 [0, n). Note that there is a total of n multichoose k combinations if
33 repetition is allowed, where n multichoose k = (n + k - 1) choose k.
34
35 Time Complexity:
36 - O(n) per call to next_combination(lo, hi), where n is the distance between
37 lo and hi.
38 - O(k) per call to next_combination(n, k, a) and
39 next_combination_with_repeats(n, k, a).
40 - O(1) per call to next_combination_mask(x).
41 - O(n*k) per call to combination_by_rank() and rank_by_combination().
42
43 Space Complexity:
44 - O(k) auxiliary heap space for combination_by_rank(n, k, r).
45 - O(1) auxiliary for all other operations.
46
47 */
48
49 #include <algorithm>
50 #include <iterator>
51 #include <vector>
52
53 template<class It>
54 bool next_combination(It lo, It mid, It hi) {
55     if (lo == mid || mid == hi) {
56         return false;
57     }
58     It l = mid - 1, h = hi - 1;
59     int len1 = 1, len2 = 1;
60     while (l != lo && !(l < *h)) {
61         --l;
62         ++len1;
63     }
64     if (l == lo && !(l < *h)) {
65         std::rotate(lo, mid, hi);

```

```

66     return false;
67 }
68 for ( ; mid < h; ++len2) {
69     if (!(*l < *--h)) {
70         ++h;
71         break;
72     }
73 }
74 if (len1 == 1 || len2 == 1) {
75     std::iter_swap(l, h);
76 } else if (len1 == len2) {
77     std::swap_ranges(l, mid, h);
78 } else {
79     std::iter_swap(l++, h++);
80     int total = (--len1) + (--len2), gcd = total;
81     for (int i = len1; i != 0; ) {
82         std::swap(gcd %= i, i);
83     }
84     int skip = total/gcd - 1;
85     for (int i = 0; i < gcd; i++) {
86         It curr = (i < len1) ? (l + i) : (h + (i - len1));
87         int k = i;
88         typename std::iterator_traits<It>::value_type prev(*curr);
89         for (int j = 0; j < skip; j++) {
90             k = (k + len1) % total;
91             It next = (k < len1) ? (l + k) : (h + (k - len1));
92             *curr = *next;
93             curr = next;
94         }
95         *curr = prev;
96     }
97 }
98 return true;
99 }
100
101 bool next_combination(int n, int k, int a[]) {
102     for (int i = k - 1; i >= 0; i--) {
103         if (a[i] < n - k + i) {
104             a[i]++;
105             while (++i < k) {
106                 a[i] = a[i - 1] + 1;
107             }
108             return true;
109         }
110     }
111     return false;
112 }
113
114 long long next_combination_mask(long long x) {
115     long long s = x & -x, r = x + s;
116     return r | (((x ^ r) >> 2)/s);
117 }
118
119 long long n_choose_k(long long n, long long k) {
120     if (k > n - k) {
121         k = n - k;
122     }
123     long long res = 1;
124     for (int i = 0; i < k; i++) {

```

```

125     res = res*(n - i)/(i + 1);
126 }
127 return res;
128 }
129
130 std::vector<int> combination_by_rank(int n, int k, long long r) {
131     std::vector<int> res(k);
132     int count = n;
133     for (int i = 0; i < k; i++) {
134         int j = 1;
135         for (;;) j++ {
136             long long am = n_choose_k(count - j, k - 1 - i);
137             if (r < am) {
138                 break;
139             }
140             r -= am;
141         }
142         res[i] = (i > 0) ? (res[i - 1] + j) : (j - 1);
143         count -= j;
144     }
145     return res;
146 }
147
148 long long rank_by_combination(int n, int k, int a[]) {
149     long long res = 0;
150     int prev = -1;
151     for (int i = 0; i < k; i++) {
152         for (int j = prev + 1; j < a[i]; j++) {
153             res += n_choose_k(n - 1 - j, k - 1 - i);
154         }
155         prev = a[i];
156     }
157     return res;
158 }
159
160 bool next_combination_with_repeats(int n, int k, int a[]) {
161     for (int i = k - 1; i >= 0; i--) {
162         if (a[i] < n - 1) {
163             for (++a[i]; ++i < k; ) {
164                 a[i] = a[i - 1];
165             }
166             return true;
167         }
168     }
169     return false;
170 }
171
172 /** Example Usage and Output:
173
174 "11234" choose 3:
175 112 113 114 123 124 134 234
176
177 "abcde" choose 3 with masks:
178 abc abd acd bcd abe ace bce ade bde
179
180 5 choose 3:
181 {0,1,2} {0,1,3} {0,1,4} {0,2,3} {0,2,4} {0,3,4} {1,2,3} {1,2,4} {1,3,4} {2,3,4}
182
183 3 multichoose 2:

```

```

184 {0,0} {0,1} {0,2} {1,1} {1,2} {2,2}
185
186 /**
187
188 #include <cassert>
189 #include <iostream>
190 using namespace std;
191
192 template<class It>
193 void print_range(It lo, It hi) {
194     cout << "[";
195     for (; lo != hi; ++lo) {
196         cout << *lo << (lo == hi - 1 ? "" : ",");
197     }
198     cout << "]";
199 }
200
201 int main() {
202 {
203     int k = 3;
204     string s = "11234";
205     cout << "\n" << s << "\ choose " << k << ":" << endl;
206     do {
207         cout << s.substr(0, k) << " ";
208     } while (next_combination(s.begin(), s.begin() + k, s.end()));
209     cout << endl;
210 }
211 { // Unordered combinations using masks.
212     int n = 5, k = 3;
213     string char_set = "abcde"; // Must be distinct.
214     cout << "\n" << char_set << "\ choose " << k << " with masks:" << endl;
215     long long mask = 0, dest = 0;
216     for (int i = 0; i < k; i++) {
217         mask |= (1 << i);
218     }
219     for (int i = k - 1; i < n; i++) {
220         dest |= (1 << i);
221     }
222     do {
223         for (int i = 0; i < n; i++) {
224             if ((mask >> i) & 1) {
225                 cout << char_set[i];
226             }
227         }
228         cout << " ";
229         mask = next_combination_mask(mask);
230     } while (mask != dest);
231     cout << endl;
232 }
233 { // Combinations of distinct integers from 0 to n - 1.
234     int n = 5, k = 3, a[] = {0, 1, 2};
235     cout << "\n" << n << " choose " << k << ":" << endl;
236     int count = 0;
237     do {
238         print_range(a, a + k);
239         vector<int> b = combination_by_rank(n, k, count);
240         assert(equal(a, a + k, b.begin()));
241         assert(rank_by_combination(n, k, a) == count);
242         count++;

```

```

243     } while (next_combination(n, k, a));
244     cout << endl;
245 }
246 { // Combinations with repeats.
247     int n = 3, k = 2, a[] = {0, 0};
248     cout << "\n" << n << " multichoose " << k << ":" << endl;
249     do {
250         print_range(a, a + k);
251     } while (next_combination_with_repeats(n, k, a));
252     cout << endl;
253 }
254 return 0;
255 }
```

5.2.5 Enumerating Partitions

```

1 /*
2
3 A partition of a natural number n is a way to write n as a sum of positive
4 integers where the order of the addends does not matter.
5
6 - next_partition(p) takes a reference to a vector p[] of positive integers as a
7   partition of n for which the function will re-assign to become the next
8   lexicographically greater partition. The function returns true if such a
9   partition exists, or false if p[] already consists of the lexicographically
10  greatest partition (i.e. the single integer n).
11 - partition_by_rank(n, r) returns the partition of n that is lexicographically
12  ranked r if addends in each partition were sorted in non-increasing order,
13  where r is a zero-based rank in the range [0, partitions(n)).
14 - rank_by_partition(p) returns an integer representing the zero-based rank of
15  the partition specified by vector p[], which must consist of positive integers
16  sorted in non-increasing order.
17 - generate_increasing_partitions(n, f) calls the function f(lo, hi) on strictly
18  increasing partitions of n in lexicographically increasing order of partition,
19  where lo and hi are random-access iterators to a range [lo, hi) of integers.
20 Note that non-strictly increasing partitions like {1, 1, 1, 1} are skipped.
21
22 Time Complexity:
23 - O(n) per call to next_partition().
24 - O(n^2) per call to partition_by_rank(n, r) and rank_by_partition(p).
25 - O(p(n)) per call to generate_increasing_partitions(n, f), where p(n) is the
26  number of partitions of n.
27
28 Space Complexity:
29 - O(1) auxiliary for next_partition().
30 - O(n^2) auxiliary heap space for partition_function(), partition_by_rank(), and
31  rank_by_partition().
32 - O(n) auxiliary stack space for generate_increasing_partitions().
33
34 */
35
36 #include <vector>
37
38 bool next_partition(std::vector<int> &p) {
39     int n = p.size();
```

```

40     if (n <= 1) {
41         return false;
42     }
43     int s = p[n - 1] - 1, i = n - 2;
44     p.pop_back();
45     for (; i > 0 && p[i] == p[i - 1]; i--) {
46         s += p[i];
47         p.pop_back();
48     }
49     for (p[i]++; s > 0; s--) {
50         p.push_back(1);
51     }
52     return true;
53 }
54
55 long long partition_function(int a, int b) {
56     static std::vector<std::vector<long long>> p(
57         1, std::vector<long long>(1, 1));
58     if (a >= (int)p.size()) {
59         int old = p.size();
60         p.resize(a + 1);
61         p[0].resize(a + 1);
62         for (int i = 1; i <= a; i++) {
63             p[i].resize(a + 1);
64             for (int j = old; j <= i; j++) {
65                 p[i][j] = p[i - 1][j - 1] + p[i - j][j];
66             }
67         }
68     }
69     return p[a][b];
70 }
71
72 std::vector<int> partition_by_rank(int n, long long r) {
73     std::vector<int> res;
74     for (int i = n, j; i > 0; i -= j) {
75         for (j = 1; ; j++) {
76             long long count = partition_function(i, j);
77             if (r < count) {
78                 break;
79             }
80             r -= count;
81         }
82         res.push_back(j);
83     }
84     return res;
85 }
86
87 long long rank_by_partition(const std::vector<int> &p) {
88     long long res = 0;
89     int sum = 0;
90     for (int i = 0; i < (int)p.size(); i++) {
91         sum += p[i];
92     }
93     for (int i = 0; i < (int)p.size(); i++) {
94         for (int j = 0; j < p[i]; j++) {
95             res += partition_function(sum, j);
96         }
97         sum -= p[i];
98     }

```

```

99     return res;
100 }
101
102 typedef void (*ReportFunction)(std::vector<int>::iterator,
103                                 std::vector<int>::iterator);
104
105 void generate_increasing_partitions(int left, int prev, int i,
106                                     std::vector<int> &p, ReportFunction f) {
107     if (left == 0) {
108         f(p.begin(), p.begin() + i);
109         return;
110     }
111     for (p[i] = prev + 1; p[i] <= left; p[i]++) {
112         generate_increasing_partitions(left - p[i], p[i], i + 1, p, f);
113     }
114 }
115
116 void generate_increasing_partitions(int n, ReportFunction f) {
117     std::vector<int> p(n, 0);
118     generate_increasing_partitions(n, 0, 0, p, f);
119 }
120
121 /** Example Usage and Output:
122
123 Partitions of 4:
124 {1,1,1,1} {2,1,1} {2,2} {3,1} {4}
125
126 Increasing partitions of 8:
127 {1,2,5} {1,3,4} {1,7} {2,6} {3,5} {8}
128
129 **/
130
131 #include <cassert>
132 #include <iostream>
133 using namespace std;
134
135 template<class It>
136 void print_range(It lo, It hi) {
137     cout << "{" ;
138     for (; lo != hi; ++lo) {
139         cout << *lo << (lo == hi - 1 ? "" : ",");
140     }
141     cout << "} ";
142 }
143
144 int main() {
145     {
146         int n = 4;
147         vector<int> a(n, 1);
148         cout << "Partitions of " << n << ":" << endl;
149         int count = 0;
150         do {
151             print_range(a.begin(), a.end());
152             vector<int> b = partition_by_rank(n, count);
153             assert(equal(a.begin(), a.end(), b.begin()));
154             assert(rank_by_partition(a) == count);
155             count++;
156         } while (next_partition(a));
157         cout << endl;

```

```

158     }
159     {
160         int n = 8;
161         cout << "\nIncreasing partitions of " << n << ":" << endl;
162         generate_increasing_partitions(n, print_range);
163         cout << endl;
164     }
165     return 0;
166 }
```

5.2.6 Enumerating Generic Combinatorial Sequences

```

1  /*
2
3  Enumerate combinatorial sequence by inheriting an abstract class. Child classes
4  of abstract_enumerator must implement the count() function which should return
5  the number of combinatorial sequences starting with the given prefix.
6
7  - to_rank(a) returns an integer representing the zero-based rank of the
8  combinatorial sequence a.
9  - from_rank(r) returns a combinatorial sequence of integers that is
10  lexicographically ranked r, where r is a zero-based rank in the range
11  [0, total_count()].
12  - enumerate(f) calls the function f(lo, hi) on every specified combinatorial
13  sequence in lexicographically increasing order, where lo and hi are two
14  random-access iterators to a range [lo, hi) of integers.
15
16 Time Complexity:
17 - O(n^2) calls will be made to count() per call to all operations, where n is
18 the length of the combinatorial sequence.
19
20 Space Complexity:
21 - O(n) auxiliary heap space per call to all operations.
22
23 */
24
25 #include <vector>
26
27 class abstract_enumerator {
28 protected:
29     int range, length;
30
31     abstract_enumerator(int r, int l) : range(r), length(l) {}
32
33     virtual long long count(const std::vector<int> &prefix) {
34         return 0;
35     }
36
37     std::vector<int> next(std::vector<int> &a) {
38         return from_rank(to_rank(a) + 1);
39     }
40
41     long long total_count() {
42         return count(std::vector<int>(0));
43     }

```

```

44
45 public:
46 long long to_rank(const std::vector<int> &a) {
47     long long res = 0;
48     for (int i = 0; i < (int)a.size(); i++) {
49         std::vector<int> prefix(a.begin(), a.end());
50         prefix.resize(i + 1);
51         for (prefix[i] = 0; prefix[i] < a[i]; prefix[i]++) {
52             res += count(prefix);
53         }
54     }
55     return res;
56 }
57
58 std::vector<int> from_rank(long long r) {
59     std::vector<int> a(length);
60     for (int i = 0; i < (int)a.size(); i++) {
61         std::vector<int> prefix(a.begin(), a.end());
62         prefix.resize(i + 1);
63         for (prefix[i] = 0; prefix[i] < range; ++prefix[i]) {
64             long long curr = count(prefix);
65             if (r < curr) {
66                 break;
67             }
68             r -= curr;
69         }
70         a[i] = prefix[i];
71     }
72     return a;
73 }
74
75 void enumerate(void (*f)(std::vector<int>::iterator,
76                         std::vector<int>::iterator)) {
77     long long total = total_count();
78     for (long long i = 0; i < total; i++) {
79         std::vector<int> curr = from_rank(i);
80         f(curr.begin(), curr.end());
81     }
82 }
83 };
84
85 class arrangement_enumerator : public abstractEnumerator {
86 public:
87     arrangement_enumerator(int n, int k) : abstractEnumerator(n, k) {}
88
89     long long count(const std::vector<int> &prefix) {
90         int n = prefix.size();
91         for (int i = 0; i < n - 1; i++) {
92             if (prefix[i] == prefix[n - 1]) {
93                 return 0;
94             }
95         }
96         long long res = 1;
97         for (int i = 0; i < length - n; i++) {
98             res *= range - n - i;
99         }
100        return res;
101    }
102};

```

```
103
104 class permutation_enumerator : public arrangement_enumerator {
105 public:
106     permutation_enumerator(int n) : arrangement_enumerator(n, n) {}
107 };
108
109 class combination_enumerator : public abstractEnumerator {
110     std::vector<std::vector<long long>> table;
111
112 public:
113     combination_enumerator(int n, int k)
114         : abstractEnumerator(n, k), table(n + 1, std::vector<long long>(n + 1)) {
115         for (int i = 0; i <= n; i++) {
116             for (int j = 0; j <= i; j++) {
117                 table[i][j] = (j == 0) ? 1 : table[i - 1][j - 1] + table[i - 1][j];
118             }
119         }
120     }
121
122     long long count(const std::vector<int> &prefix) {
123         int n = prefix.size();
124         if (n >= 2 && prefix[n - 1] <= prefix[n - 2]) {
125             return 0;
126         }
127         if (n == 0) {
128             return table[range][length - n];
129         }
130         return table[range - prefix[n - 1] - 1][length - n];
131     }
132 };
133
134 class partition_enumerator : public abstractEnumerator {
135     std::vector<std::vector<long long>> table;
136
137 public:
138     partition_enumerator(int n) : abstractEnumerator(n + 1, n),
139                                 table(n + 1, std::vector<long long>(n + 1)) {
140         std::vector<std::vector<long long>> tmp(table);
141         tmp[0][0] = 1;
142         for (int i = 1; i <= n; i++) {
143             for (int j = 1; j <= i; j++) {
144                 tmp[i][j] = tmp[i - 1][j - 1] + tmp[i - j][j];
145             }
146         }
147         for (int i = 1; i <= n; i++) {
148             for (int j = 1; j <= n; j++) {
149                 table[i][j] = tmp[i][j] + table[i][j - 1];
150             }
151         }
152     }
153
154     long long count(const std::vector<int> &prefix) {
155         int n = (int)prefix.size(), sum = 0;
156         for (int i = 0; i < n; i++) {
157             sum += prefix[i];
158         }
159         if (sum == range - 1) {
160             return 1;
161         }
162     }
163 }
```

```

162     if (sum > range - 1 || (n > 0 && prefix[n - 1] == 0) ||
163         (n >= 2 && prefix[n - 1] > prefix[n - 2])) {
164         return 0;
165     }
166     if (n == 0) {
167         return table[range - sum - 1][range - 1];
168     }
169     return table[range - sum - 1][prefix[n - 1]];
170 }
171 };
172
173 /*** Example Usage and Output:
174
175 3 permute 2 arrangements:
176 {0,1} {0,2} {1,0} {1,2} {2,0} {2,1}
177
178 Permutatations of [0, 3):
179 {0,1,2} {0,2,1} {1,0,2} {1,2,0} {2,0,1} {2,1,0}
180
181 4 choose 3 combinations:
182 {0,1,2} {0,1,3} {0,2,3} {1,2,3}
183
184 Partition of 4:
185 {1,1,1,1} {2,1,1,0} {2,2,0,0} {3,1,0,0} {4,0,0,0}
186
187 */
188
189 #include <iostream>
190 using namespace std;
191
192 template<class It>
193 void print_range(It lo, It hi) {
194     cout << "{";
195     for (; lo != hi; ++lo) {
196         cout << *lo << (lo == hi - 1 ? "" : ",");
197     }
198     cout << "}";
199 }
200
201 int main() {
202 {
203     cout << "3 permute 2 arrangement_enumerator:" << endl;
204     arrangement_enumerator arr(3, 2);
205     arr.enumerate(print_range);
206     cout << endl;
207 }
208 {
209     cout << "\nPermutatations of [0, 3):" << endl;
210     permutation_enumerator perm(3);
211     perm.enumerate(print_range);
212     cout << endl;
213 }
214 {
215     cout << "\n4 choose 3 combinations:" << endl;
216     combination_enumerator comb(4, 3);
217     comb.enumerate(print_range);
218     cout << endl;
219 }
220 {

```

```

221     cout << "\nPartition of 4:" << endl;
222     partition_enumerator part(4);
223     part.enumerate(print_range);
224     cout << endl;
225 }
226 return 0;
227 }
```

5.3 Number Theory

5.3.1 GCD, LCM, Mod Inverse, Chinese Remainder

```

1  /*
2
3 Common number theory operations relating to modular arithmetic.
4
5 - gcd(a, b) and gcd2(a, b) both return the greatest common division of a and b
6   using the Euclidean algorithm.
7 - lcm(a, b) returns the lowest common multiple of a and b.
8 - extended_euclid(a, b) and extended_euclid2(a, b) both return a pair (x, y) of
9   integers such that gcd(a, b) = a*x + b*y.
10 - mod(a, b) returns the value of a mod b under the true Euclidean definition of
11   modulo, that is, the smallest nonnegative integer m satisfying a + b*n = m for
12   some integer n. Note that this is identical to the remainder operator % in C++
13   for nonnegative operands a and b, but the result will differ when an operand
14   is negative.
15 - mod_inverse(a, m) and mod_inverse2(a, m) both return an integer x such that
16   a*x = 1 (mod m), where the arguments must satisfy m > 0 and gcd(a, m) = 1.
17 - generate_inverse(p) returns a vector of integers where for each index i in
18   the vector, i*v[i] = 1 (mod p), where the argument p is prime.
19 - simple_restore(n, a, p) and garner_restore(n, a, p) both return the solution x
20   for the system of simultaneous congruences x = a[i] (mod p[i]) for all indices
21   i in [0, n), where p[] consist of pairwise coprime integers. The solution x is
22   guaranteed to be unique by the Chinese remainder theorem.
23
24 Time Complexity:
25 - O(log(a + b)) per call to gcd(a, b), gcd2(a, b), lcm(a, b),
26   extended_euclid(a, b), extended_euclid2(a, b), mod_inverse(a, b), and
27   mod_inverse2(a, b).
28 - O(1) for mod(a, b).
29 - O(p) for generate_inverse(p).
30 - Exponential for simple_restore(n, a, p).
31 - O(n^2) for garner_restore(n, a, p).
32
33 Space Complexity:
34 - O(log(a + b)) auxiliary stack space for gcd2(a, b), extended_euclid2(a, b),
35   and mod_inverse2(a, b).
36 - O(p) auxiliary heap space for generate_inverse(p).
37 - O(n) auxiliary heap space for garner_restore(n, a, p).
38 - O(1) auxiliary space for all other operations.
39
40 */
41
```

```

42 #include <cstdlib>
43 #include <utility>
44 #include <vector>
45
46 template<class Int>
47 Int gcd(Int a, Int b) {
48     while (b != 0) {
49         Int t = b;
50         b = a % b;
51         a = t;
52     }
53     return abs(a);
54 }
55
56 template<class Int>
57 Int gcd2(Int a, Int b) {
58     return (b == 0) ? abs(a) : gcd(b, a % b);
59 }
60
61 template<class Int>
62 Int lcm(Int a, Int b) {
63     return abs(a / gcd(a, b) * b);
64 }
65
66 template<class Int>
67 std::pair<Int, Int> extended_euclid(Int a, Int b) {
68     Int x = 1, y = 0, x1 = 0, y1 = 1;
69     while (b != 0) {
70         Int q = a/b, prev_x1 = x1, prev_y1 = y1, prev_b = b;
71         x1 = x - q*x1;
72         y1 = y - q*y1;
73         b = a - q*b;
74         x = prev_x1;
75         y = prev_y1;
76         a = prev_b;
77     }
78     return (a > 0) ? std::make_pair(x, y) : std::make_pair(-x, -y);
79 }
80
81 template<class Int>
82 std::pair<Int, Int> extended_euclid2(Int a, Int b) {
83     if (b == 0) {
84         return (a > 0) ? std::make_pair(1, 0) : std::make_pair(-1, 0);
85     }
86     std::pair<Int, Int> r = extended_euclid2(b, a % b);
87     return std::make_pair(r.second, r.first - a/b*r.second);
88 }
89
90 template<class Int>
91 Int mod(Int a, Int m) {
92     Int r = a % m;
93     return (r >= 0) ? r : (r + m);
94 }
95
96 template<class Int>
97 Int mod_inverse(Int a, Int m) {
98     a = mod(a, m);
99     return (a == 0) ? 0 : mod((1 - m*mod_inverse(m % a, a)) / a, m);
100 }
```

```

101
102 template<class Int>
103 Int mod_inverse2(Int a, Int m) {
104     return mod(extended_euclid(a, m).first, m);
105 }
106
107 std::vector<int> generate_inverse(int p) {
108     std::vector<int> res(p);
109     res[1] = 1;
110     for (int i = 2; i < p; i++) {
111         res[i] = (p - (p / i)*res[p % i] % p) % p;
112     }
113     return res;
114 }
115
116 long long simple_restore(int n, int a[], int p[]) {
117     long long res = 0, m = 1;
118     for (int i = 0; i < n; i++) {
119         while (res % p[i] != a[i]) {
120             res += m;
121         }
122         m *= p[i];
123     }
124     return res;
125 }
126
127 long long garner_restore(int n, int a[], int p[]) {
128     std::vector<int> x(a, a + n);
129     for (int i = 0; i < n; i++) {
130         for (int j = 0; j < i; j++) {
131             x[i] = mod_inverse((long long)p[j], (long long)p[i])*x[i] - x[j];
132         }
133         x[i] = (x[i] % p[i] + p[i]) % p[i];
134     }
135     long long res = x[0], m = 1;
136     for (int i = 1; i < n; i++) {
137         m *= p[i - 1];
138         res += x[i] * m;
139     }
140     return res;
141 }
142
143 /** Example Usage ***/
144
145 #include <cassert>
146 using namespace std;
147
148 int main() {
149 {
150     for (int steps = 0; steps < 10000; steps++) {
151         int a = rand() % 200 - 10, b = rand() % 200 - 10, g = gcd(a, b);
152         assert(g == gcd2(a, b));
153         if (g == 1 && b > 1) {
154             int inv = mod_inverse(a, b);
155             assert(inv == mod_inverse2(a, b) && mod(a*inv, b) == 1);
156         }
157         pair<int, int> res = extended_euclid(a, b);
158         assert(res == extended_euclid2(a, b));
159         assert(g == a*res.first + b*res.second);

```

```

160     }
161 }
162 {
163     int p = 17;
164     std::vector<int> res = generate_inverse(p);
165     for (int i = 0; i < p; i++) {
166         if (i > 0) {
167             assert(mod(i*res[i], p) == 1);
168         }
169     }
170 }
171 {
172     int n = 3, a[] = {2, 3, 1}, m[] = {3, 4, 5};
173     int x = simple_restore(n, a, m);
174     assert(x == garner_restore(n, a, m));
175     for (int i = 0; i < n; i++) {
176         assert(mod(x, m[i]) == a[i]);
177     }
178     assert(x == 11);
179 }
180     return 0;
181 }
```

5.3.2 Prime Generation

```

1 /*
2
3 Generate prime numbers using the Sieve of Eratosthenes.
4
5 - sieve(n) returns a vector of all the primes less than or equal to n.
6 - sieve(lo, hi) returns a vector of all the primes in the range [lo, hi].
7
8 Time Complexity:
9 - O(n log(log(n))) per call to sieve(n).
10 - O(sqrt(hi)*log(log(hi - lo))) per call to sieve(lo, hi).
11
12 Space Complexity:
13 - O(n) auxiliary heap space per call to sieve(n).
14 - O(hi - lo + sqrt(hi)) auxiliary heap space per call to sieve(lo, hi).
15
16 */
17
18 #include <cmath>
19 #include <vector>
20
21 std::vector<int> sieve(int n) {
22     std::vector<bool> prime(n + 1, true);
23     int sqrt_n = ceil(sqrt(n));
24     for (int i = 2; i <= sqrt_n; i++) {
25         if (prime[i]) {
26             for (int j = i*i; j <= n; j += i) {
27                 prime[j] = false;
28             }
29         }
30     }
```

```
31     std::vector<int> res;
32     for (int i = 2; i <= n; i++) {
33         if (prime[i]) {
34             res.push_back(i);
35         }
36     }
37     return res;
38 }
39
40 std::vector<int> sieve(int lo, int hi) {
41     int sqrt_hi = ceil(sqrt(hi)), fourth_root_hi = ceil(sqrt(sqrt_hi));
42     std::vector<bool> prime1(sqrt_hi + 1, true), prime2(hi - lo + 1, true);
43     for (int i = 2; i <= fourth_root_hi; i++) {
44         if (prime1[i]) {
45             for (int j = i*i; j <= sqrt_hi; j += i) {
46                 prime1[j] = false;
47             }
48         }
49     }
50     for (int i = 2, n = hi - lo; i <= sqrt_hi; i++) {
51         if (prime1[i]) {
52             for (int j = (lo / i)*i - lo; j <= n; j += i) {
53                 if (j >= 0 && j + lo != i) {
54                     prime2[j] = false;
55                 }
56             }
57         }
58     }
59     std::vector<int> res;
60     for (int i = (lo > 1) ? lo : 2; i <= hi; i++) {
61         if (prime2[i - lo]) {
62             res.push_back(i);
63         }
64     }
65     return res;
66 }
67
68 /** Example Usage and Output:
69
70 sieve(n=10000000): 0.059s
71 atkins(n=10000000): 0.08s
72 sieve([1000000000, 1005000000]): 0.034s
73
74 */
75
76 #include <ctime>
77 #include <iostream>
78 using namespace std;
79
80 int main() {
81     int pmax = 10000000;
82     vector<int> p;
83     time_t start;
84     double delta;
85
86     start = clock();
87     p = sieve(pmax);
88     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
89     cout << "sieve(n=" << pmax << "): " << delta << "s" << endl;
```

```

90
91     int l = 1000000000, h = 1005000000;
92     start = clock();
93     p = sieve(l, h);
94     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
95     cout << "sieve([" << l << ", " << h << "]): " << delta << "s" << endl;
96     return 0;
97 }
```

5.3.3 Primality Testing

```

1  /*
2
3 Determine whether an integer n is prime. This can be done deterministically by
4 testing all numbers under sqrt(n) using trial division, probabilistically using
5 the Miller-Rabin test, or deterministically using the Miller-Rabin test if the
6 maximum input is known ( $2^{63} - 1$  for the purposes here).
7
8 - is_prime(n) returns whether the integer n is prime using an optimized trial
9   division technique based on the fact that all primes greater than 6 must take
10  the form  $6n + 1$  or  $6n - 1$ .
11 - is_probable_prime(n, k) returns true if the integer n is prime, or false with
12  an error probability of  $(1/4)^k$  if n is composite. In other words, the result
13  is guaranteed to be correct if n is prime, but could be wrong with probability
14   $(1/4)^k$  if n is composite. This implementation uses exponentiation by
15  squaring to support all signed 64-bit integers (up to and including  $2^{63} - 1$ ).
16 - is_prime_fast(n) returns whether the signed 64-bit integer n is prime using
17  a fully deterministic version of the Miller-Rabin test.
18
19 Time Complexity:
20 -  $O(\sqrt{n})$  per call to is_prime(n).
21 -  $O(k \log^3(n))$  per call to is_probable_prime(n, k).
22 -  $O(\log^3(n))$  per call to is_prime_fast(n).
23
24 Space Complexity:
25 -  $O(1)$  auxiliary space for all operations.
26
27 */
28
29 #include <cstdlib>
30
31 template<class Int>
32 bool is_prime(Int n) {
33     if (n == 2 || n == 3) {
34         return true;
35     }
36     if (n < 2 || n % 2 == 0 || n % 3 == 0) {
37         return false;
38     }
39     for (Int i = 5, w = 4; i*i <= n; i += w) {
40         if (n % i == 0) {
41             return false;
42         }
43         w = 6 - w;
44     }
}
```

```
45     return true;
46 }
47
48 typedef unsigned long long uint64;
49
50 uint64 mulmod(uint64 x, uint64 n, uint64 m) {
51     uint64 a = 0, b = x % m;
52     for (; n > 0; n >>= 1) {
53         if (n & 1) {
54             a = (a + b) % m;
55         }
56         b = (b << 1) % m;
57     }
58     return a % m;
59 }
60
61 uint64 powmod(uint64 x, uint64 n, uint64 m) {
62     uint64 a = 1, b = x;
63     for (; n > 0; n >>= 1) {
64         if (n & 1) {
65             a = mulmod(a, b, m);
66         }
67         b = mulmod(b, b, m);
68     }
69     return a % m;
70 }
71
72 uint64 rand64u() {
73     return ((uint64)(rand() & 0xf) << 60) |
74         ((uint64)(rand() & 0x7fff) << 45) |
75         ((uint64)(rand() & 0x7fff) << 30) |
76         ((uint64)(rand() & 0x7fff) << 15) |
77         ((uint64)(rand() & 0x7fff));
78 }
79
80 bool is_probable_prime(long long n, int k = 20) {
81     if (n == 2 || n == 3) {
82         return true;
83     }
84     if (n < 2 || n % 2 == 0 || n % 3 == 0) {
85         return false;
86     }
87     uint64 s = n - 1, p = n - 1;
88     while (!(s & 1)) {
89         s >>= 1;
90     }
91     for (int i = 0; i < k; i++) {
92         uint64 x, r = powmod(rand64u() % p + 1, s, n);
93         for (x = s; x != p && r != 1 && r != p; x <= 1) {
94             r = mulmod(r, r, n);
95         }
96         if (r != p && !(x & 1)) {
97             return false;
98         }
99     }
100    return true;
101 }
102
103 bool is_prime_fast(long long n) {
```

```

104     static const int np = 9, p[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
105     for (int i = 0; i < np; i++) {
106         if (n % p[i] == 0) {
107             return n == p[i];
108         }
109     }
110     if (n < p[np - 1]) {
111         return false;
112     }
113     uint64 t;
114     int s = 0;
115     for (t = n - 1; !(t & 1); t >>= 1) {
116         s++;
117     }
118     for (int i = 0; i < np; i++) {
119         uint64 r = powmod(p[i], t, n);
120         if (r == 1) {
121             continue;
122         }
123         bool ok = false;
124         for (int j = 0; j < s && !ok; j++) {
125             ok |= (r == (uint64)n - 1);
126             r = mulmod(r, r, n);
127         }
128         if (!ok) {
129             return false;
130         }
131     }
132     return true;
133 }
134 /**
135  *** Example Usage ***
136
137 #include <cassert>
138
139 int main() {
140     int len = 20;
141     long long tests[] = {
142         -1, 0, 1, 2, 3, 4, 5, 1000000LL, 772023803LL, 792904103LL, 813815117LL,
143         834753187LL, 855718739LL, 876717799LL, 897746119LL, 2147483647LL,
144         5705234089LL, 5914686649LL, 6114145249LL, 6339503641LL, 6548531929LL
145     };
146     for (int i = 0; i < len; i++) {
147         bool p = is_prime(tests[i]);
148         assert(p == is_prime_fast(tests[i]));
149         assert(p == is_probable_prime(tests[i]));
150     }
151     return 0;
152 }
```

5.3.4 Integer Factorization

```

1 /*
2
3 Compute the prime factorization of an integer. In the following implementations,
```

```

4 the prime factorization of n is represented as a sorted vector of prime integers
5 which together multiply to n. Note that factors are duplicated in the vector in
6 accordance to their multiplicity in the prime factorization of n. For 0, 1, and
7 prime numbers, the prime factorization is considered to be a vector consisting
8 of a single element - the input itself.
9
10 - prime_factorize(n) returns the prime factorization of n using trial division.
11 - get_divisors(n) returns a sorted vector of all (not merely prime) divisors of
12 n using trial division.
13 - fermat(n) returns a factor of n (possibly 1 or itself) that is not necessarily
14 prime. This algorithm is efficient for integers with two factors near sqrt(n),
15 but is roughly as slow as trial division otherwise.
16 - pollards_rho_brent(n) returns a factor of n that is not necessarily prime
17 using Pollard's rho algorithm with Brent's optimization. If n is prime, then n
18 itself is returned. While this algorithm is non-deterministic and may fail to
19 detect factors on certain runs of the same input, it can be placed in a loop
20 to deterministically factor large integers, as done in prime_factorize_big().
21 - prime_factorize_big(n, trial_division_cutoff) returns the prime factorization
22 of a 64-bit integer n using a combination of trial division, the Miller-Rabin
23 primality test, and Pollard's rho algorithm. trial_division_cutoff specifies
24 the largest factor to test with trial division before falling back to the rho
25 algorithm. This supports 64-bit integers up to and including  $2^{63} - 1$ .
26
27 Time Complexity:
28 -  $O(\sqrt{n})$  per call to prime_factorize(n), get_divisors(n), and fermat(n).
29 - Unknown, but approximately  $O(n^{1/4})$  per call to pollards_rho_brent(n) and
30 prime_factorize_big(n).
31
32 Space Complexity:
33 -  $O(f)$  auxiliary heap space for all operations, where f is the number of factors
34 returned.
35
36 */
37
38 #include <algorithm>
39 #include <cmath>
40 #include <cstdlib>
41 #include <vector>
42
43 template<class Int>
44 std::vector<Int> prime_factorize(Int n) {
45     if (n <= 3) {
46         return std::vector<Int>(1, n);
47     }
48     std::vector<Int> res;
49     for (Int i = 2; ; i++) {
50         int p = 0, q = n/i, r = n - q*i;
51         if (i > q || (i == q && r > 0)) {
52             break;
53         }
54         while (r == 0) {
55             p++;
56             n = q;
57             q = n/i;
58             r = n - q*i;
59         }
60         for (int j = 0; j < p; j++) {
61             res.push_back(i);
62         }
63     }
64 }
```

```

63     }
64     if (n > 1) {
65         res.push_back(n);
66     }
67     return res;
68 }
69
70 template<class Int>
71 std::vector<Int> get_divisors(Int n) {
72     if (n <= 1) {
73         return (n < 1) ? std::vector<Int>() : std::vector<Int>(1, 1);
74     }
75     std::vector<Int> res;
76     for (Int i = 1; i*i <= n; i++) {
77         if (n % i == 0) {
78             res.push_back(i);
79             if (i*i != n) {
80                 res.push_back(n/i);
81             }
82         }
83     }
84     std::sort(res.begin(), res.end());
85     return res;
86 }
87
88 long long fermat(long long n) {
89     if (n % 2 == 0) {
90         return 2;
91     }
92     long long x = sqrt(n), y = 0, r = x*x - y*y - n;
93     while (r != 0) {
94         if (r < 0) {
95             r += x + x + 1;
96             x++;
97         } else {
98             r -= y + y + 1;
99             y++;
100        }
101    }
102    return (x == y) ? (x + y) : (x - y);
103 }
104
105 typedef unsigned long long uint64;
106
107 uint64 mulmod(uint64 x, uint64 n, uint64 m) {
108     uint64 a = 0, b = x % m;
109     for (; n > 0; n >>= 1) {
110         if (n & 1) {
111             a = (a + b) % m;
112         }
113         b = (b << 1) % m;
114     }
115     return a % m;
116 }
117
118 uint64 powmod(uint64 x, uint64 n, uint64 m) {
119     uint64 a = 1, b = x;
120     for (; n > 0; n >>= 1) {
121         if (n & 1) {

```

```

122     a = mulmod(a, b, m);
123 }
124     b = mulmod(b, b, m);
125 }
126     return a % m;
127 }
128
129 uint64 rand64u() {
130     return ((uint64)(rand() & 0xf) << 60) |
131         ((uint64)(rand() & 0x7fff) << 45) |
132         ((uint64)(rand() & 0x7fff) << 30) |
133         ((uint64)(rand() & 0x7fff) << 15) |
134         ((uint64)(rand() & 0x7fff));
135 }
136
137 uint64 gcd(uint64 a, uint64 b) {
138     while (b != 0) {
139         uint64 t = b;
140         b = a % b;
141         a = t;
142     }
143     return a;
144 }
145
146 long long pollards_rho_brent(long long n) {
147     if (n % 2 == 0) {
148         return 2;
149     }
150     uint64 y = rand64u() % (n - 1) + 1;
151     uint64 c = rand64u() % (n - 1) + 1;
152     uint64 m = rand64u() % (n - 1) + 1;
153     uint64 g = 1, r = 1, q = 1, ys = 0, x = 0;
154     for (r = 1; g == 1; r <= 1) {
155         x = y;
156         for (int i = 0; i < r; i++) {
157             y = (mulmod(y, y, n) + c) % n;
158         }
159         for (long long k = 0; k < r && g == 1; k += m) {
160             ys = y;
161             long long lim = std::min(m, r - k);
162             for (int j = 0; j < lim; j++) {
163                 y = (mulmod(y, y, n) + c) % n;
164                 q = mulmod(q, (x > y) ? (x - y) : (y - x), n);
165             }
166             g = gcd(q, n);
167         }
168     }
169     if (g == n) {
170         do {
171             ys = (mulmod(ys, ys, n) + c) % n;
172             g = gcd((x > ys) ? (x - ys) : (ys - x), n);
173         } while (g <= 1);
174     }
175     return g;
176 }
177
178 bool is_prime(long long n) {
179     static const int np = 9, p[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
180     for (int i = 0; i < np; i++) {

```

```

181     if (n % p[i] == 0) {
182         return n == p[i];
183     }
184 }
185 if (n < p[np - 1]) {
186     return false;
187 }
188 uint64 t;
189 int s = 0;
190 for (t = n - 1; !(t & 1); t >>= 1) {
191     s++;
192 }
193 for (int i = 0; i < np; i++) {
194     uint64 r = powmod(p[i], t, n);
195     if (r == 1) {
196         continue;
197     }
198     bool ok = false;
199     for (int j = 0; j < s && !ok; j++) {
200         ok |= (r == (uint64)n - 1);
201         r = mulmod(r, r, n);
202     }
203     if (!ok) {
204         return false;
205     }
206 }
207 return true;
208 }

209 std::vector<long long> prime_factorize_big(
210     long long n, long long trial_division_cutoff = 1000000LL) {
211 if (n <= 3) {
212     return std::vector<long long>(1, n);
213 }
214 std::vector<long long> res;
215 for (; n % 2 == 0; n /= 2) {
216     res.push_back(2);
217 }
218 for (; n % 3 == 0; n /= 3) {
219     res.push_back(3);
220 }
221 for (int i = 5, w = 4; i <= trial_division_cutoff && i*i <= n; i += w) {
222     for (; n % i == 0; n /= i) {
223         res.push_back(i);
224     }
225     w = 6 - w;
226 }
227 for (long long p; n > trial_division_cutoff && !is_prime(n); n /= p) {
228     do {
229         p = pollards_rho_brent(n);
230     } while (p == n);
231     res.push_back(p);
232 }
233 if (n != 1) {
234     res.push_back(n);
235 }
236 sort(res.begin(), res.end());
237 return res;
238 }

```

```
240
241 /** Example Usage */
242
243 #include <cassert>
244 #include <set>
245 using namespace std;
246
247 void validate(long long n, const vector<long long> &factors) {
248     if (n == 1 || is_prime(n)) {
249         assert(factors == vector<long long>(1, n));
250         return;
251     }
252     long long prod = 1;
253     for (int i = 0; i < factors.size(); i++) {
254         assert(is_prime(factors[i]));
255         prod *= factors[i];
256     }
257     assert(prod == n);
258 }
259
260 int main() {
261     // Small tests.
262     for (int i = 1; i <= 10000; i++) {
263         vector<long long> v1 = prime_factorize((long long)i);
264         vector<long long> v2 = prime_factorize_big(i);
265         validate(i, v1);
266         assert(v1 == v2);
267         vector<int> d = get_divisors(i);
268         set<int> s(d.begin(), d.end());
269         assert(d.size() == s.size());
270         for (int j = 1; j <= i; j++) {
271             if (i % j == 0) {
272                 assert(s.count(j));
273             }
274         }
275     }
276 }
277 // Fermat works best for numbers with two factors close to each other.
278 long long n = 1000003LL*100000037;
279 assert(fermat(n) == 1000003);
280 }
281 // Large tests.
282 const int ntets = 7;
283 const long long tests[] = {
284     3LL*3*5*7*9949*9967*1000003,
285     2LL*1000003*1000000007,
286     999961LL*1000033,
287     357267896789127671LL,
288     2LL*2*2*2*2*2*3*3*3*5*5*7*7*11*13*17*19*23*29*31*37,
289     2LL*2*2*2*2*2*3*3*3*3*5*5*7*7*35336848213,
290     2LL*2*2*2*2*2*3*3*3*3*5*5*7*7*186917*186947,
291 };
292 for (int i = 0; i < ntets; i++) {
293     validate(tests[i], prime_factorize_big(tests[i]));
294 }
295 }
296 return 0;
297 }
```

5.3.5 Euler's Totient Function

```

1  /*
2
3 Euler's totient function phi(n) returns the number of positive integers less
4 than or equal to n that are relatively prime to n. That is, phi(n) is the number
5 of integers k in the range [1, n] for which gcd(n, k) = 1. The computation of
6 phi(1..n) can be performed simultaneously, as done so by phi_table(n) which
7 returns a vector v such that v[i] stores phi(i) for i in the range [0, n].
8
9 Time Complexity:
10 - O(n log(log(n))) per call to phi(n) and phi_table(n).
11
12 Space Complexity:
13 - O(1) auxiliary space for phi(n).
14 - O(n) auxiliary heap space for phi_table(n).
15
16 */
17
18 #include <vector>
19
20 int phi(int n) {
21     int res = n;
22     for (int i = 2; i*i <= n; i++) {
23         if (n % i == 0) {
24             while (n % i == 0) {
25                 n /= i;
26             }
27             res -= res/i;
28         }
29     }
30     if (n > 1) {
31         res -= res/n;
32     }
33     return res;
34 }
35
36 std::vector<int> phi_table(int n) {
37     std::vector<int> res(n + 1);
38     for (int i = 0; i <= n; i++) {
39         res[i] = i;
40     }
41     for (int i = 1; i <= n; i++) {
42         for (int j = 2*i; j <= n; j += i) {
43             res[j] -= res[i];
44         }
45     }
46     return res;
47 }
48
49 /** Example Usage ***/
50
51 #include <cassert>
52 using namespace std;
53
54 int main() {
55     assert(phi(1) == 1);

```

```

56     assert(phi(9) == 6);
57     assert(phi(1234567) == 1224720);
58     const int n = 1000;
59     vector<int> v = phi_table(n);
60     for (int i = 0; i <= n; i++) {
61         assert(v[i] == phi(i));
62     }
63     return 0;
64 }
```

5.3.6 Binary Exponentiation

```

1  /*
2
3 Given three unsigned 64-bit integers x, n, and m, powmod() returns x raised to
4 the power of n (modulo m). mulmod() returns x multiplied by n (modulo m).
5 Despite the fact that both functions use unsigned 64-bit integers for arguments
6 and intermediate calculations, arguments x and n must not exceed  $2^{63} - 1$  (the
7 maximum value of a signed 64-bit integer) for the result to be correctly
8 computed without overflow.
9
10 Binary exponentiation, also known as exponentiation by squaring, decomposes the
11 exponentiation into a logarithmic number of multiplications while avoiding
12 overflow. To further prevent overflow in the intermediate squaring computations,
13 multiplication is performed using a similar principle of repeated addition.
14
15 Time Complexity:
16 -  $O(\log n)$  per call to mulmod() and powmod(), where n is the second argument.
17
18 Space Complexity:
19 -  $O(1)$  auxiliary.
20
21 */
22
23 typedef unsigned long long uint64;
24
25 uint64 mulmod(uint64 x, uint64 n, uint64 m) {
26     uint64 a = 0, b = x % m;
27     for (; n > 0; n >>= 1) {
28         if (n & 1) {
29             a = (a + b) % m;
30         }
31         b = (b << 1) % m;
32     }
33     return a % m;
34 }
35
36 uint64 powmod(uint64 x, uint64 n, uint64 m) {
37     uint64 a = 1, b = x;
38     for (; n > 0; n >>= 1) {
39         if (n & 1) {
40             a = mulmod(a, b, m);
41         }
42         b = mulmod(b, b, m);
43     }

```

```

44     return a % m;
45 }
46
47 /** Example Usage **/
48
49 #include <cassert>
50
51 int main() {
52     assert(powmod(2, 10, 1000000007) == 1024);
53     assert(powmod(2, 62, 1000000) == 387904);
54     assert(powmod(10001, 10001, 100000) == 10001);
55     return 0;
56 }
```

5.4 Arbitrary Precision Arithmetic

5.4.1 Big Integers (Simple)

```

1 /*
2
3 Perform simple arithmetic operations on arbitrary precision big integers whose
4 digits are internally represented as an std::string in little-endian order.
5
6 - bigint(n) constructs a big integer from a long long (default = 0).
7 - bigint(s) constructs a big integer from a string s, which must strictly
8   consist of a sequence of numeric digits, optionally preceded by a minus sign.
9 - str() returns the string representation of the big integer.
10 - comp(a, b) returns -1, 0, or 1 depending on whether the big integers a and b
11   compare less, equal, or greater, respectively.
12 - add(a, b) returns the sum of big integers a and b.
13 - sub(a, b) returns the difference of big integers a and b.
14 - mul(a, b) returns the product of big integers a and b.
15 - div(a, b) returns the quotient of big integers a and b.
16
17 Time Complexity:
18 - O(n) per call to the constructor, str(), comp(), add(), and sub(), where n is
19   total number of digits in the argument(s) and result for each operation.
20 - O(n*m) per call to mul(a, b) and div(a, b) where n is the number of digits in
21   a and m is the number of digits in b.
22
23 Space Complexity:
24 - O(n) for storage of the big integer, where n is the number of the digits.
25 - O(n) auxiliary heap space for str(), add(), sub(), mul(), and div(), where n
26   the total number of digits in the argument(s) and result for each operation.
27
28 */
29
30 #include <algorithm>
31 #include <cctype>
32 #include <stdexcept>
33 #include <string>
34
35 class bigint {
```

```
36     std::string digits;
37     int sign;
38
39     void normalize() {
40         size_t pos = digits.find_last_not_of('0');
41         if (pos != std::string::npos) {
42             digits.erase(pos + 1);
43         }
44         if (digits.empty()) {
45             digits = "0";
46         }
47         if (digits.size() == 1 && digits[0] == '0') {
48             sign = 1;
49             return;
50         }
51     }
52
53     static int comp(const std::string &a, const std::string &b,
54                     int asign, int bsign) {
55         if (asign != bsign) {
56             return asign < bsign ? -1 : 1;
57         }
58         if (a.size() != b.size()) {
59             return a.size() < b.size() ? -asign : asign;
60         }
61         for (int i = (int)a.size() - 1; i >= 0; i--) {
62             if (a[i] != b[i]) {
63                 return a[i] < b[i] ? -asign : asign;
64             }
65         }
66         return 0;
67     }
68
69     static bigint add(const std::string &a, const std::string &b,
70                       int asign, int bsign) {
71         if (asign != bsign) {
72             return (asign == 1) ? sub(a, b, asign, 1) : sub(b, a, bsign, 1);
73         }
74         bigint res;
75         res.sign = asign;
76         res.digits.resize(std::max(a.size(), b.size()) + 1, '0');
77         for (int i = 0, carry = 0; i < (int)res.digits.size(); i++) {
78             int d = carry;
79             if (i < (int)a.size()) {
80                 d += a[i] - '0';
81             }
82             if (i < (int)b.size()) {
83                 d += b[i] - '0';
84             }
85             res.digits[i] = '0' + (d % 10);
86             carry = d/10;
87         }
88         res.normalize();
89         return res;
90     }
91
92     static bigint sub(const std::string &a, const std::string &b,
93                      int asign, int bsign) {
94         if (asign == -1 || bsign == -1) {
```

```

95     return add(a, b, asign, -bsign);
96 }
97 bigint res;
98 if (comp(a, b, asign, bsign) < 0) {
99     res = sub(b, a, bsign, asign);
100    res.sign = -1;
101    return res;
102 }
103 res.digits.assign(a.size(), '0');
104 for (int i = 0, borrow = 0; i < (int)res.digits.size(); i++) {
105     int d = (i < (int)b.size() ? a[i] - b[i] : a[i] - '0') - borrow;
106     if (a[i] > '0') {
107         borrow = 0;
108     }
109     if (d < 0) {
110         d += 10;
111         borrow = 1;
112     }
113     res.digits[i] = '0' + (d % 10);
114 }
115 res.normalize();
116 return res;
117 }

118 public:
119 bigint(long long n = 0) {
120     sign = (n < 0) ? -1 : 1;
121     if (n == 0) {
122         digits = "0";
123         return;
124     }
125     for (n = (n > 0) ? n : -n; n > 0; n /= 10) {
126         digits += '0' + (n % 10);
127     }
128     normalize();
129 }
130 }

131 bigint(const std::string &s) {
132     if (s.empty() || (s[0] == '-' && s.size() == 1)) {
133         throw std::runtime_error("Invalid string format to construct bigint.");
134     }
135     digits.assign(s.rbegin(), s.rend());
136     if (s[0] == '-') {
137         sign = -1;
138         digits.erase(digits.size() - 1);
139     } else {
140         sign = 1;
141     }
142     if (digits.find_first_not_of("0123456789") != std::string::npos) {
143         throw std::runtime_error("Invalid string format to construct bigint.");
144     }
145     normalize();
146 }
147

148 std::string to_string() const {
149     return (sign < 0 ? "-" : "") + std::string(digits.rbegin(), digits.rend());
150 }
151

152 friend int comp(const bigint &a, const bigint &b) {

```

```
154     return comp(a.digits, b.digits, a.sign, b.sign);
155 }
156
157 friend bigint add(const bigint &a, const bigint &b) {
158     return add(a.digits, b.digits, a.sign, b.sign);
159 }
160
161 friend bigint sub(const bigint &a, const bigint &b) {
162     return sub(a.digits, b.digits, a.sign, b.sign);
163 }
164
165 friend bigint mul(const bigint &a, const bigint &b) {
166     bigint res, row(a);
167     for (int i = 0; i < (int)b.digits.size(); i++) {
168         for (int j = 0; j < (b.digits[i] - '0'); j++) {
169             res = add(res.digits, row.digits, res.sign, row.sign);
170         }
171         if (row.digits.size() > 1 || row.digits[0] != '0') {
172             row.digits.insert(0, "0");
173         }
174     }
175     res.sign = a.sign*b.sign;
176     res.normalize();
177     return res;
178 }
179
180 friend bigint div(const bigint &a, const bigint &b) {
181     bigint res, row;
182     res.digits.assign(a.digits.size(), '0');
183     for (int i = (int)a.digits.size() - 1; i >= 0; i--) {
184         row.digits.insert(row.digits.begin(), a.digits[i]);
185         while (comp(row.digits, b.digits, row.sign, 1) > 0) {
186             res.digits[i]++;
187             row = sub(row.digits, b.digits, row.sign, 1);
188         }
189     }
190     res.sign = a.sign*b.sign;
191     res.normalize();
192     return res;
193 }
194 };
195
196 /**
197 #include <cassert>
198
199 int main() {
200     bigint a("-9899819294989142124"), b("12398124981294214");
201     assert(add(a, b).to_string() == "-9887421170007847910");
202     assert(sub(a, b).to_string() == "-9912217419970436338");
203     assert(mul(a, b).to_string() == "-122739196911503356525379735104870536");
204     assert(div(a, b).to_string() == "-798");
205     assert(comp(a, b) == -1 && comp(a, a) == 0 && comp(b, a) == 1);
206     return 0;
207 }
208 }
```

5.4.2 Big Integers

```

1  /*
2
3 Perform operations on arbitrary precision big integers internally represented as
4 a vector of base-1000000000 digits in little-endian order. Typical arithmetic
5 operations involving mixed numeric primitives and strings are supported using
6 templates and operator overloading, as long as at least one operand is a bigint
7 at any given level of evaluation.
8
9 - bigint(n) constructs a big integer from a long long (default = 0).
10 - bigint(s) constructs a big integer from a C string or an std::string s.
11 - operator = is defined to copy from another big integer or to assign from an
12   64-bit integer primitive.
13 - size() returns the number of digits in the base-10 representation.
14 - operators >> and << are defined to support stream-based input and output.
15 - v.to_string(), v.to_llong(), v.to_double(), and v.to_ldouble() return the big
16   integer v converted to an std::string, long long, double, and long double
17   respectively. For the latter three data types, overflow behavior is based on
18   that of inputting from std::istream.
19 - v.abs() returns the absolute value of big integer v.
20 - a.comp(b) returns -1, 0, or 1 depending on whether the big integers a and b
21   compare less, equal, or greater, respectively.
22 - operators <, >, <=, >=, ==, !=, +, -, *, /, %, ++, --, +=, -=, *=, /=, and %=
23   are defined analogous to those on integer primitives. Addition, subtraction,
24   and comparisons are performed using the standard linear algorithms.
25 Multiplication is performed using a combination of the grade school algorithm
26 (for smaller inputs) and either the Karatsuba algorithm (if the USE_FFT_MULT
27 flag is set to false) or the Schonhage-Strassen algorithm (if USE_FFT_MULT is
28 set to true). Division and modulo are computed simultaneously using the grade
29 school method.
30 - a.div(b) returns a pair consisting of the quotient and remainder.
31 - v.pow(n) returns v raised to the power of n.
32 - v.sqrt() returns the integral part of the square root of big integer v.
33 - v.nth_root(n) returns the integral part of the n-th root of big integer v.
34 - rand(n) returns a random, positive big integer with n digits.
35
36 Time Complexity:
37 - O(n) per call to the constructors, size(), to_string(), to_llong(),
38   to_double(), to_ldouble(), abs(), comp(), rand(), and all comparison and
39   arithmetic operators except multiplication, division, and modulo, where n is
40   total number of digits in the argument(s) and result for each operation.
41 - O(n*log(n)*log(log(n))) or O(n^1.585) per call to multiplication operations,
42   depending on whether USE_FFT_MULT is set to true or false.
43 - O(n*m) per call to division and modulo operations, where n and m are the
44   number of digits in the dividend and divisor, respectively.
45 - O(M(m) log n) per call to pow(n), where m is the length of the big integer.
46
47 Space Complexity:
48 - O(n) for storage of the big integer.
49 - O(n) auxiliary heap space for negation, addition, subtraction, multiplication,
50   division, abs(), sqrt(), pow(), and nth_root().
51 - O(1) auxiliary space for all other operations.
52
53 */
54 #include <algorithm>

```

```

56 #include <cmath>
57 #include <complex>
58 #include <cstdlib>
59 #include <cstring>
60 #include <iomanip>
61 #include <iostream>
62 #include <ostream>
63 #include <sstream>
64 #include <stdexcept>
65 #include <string>
66 #include <utility>
67 #include <vector>
68
69 class bigint {
70     static const int BASE = 1000000000, BASE_DIGITS = 9;
71     static const bool USE_FFT_MULT = true;
72
73     typedef std::vector<int> vint;
74     typedef std::vector<long long> vll;
75     typedef std::vector<std::complex<double>> vcd;
76
77     vint digits;
78     int sign;
79
80     void normalize() {
81         while (!digits.empty() && digits.back() == 0) {
82             digits.pop_back();
83         }
84         if (digits.empty()) {
85             sign = 1;
86         }
87     }
88
89     void read(int n, const char *s) {
90         sign = 1;
91         digits.clear();
92         int pos = 0;
93         while (pos < n && (s[pos] == '-' || s[pos] == '+')) {
94             if (s[pos] == '-') {
95                 sign = -sign;
96             }
97             pos++;
98         }
99         for (int i = n - 1; i >= pos; i -= BASE_DIGITS) {
100             int x = 0;
101             for (int j = std::max(pos, i - BASE_DIGITS + 1); j <= i; j++) {
102                 x = x*10 + s[j] - '0';
103             }
104             digits.push_back(x);
105         }
106         normalize();
107     }
108
109     static int comp(const vint &a, const vint &b, int asign, int bsign) {
110         if (asign != bsign) {
111             return asign < bsign ? -1 : 1;
112         }
113         if (a.size() != b.size()) {
114             return a.size() < b.size() ? -asign : asign;

```

```

115     }
116     for (int i = (int)a.size() - 1; i >= 0; i--) {
117         if (a[i] != b[i]) {
118             return a[i] < b[i] ? -asign : asign;
119         }
120     }
121     return 0;
122 }
123
124 static bigint add(const vint &a, const vint &b, int asign, int bsign) {
125     if (asign != bsign) {
126         return (asign == 1) ? sub(a, b, asign, 1) : sub(b, a, bsign, 1);
127     }
128     bigint res;
129     res.digits = a;
130     res.sign = asign;
131     int carry = 0, size = (int)std::max(a.size(), b.size());
132     for (int i = 0; i < size || carry; i++) {
133         if (i == (int)res.digits.size()) {
134             res.digits.push_back(0);
135         }
136         res.digits[i] += carry + (i < (int)b.size() ? b[i] : 0);
137         carry = (res.digits[i] >= BASE) ? 1 : 0;
138         if (carry) {
139             res.digits[i] -= BASE;
140         }
141     }
142     return res;
143 }
144
145 static bigint sub(const vint &a, const vint &b, int asign, int bsign) {
146     if (asign == -1 || bsign == -1) {
147         return add(a, b, asign, -bsign);
148     }
149     bigint res;
150     if (comp(a, b, asign, bsign) < 0) {
151         res = sub(b, a, bsign, asign);
152         res.sign = -1;
153         return res;
154     }
155     res.digits = a;
156     res.sign = asign;
157     for (int i = 0, borrow = 0; i < (int)a.size() || borrow; i++) {
158         res.digits[i] -= borrow + (i < (int)b.size() ? b[i] : 0);
159         borrow = res.digits[i] < 0;
160         if (borrow) {
161             res.digits[i] += BASE;
162         }
163     }
164     res.normalize();
165     return res;
166 }
167
168 static vint convert_base(const vint &digits, int l1, int l2) {
169     vll p(std::max(l1, l2) + 1);
170     p[0] = 1;
171     for (int i = 1; i < (int)p.size(); i++) {
172         p[i] = p[i - 1]*10;
173     }

```

```

174     vint res;
175     long long curr = 0;
176     for (int i = 0, curr_digits = 0; i < (int)digits.size(); i++) {
177         curr += digits[i]*p[curr_digits];
178         curr_digits += 11;
179         while (curr_digits >= 12) {
180             res.push_back((int)(curr % p[12]));
181             curr /= p[12];
182             curr_digits -= 12;
183         }
184     }
185     res.push_back((int)curr);
186     while (!res.empty() && res.back() == 0) {
187         res.pop_back();
188     }
189     return res;
190 }
191
192 template<class It>
193 static vll karatsuba(It alo, It ahi, It blo, It bhi) {
194     int n = std::distance(alo, ahi), k = n/2;
195     vll res(n*2);
196     if (n <= 32) {
197         for (int i = 0; i < n; i++) {
198             for (int j = 0; j < n; j++) {
199                 res[i + j] += alo[i]*blo[j];
200             }
201         }
202         return res;
203     }
204     vll a1b1 = karatsuba(alo, alo + k, blo, blo + k);
205     vll a2b2 = karatsuba(allo + k, ahi, blo + k, bhi);
206     vll a2(allo + k, ahi), b2(blo + k, bhi);
207     for (int i = 0; i < k; i++) {
208         a2[i] += alo[i];
209         b2[i] += blo[i];
210     }
211     vll r = karatsuba(a2.begin(), a2.end(), b2.begin(), b2.end());
212     for (int i = 0; i < (int)a1b1.size(); i++) {
213         r[i] -= a1b1[i];
214         res[i] += a1b1[i];
215     }
216     for (int i = 0; i < (int)a2b2.size(); i++) {
217         r[i] -= a2b2[i];
218         res[i + n] += a2b2[i];
219     }
220     for (int i = 0; i < (int)r.size(); i++) {
221         res[i + k] += r[i];
222     }
223     return res;
224 }
225
226 template<class It>
227 static vcd fft(It lo, It hi, bool invert = false) {
228     int n = std::distance(lo, hi), k = 0, high1 = -1;
229     while ((1 << k) < n) {
230         k++;
231     }
232     std::vector<int> rev(n, 0);

```

```

233     for (int i = 1; i < n; i++) {
234         if (!(i & (i - 1))) {
235             high1++;
236         }
237         rev[i] = rev[i ^ (1 << high1)];
238         rev[i] |= (1 << (k - high1 - 1));
239     }
240     vcd roots(n), res(n);
241     for (int i = 0; i < n; i++) {
242         double alpha = 2*3.14159265358979323846*i/n;
243         roots[i] = std::complex<double>(cos(alpha), sin(alpha));
244         res[i] = *(lo + rev[i]);
245     }
246     for (int len = 1; len < n; len <= 1) {
247         vcd tmp(n);
248         int rstep = roots.size()/(len << 1);
249         for (int pdest = 0; pdest < n; pdest += len) {
250             int p = pdest;
251             for (int i = 0; i < len; i++) {
252                 std::complex<double> c = roots[i*rstep]*res[p + len];
253                 tmp[pdest] = res[p] + c;
254                 tmp[pdest + len] = res[p] - c;
255                 pdest++;
256                 p++;
257             }
258         }
259         res.swap(tmp);
260     }
261     if (invert) {
262         for (int i = 0; i < (int)res.size(); i++) {
263             res[i] /= n;
264         }
265         std::reverse(res.begin() + 1, res.end());
266     }
267     return res;
268 }
269
270 public:
271     bigint() : sign(1) {}
272     bigint(int v) { *this = (long long)v; }
273     bigint(long long v) { *this = v; }
274     bigint(const char *s) { read(strlen(s), s); }
275     bigint(const std::string &s) { read(s.size(), s.c_str()); }
276
277     void operator=(const bigint &v) {
278         sign = v.sign;
279         digits = v.digits;
280     }
281
282     void operator=(long long v) {
283         sign = 1;
284         if (v < 0) {
285             sign = -1;
286             v = -v;
287         }
288         digits.clear();
289         for (; v > 0; v /= BASE) {
290             digits.push_back(v % BASE);
291         }

```

```
292     }
293
294     int size() const {
295         if (digits.empty()) {
296             return 1;
297         }
298         std::ostringstream oss;
299         oss << digits.back();
300         return oss.str().length() + BASE_DIGITS*(digits.size() - 1);
301     }
302
303     friend std::istream& operator>>(std::istream &in, bigint &v) {
304         std::string s;
305         in >> s;
306         v.read(s.size(), s.c_str());
307         return in;
308     }
309
310     friend std::ostream& operator<<(std::ostream &out, const bigint &v) {
311         if (v.sign == -1) {
312             out << '-';
313         }
314         out << (v.digits.empty() ? 0 : v.digits.back());
315         for (int i = (int)v.digits.size() - 2; i >= 0; i--) {
316             out << std::setw(BASE_DIGITS) << std::setfill('0') << v.digits[i];
317         }
318         return out;
319     }
320
321     std::string to_string() const {
322         std::ostringstream oss;
323         if (sign == -1) {
324             oss << '-';
325         }
326         oss << (digits.empty() ? 0 : digits.back());
327         for (int i = (int)digits.size() - 2; i >= 0; i--) {
328             oss << std::setw(BASE_DIGITS) << std::setfill('0') << digits[i];
329         }
330         return oss.str();
331     }
332
333     long long to_llong() const {
334         long long res = 0;
335         for (int i = (int)digits.size() - 1; i >= 0; i--) {
336             res = res*BASE + digits[i];
337         }
338         return res*sign;
339     }
340
341     double to_double() const {
342         std::stringstream ss(to_string());
343         double res;
344         ss >> res;
345         return res;
346     }
347
348     long double to_ldouble() const {
349         std::stringstream ss(to_string());
350         long double res;
```

```

351     ss >> res;
352     return res;
353 }
354
355 int comp(const bigint &v) const {
356     return comp(digits, v.digits, sign, v.sign);
357 }
358
359 bool operator<(const bigint &v) const { return comp(v) < 0; }
360 bool operator>(const bigint &v) const { return comp(v) > 0; }
361 bool operator<=(const bigint &v) const { return comp(v) <= 0; }
362 bool operator>=(const bigint &v) const { return comp(v) >= 0; }
363 bool operator==(const bigint &v) const { return comp(v) == 0; }
364 bool operator!=(const bigint &v) const { return comp(v) != 0; }
365
366 template<class T>
367 friend bool operator<(const T &a, const bigint &b) { return bigint(a) < b; }
368
369 template<class T>
370 friend bool operator>(const T &a, const bigint &b) { return bigint(a) > b; }
371
372 template<class T>
373 friend bool operator<=(const T &a, const bigint &b) { return bigint(a) <= b; }
374
375 template<class T>
376 friend bool operator>=(const T &a, const bigint &b) { return bigint(a) >= b; }
377
378 template<class T>
379 friend bool operator==(const T &a, const bigint &b) { return bigint(a) == b; }
380
381 template<class T>
382 friend bool operator!=(const T &a, const bigint &b) { return bigint(a) != b; }
383
384 bigint abs() const {
385     bigint res(*this);
386     res.sign = 1;
387     return res;
388 }
389
390 bigint operator-() const {
391     bigint res(*this);
392     res.sign = -sign;
393     return res;
394 }
395
396 bigint operator+(const bigint &v) const {
397     return add(digits, v.digits, sign, v.sign);
398 }
399
400 bigint operator-(const bigint &v) const {
401     return sub(digits, v.digits, sign, v.sign);
402 }
403
404 void operator*=(int v) {
405     if (v < 0) {
406         sign = -sign;
407         v = -v;
408     }
409     for (int i = 0, carry = 0; i < (int)digits.size() || carry; i++) {

```

```

410     if (i == (int)digits.size()) {
411         digits.push_back(0);
412     }
413     long long curr = digits[i]*(long long)v + carry;
414     carry = (int)(curr/BASE);
415     digits[i] = (int)(curr % BASE);
416 }
417 normalize();
418 }
419
420 bigint operator*(int v) const {
421     bigint res(*this);
422     res *= v;
423     return res;
424 }
425
426 bigint operator*(const bigint &v) const {
427     static const int TEMP_BASE = 10000, TEMP_BASE_DIGITS = 4;
428     vint a = convert_base(digits, BASE_DIGITS, TEMP_BASE_DIGITS);
429     vint b = convert_base(v.digits, BASE_DIGITS, TEMP_BASE_DIGITS);
430     int n = 1 << (33 - __builtin_clz(std::max(a.size(), b.size()) - 1));
431     a.resize(n, 0);
432     b.resize(n, 0);
433     vll c;
434     if (USE_FFT_MULT) {
435         vcd at = fft(a.begin(), a.end()), bt = fft(b.begin(), b.end());
436         for (int i = 0; i < n; i++) {
437             at[i] *= bt[i];
438         }
439         at = fft(at.begin(), at.end(), true);
440         c.resize(n);
441         for (int i = 0; i < n; i++) {
442             c[i] = at[i].real() + 0.5;
443         }
444     } else {
445         c = karatsuba(a.begin(), a.end(), b.begin(), b.end());
446     }
447     bigint res;
448     res.sign = sign*v.sign;
449     for (int i = 0, carry = 0; i < (int)c.size(); i++) {
450         long long d = c[i] + carry;
451         res.digits.push_back(d % TEMP_BASE);
452         carry = d/TEMP_BASE;
453     }
454     res.digits = convert_base(res.digits, TEMP_BASE_DIGITS, BASE_DIGITS);
455     res.normalize();
456     return res;
457 }
458
459 bigint& operator/=(int v) {
460     if (v == 0) {
461         throw std::runtime_error("Division by zero in bigint.");
462     }
463     if (v < 0) {
464         sign = -sign;
465         v = -v;
466     }
467     for (int i = (int)digits.size() - 1, rem = 0; i >= 0; i--) {
468         long long curr = digits[i] + rem*(long long)BASE;

```

```

469     digits[i] = (int)(curr/v);
470     rem = (int)(curr % v);
471 }
472 normalize();
473 return *this;
474 }
475
476 bigint operator/(int v) const {
477     bigint res(*this);
478     res /= v;
479     return res;
480 }
481
482 int operator%(int v) const {
483     if (v == 0) {
484         throw std::runtime_error("Division by zero in bigint.");
485     }
486     if (v < 0) {
487         v = -v;
488     }
489     int m = 0;
490     for (int i = (int)digits.size() - 1; i >= 0; i--) {
491         m = (digits[i] + m*(long long)BASE) % v;
492     }
493     return m*sign;
494 }
495
496 std::pair<bigint, bigint> div(const bigint &v) const {
497     if (v == 0) {
498         throw std::runtime_error("Division by zero in bigint.");
499     }
500     if (comp(digits, v.digits, 1, 1) < 0) {
501         return std::make_pair(0, *this);
502     }
503     int norm = BASE/(v.digits.back() + 1);
504     bigint an = abs()*norm, bn = v.abs()*norm, q, r;
505     q.digits.resize(an.digits.size());
506     for (int i = (int)an.digits.size() - 1; i >= 0; i--) {
507         r *= BASE;
508         r += an.digits[i];
509         int s1 = (r.digits.size() <= bn.digits.size())
510                 ? 0 : r.digits[bn.digits.size()];
511         int s2 = (r.digits.size() <= bn.digits.size() - 1)
512                 ? 0 : r.digits[bn.digits.size() - 1];
513         int d = ((long long)s1*BASE + s2)/bn.digits.back();
514         for (r -= bn*d; r < 0; r += bn) {
515             d--;
516         }
517         q.digits[i] = d;
518     }
519     q.sign = sign*v.sign;
520     r.sign = sign;
521     q.normalize();
522     r.normalize();
523     return std::make_pair(q, r/norm);
524 }
525
526 bigint operator/(const bigint &v) const { return div(v).first; }
527 bigint operator%<(const bigint &v) const { return div(v).second; }

```

```

528     bigint operator++(int) { bigint t(*this); operator++(); return t; }
529     bigint operator--(int) { bigint t(*this); operator--(); return t; }
530     bigint& operator++() { *this = *this + bigint(1); return *this; }
531     bigint& operator--() { *this = *this - bigint(1); return *this; }
532     bigint& operator+=(const bigint &v) { *this = *this + v; return *this; }
533     bigint& operator-=(const bigint &v) { *this = *this - v; return *this; }
534     bigint& operator*=(const bigint &v) { *this = *this * v; return *this; }
535     bigint& operator/=(const bigint &v) { *this = *this / v; return *this; }
536     bigint& operator%=(const bigint &v) { *this = *this % v; return *this; }
537
538     template<class T>
539     friend bigint operator+(const T &a, const bigint &b) { return bigint(a) + b; }
540
541     template<class T>
542     friend bigint operator-(const T &a, const bigint &b) { return bigint(a) - b; }
543
544     bigint pow(int n) const {
545         if (n == 0) {
546             return bigint(1);
547         }
548         if (*this == 0 || n < 0) {
549             return bigint(0);
550         }
551         bigint x(*this), res(1);
552         for (; n != 0; n >>= 1) {
553             if (n & 1) {
554                 res *= x;
555             }
556             x *= x;
557         }
558         return res;
559     }
560
561     bigint sqrt() const {
562         if (sign == -1) {
563             throw std::runtime_error("Cannot take square root of a negative number.");
564         }
565         bigint v(*this);
566         while (v.digits.empty() || v.digits.size() % 2 == 1) {
567             v.digits.push_back(0);
568         }
569         int n = v.digits.size();
570         int ldig = (int)::sqrt((double)v.digits[n - 1]*BASE + v.digits[n - 2]);
571         int norm = BASE/(ldig + 1);
572         v *= norm;
573         v *= norm;
574         while (v.digits.empty() || v.digits.size() % 2 == 1) {
575             v.digits.push_back(0);
576         }
577         bigint r((long long)v.digits[n - 1]*BASE + v.digits[n - 2]);
578         int q = ldig = (int)::sqrt((double)v.digits[n - 1]*BASE + v.digits[n - 2]);
579         bigint res;
580         for (int j = n/2 - 1; j >= 0; j--) {
581             for (;; q--) {
582                 bigint r1 = (r - (res*2*BASE + q)*q)*BASE*BASE +
583                     (j > 0 ? (long long)v.digits[2*j - 1]*BASE + v.digits[2*j - 2] : 0);
584                 if (r1 >= 0) {
585                     r = r1;
586                     break;
587                 }
588             }
589             res *= norm;
590         }
591         return res;
592     }

```

```

587         }
588     }
589     res = res*BASE + q;
590     if (j > 0) {
591         int sz1 = res.digits.size(), sz2 = r.digits.size();
592         int d1 = (sz1 + 2 < sz2) ? r.digits[sz1 + 2] : 0;
593         int d2 = (sz1 + 1 < sz2) ? r.digits[sz1 + 1] : 0;
594         int d3 = (sz1 < sz2) ? r.digits[sz1] : 0;
595         q = ((long long)d1*BASE*BASE + (long long)d2*BASE + d3)/(ldig*2);
596     }
597 }
598 res.normalize();
599 return res/norm;
600 }
601
602 bigint nth_root(int n) const {
603     if (sign == -1 && n % 2 == 0) {
604         throw std::runtime_error("Cannot take even root of a negative number.");
605     }
606     if (*this == 0 || n < 0) {
607         return bigint(0);
608     }
609     if (n >= size()) {
610         int p = 1;
611         while (comp(bigint(p).pow(n)) > 0) {
612             p++;
613         }
614         return comp(bigint(p).pow(n)) < 0 ? p - 1 : p;
615     }
616     bigint lo(bigint(10).pow((int)ceil((double)size()/n) - 1)), hi(lo*10), mid;
617     while (lo < hi) {
618         mid = (lo + hi)/2;
619         int cmp = comp(digits, mid.pow(n).digits, 1, 1);
620         if (lo < mid && cmp > 0) {
621             lo = mid;
622         } else if (mid < hi && cmp < 0) {
623             hi = mid;
624         } else {
625             return (sign == -1) ? -mid : mid;
626         }
627     }
628     return (sign == -1) ? -(mid + 1) : (mid + 1);
629 }
630
631 static bigint rand(int n) {
632     if (n == 0) {
633         return bigint(0);
634     }
635     std::string s(1, '1' + (rand() % 9));
636     for (int i = 1; i < n; i++) {
637         s += '0' + (rand() % 10);
638     }
639     return bigint(s);
640 }
641
642 friend int comp(const bigint &a, const bigint &b) { return a.comp(b); }
643 friend bigint abs(const bigint &v) { return v.abs(); }
644 friend bigint pow(const bigint &v, int n) { return v.pow(n); }
645 friend bigint sqrt(const bigint &v) { return v.sqrt(); }

```

```

646     friend bigint nth_root(const bigint &v, int n) { return v.nth_root(n); }
647 };
648
649 /** Example Usage ***/
650
651 #include <cassert>
652
653 int main() {
654     bigint a("-9899819294989142124"), b("12398124981294214");
655     assert(a + b == "-9887421170007847910");
656     assert(a - b == "-9912217419970436338");
657     assert(a * b == "-122739196911503356525379735104870536");
658     assert(a / b == "-798");
659     assert(bigint(20).pow(12345).size() == 16062);
660     assert(bigint("9812985918924981892491829").nth_root(4) == 1769906);
661     for (int i = -100; i <= 100; i++) {
662         if (i >= 0) {
663             assert(bigint(i).sqrt() == (int)sqrt(i));
664         }
665         for (int j = -100; j <= 100; j++) {
666             assert(bigint(i) + bigint(j) == i + j);
667             assert(bigint(i) - bigint(j) == i - j);
668             assert(bigint(i) * bigint(j) == i * j);
669             if (j != 0) {
670                 assert(bigint(i) / bigint(j) == i / j);
671             }
672             if (0 < i && i <= 10 && 0 < j && j <= 10) {
673                 assert(bigint(i).nth_root(j) == (long long)(pow(i, 1.0/j) + 1E-5));
674                 long long p = 1;
675                 for (int k = 0; k < j; k++) {
676                     p *= i;
677                 }
678                 assert(bigint(i).pow(j) == p);
679             }
680         }
681     }
682     for (int i = 0; i < 20; i++) {
683         int n = rand() % 100 + 1;
684         bigint a(bigint::rand(n)), s(a.sqrt()), xx(s*s), yy(s + 1);
685         yy *= yy;
686         assert(xx <= a && a < yy);
687         bigint b(bigint::rand(rand() % n + 1) + 1), q(a/b);
688         xx = q*b;
689         yy = b*(q + 1);
690         assert(a >= xx && a < yy);
691     }
692     bigint x(-6);
693     assert(x.to_string() == "-6");
694     assert(x.to_llong() == -6LL);
695     assert(x.to_double() == -6.0);
696     assert(x.to_ldouble() == -6.0);
697     return 0;
698 }
```

5.4.3 Rational Numbers

```

1  /*
2
3 Perform operations on rational numbers internally represented as two integers, a
4 numerator and a denominator. The template integer type must support streamed
5 input/output, comparisons, and arithmetic operations. Overflow is not checked
6 for in internal operations.
7
8 - rational(n) constructs a rational with numerator n and denominator 1.
9 - rational(n, d) constructs a rational with numerator n and denominator d.
10 - operator >> inputs a rational using the next integer from the stream as the
11   numerator and 1 as the denominator.
12 - operator << outputs a rational as a string consisting of possibly a minus sign
13   followed by the numerator, followed by a slash, followed by the denominator.
14 - v.to_string(), v.to_llong(), v.to_double(), and v.to_ldouble() return the big
15   integer v converted to an std::string, long long, double, and long double
16   respectively.
17 - operators <, >, <=, >=, ==, !=, +, -, *, /, %, ++, --, +=, -=, *=, /=, and %=
18   are defined analogous to those on numerical primitives.
19
20 Time Complexity:
21 - O(log(n + d)) per call to constructor rational(n, d).
22 - O(1) per call to all other operations, assuming that corresponding operations
23   on the template integer type are O(1) as well.
24
25 Space Complexity:
26 - O(1) for storage of the rational.
27 - O(1) auxiliary space for all operations.
28
29 */
30
31 #include <iostream>
32 #include <ostream>
33 #include <sstream>
34 #include <string>
35
36 template<class Int = long long>
37 class rational {
38     Int num, den;
39
40 public:
41     rational(): num(0), den(1) {}
42     rational(const Int &n) : num(n), den(1) {}
43
44     template<class T1, class T2>
45     rational(const T1 &n, const T2 &d): num(n), den(d) {
46         if (den == 0) {
47             throw std::runtime_error("Division by zero in rational.");
48         }
49         if (den < 0) {
50             num = -num;
51             den = -den;
52         }
53         Int a(num < 0 ? -num : num), b(den), tmp;
54         while (a != 0 && b != 0) {
55             tmp = a % b;
56             a = b;
57             b = tmp;
58         }
59         Int gcd = (b == 0) ? a : b;

```

```
60     num /= gcd;
61     den /= gcd;
62 }
63
64 friend std::istream& operator>>(std::istream &in, rational &r) {
65     std::string s;
66     in >> r.num;
67     r.den = 1;
68     return in;
69 }
70
71 friend std::ostream& operator<<(std::ostream &out, const rational &r) {
72     out << r.num << "/" << r.den;
73     return out;
74 }
75
76 std::string to_string() const {
77     std::stringstream ss;
78     ss << num << " " << den;
79     std::string n, d;
80     ss >> n >> d;
81     return n + "/" + d;
82 }
83
84 long long to_llong() const {
85     std::stringstream ss;
86     ss << num << " " << den;
87     long long n, d;
88     ss >> n >> d;
89     return n/d;
90 }
91
92 double to_double() const {
93     std::stringstream ss;
94     ss << num << " " << den;
95     double n, d;
96     ss >> n >> d;
97     return n/d;
98 }
99
100 long double to_ldouble() const {
101     long double n, d;
102     std::stringstream ss;
103     ss << num << " " << den;
104     ss >> n >> d;
105     return n/d;
106 }
107
108 bool operator<(const rational &r) const {
109     return num*r.den < r.num*den;
110 }
111
112 bool operator>(const rational &r) const {
113     return r.num*den < num*r.den;
114 }
115
116 bool operator<=(const rational &r) const {
117     return !(r < *this);
118 }
```

```
119
120     bool operator>=(const rational &r) const {
121         return !(*this < r);
122     }
123
124     bool operator==(const rational &r) const {
125         return num == r.num && den == r.den;
126     }
127
128     bool operator!=(const rational &r) const {
129         return num != r.num || den != r.den;
130     }
131
132     template<class T>
133     friend bool operator<(const T &a, const rational &b) {
134         return rational(a) < b;
135     }
136
137     template<class T>
138     friend bool operator>(const T &a, const rational &b) {
139         return rational(a) > b;
140     }
141
142     template<class T>
143     friend bool operator<=(const T &a, const rational &b) {
144         return rational(a) <= b;
145     }
146
147     template<class T>
148     friend bool operator>=(const T &a, const rational &b) {
149         return rational(a) >= b;
150     }
151
152     template<class T>
153     friend bool operator==(const T &a, const rational &b) {
154         return rational(a) == b;
155     }
156
157     template<class T>
158     friend bool operator!=(const T &a, const rational &b) {
159         return rational(a) != b;
160     }
161
162     rational abs() const {
163         return rational(num < 0 ? -num : num, den);
164     }
165
166     friend rational abs(const rational &r) { return r.abs(); }
167
168     rational operator+(const rational &r) const {
169         return rational(num*r.den + r.num*den, den*r.den);
170     }
171
172     rational operator-(const rational &r) const {
173         return rational(num*r.den - r.num*den, r.den*den);
174     }
175
176     rational operator*(const rational &r) const {
177         return rational(num*r.num, r.den*den);
```

```
178 }
179
180 rational operator/(const rational &r) const {
181     return rational(num*r.den, den*r.num);
182 }
183
184 rational operator%(const rational &r) const {
185     return *this - r*rational(num*r.den/(r.num*den), 1);
186 }
187
188 template<class T>
189 friend rational operator+(const T &a, const rational &b) {
190     return rational(a) + b;
191 }
192
193 template<class T>
194 friend rational operator-(const T &a, const rational &b) {
195     return rational(a) - b;
196 }
197
198 template<class T>
199 friend rational operator*(const T &a, const rational &b) {
200     return rational(a) * b;
201 }
202
203 template<class T>
204 friend rational operator/(const T &a, const rational &b) {
205     return rational(a) / b;
206 }
207
208 template<class T>
209 friend rational operator%(const T &a, const rational &b) {
210     return rational(a) % b;
211 }
212
213 rational operator-() const { return rational(-num, den); }
214 rational operator++(int) { rational t(*this); operator++(); return t; }
215 rational operator--(int) { rational t(*this); operator--(); return t; }
216 rational& operator++() { *this = *this + 1; return *this; }
217 rational& operator--() { *this = *this - 1; return *this; }
218 rational& operator+=(const rational &r) { *this = *this + r; return *this; }
219 rational& operator-=(const rational &r) { *this = *this - r; return *this; }
220 rational& operator*=(const rational &r) { *this = *this * r; return *this; }
221 rational& operator/=(const rational &r) { *this = *this / r; return *this; }
222 rational& operator%=(const rational &r) { *this = *this % r; return *this; }
223 };
224
225 /* Example Usage */
226
227 #include <cassert>
228 #include <cmath>
229
230 int main() {
231     #define EQ(a, b) (fabs((a) - (b)) <= 1E-9)
232     typedef rational<long long> rational;
233
234     assert(rational(-21, 1) % 2 == -1);
235     rational r(rational(-53, 10) % rational(-17, 10));
236     assert(EQ(r.to_ldouble(), fmod(-5.3, -1.7)));
237 }
```

```

237     assert(r.to_string() == "-1/5");
238     return 0;
239 }
```

5.5 Linear Algebra

5.5.1 Matrix Utilities

```

1  /*
2
3 Basic matrix operations defined on a two-dimensional vector of numeric values.
4
5 - make_matrix(r, c, v) constructs and returns a matrix with r rows and c columns
6 where the value at every index is initialized to v.
7 - make_matrix(a) returns a matrix constructed from the two dimensional array a.
8 - identity_matrix(n) returns the n by n identity matrix, that is, a matrix where
9   a[i][j] equals 1 (if i == j), or 0 otherwise, for every i and j in [0, n).
10 - rows(a) returns the number of rows r in an r by c matrix a.
11 - columns(a) returns the number of columns c in an r by c matrix a.
12 - a[i][j] may be used to access or modify the entry at row i, column j of an r
13 by c matrix a, for every i in [0, r) and j in [0, c).
14 - operators <, >, <=, >=, ==, and != defines lexicographical comparison based on
15 that of std::vector.
16 - operators +, -, *, /, +=, -=, *=, and /= defines scalar addition, subtraction,
17 multiplication, and division involving a matrix a numeric scalar value v.
18 - operators * and *= defines vector and matrix multiplication.
19 - operators ^ and ^= defines matrix exponentiation of a square matrix a by an
20 integer power p.
21 - power_sum(a, p) returns the power sum of a square matrix a up to an integer
22 power p, that is, a + a^2 + ... + a^p.
23 - transpose(a) returns the transpose of an r by c matrix a, that is, a new c by
24 r matrix b such that a[i][j] == b[j][i] for every i in [0, r) and j in [0, c).
25 - transpose_in_place(a) assigns the square matrix a to its transpose, returning
26 a reference to the modified argument itself.
27 - rotate(a, d) returns the matrix a rotated d degrees clockwise. A negative d
28 specifies a counter-clockwise rotation, and d must be a multiple of 90.
29 - rotate_in_place(a, d) assigns the square matrix a to its rotation by d degrees
30 clockwise, returning a reference to the modified argument itself. A negative d
31 specifies a counter-clockwise rotation, and d must be a multiple of 90.
32
33 Time Complexity:
34 - O(n*m) for construction, output, comparison, and scalar arithmetic of n by m
35 matrices.
36 - O(1) for rows(a) and columns(a).
37 - O(n*m) for matrix-matrix addition and subtraction of n by m matrices.
38 - O(n*m*log(p)) for exponentiation of an n by m matrix to power p.
39 - O(n*m*log^2(p)) for power sum of an n by m matrix to power p.
40 - O(n*m*k) for multiplication of an n by m matrix by an m by k matrix.
41 - O(n*m) for transpose(), transpose_in_place(), rotate(), and rotate_in_place()
42 of n by m matrices.
43
44 Space Complexity:
45 - O(1) auxiliary space for rows(), columns(), a[i][j] access, comparison
```

```
46     operators, and in-place operations.
47 - O(n*m*log(p)) auxiliary stack and heap space for exponentiation of an n by m
48   matrix to power p, as well as the power sum of an n by m matrix up to power p.
49 - O(n*m) auxiliary heap space for all non-in-place operations returning an n by
50   m matrix, transpose(), and rotate().
51
52 */
53
54 #include <algorithm>
55 #include <cstddef>
56 #include <iomanip>
57 #include <iostream>
58 #include <stdexcept>
59 #include <stdexcept>
60 #include <vector>
61
62 typedef std::vector<std::vector<int>> matrix;
63
64 matrix make_matrix(int r, int c) {
65     return matrix(r, matrix::value_type(c));
66 }
67
68 template<class T>
69 matrix make_matrix(int r, int c, const T &v) {
70     return matrix(r, matrix::value_type(c, v));
71 }
72
73 template<class T, size_t r, size_t c>
74 matrix make_matrix(T (&a)[r][c]) {
75     matrix res(r, matrix::value_type(c));
76     for (size_t i = 0; i < r; i++) {
77         for (size_t j = 0; j < c; j++) {
78             res[i][j] = a[i][j];
79         }
80     }
81     return res;
82 }
83
84 matrix identity_matrix(int n) {
85     matrix res(n, matrix::value_type(n, 0));
86     for (int i = 0; i < n; i++) {
87         res[i][i] = 1;
88     }
89     return res;
90 }
91
92 int rows(const matrix &a) { return a.size(); }
93 int columns(const matrix &a) { return a.empty() ? 0 : a[0].size(); }
94
95 std::ostream& operator<<(std::ostream &out, const matrix &a) {
96     static const int W = 10, P = 5;
97     for (int i = 0; i < rows(a); i++) {
98         for (int j = 0; j < columns(a); j++) {
99             out << std::setw(W) << std::fixed << std::setprecision(P) << a[i][j];
100        }
101        out << std::endl;
102    }
103    return out;
104 }
```

```

105 template<class T>
106 matrixx& operator+=(matrix &a, const T &v) {
107     for (int i = 0; i < rows(a); i++) {
108         for (int j = 0; j < columns(a); j++) {
109             a[i][j] += v;
110         }
111     }
112     return a;
113 }
114
115 template<class T>
116 matrixx& operator-=(matrix &a, const T &v) {
117     for (int i = 0; i < rows(a); i++) {
118         for (int j = 0; j < columns(a); j++) {
119             a[i][j] -= v;
120         }
121     }
122     return a;
123 }
124
125 template<class T>
126 matrixx& operator*=(matrix &a, const T &v) {
127     for (int i = 0; i < rows(a); i++) {
128         for (int j = 0; j < columns(a); j++) {
129             a[i][j] *= v;
130         }
131     }
132     return a;
133 }
134
135 template<class T>
136 matrixx& operator/=(matrix &a, const T &v) {
137     for (int i = 0; i < rows(a); i++) {
138         for (int j = 0; j < columns(a); j++) {
139             a[i][j] /= v;
140         }
141     }
142     return a;
143 }
144
145 matrixx& operator+=(matrix &a, const matrix &b) {
146     if (rows(a) != rows(b) || columns(a) != columns(b)) {
147         throw std::runtime_error("Invalid dimensions for matrix addition.");
148     }
149     for (int i = 0; i < rows(a); i++) {
150         for (int j = 0; j < columns(a); j++) {
151             a[i][j] += b[i][j];
152         }
153     }
154     return a;
155 }
156
157 matrixx& operator-=(matrix &a, const matrix &b) {
158     if (rows(a) != rows(b) || columns(a) != columns(b)) {
159         throw std::runtime_error("Invalid dimensions for matrix addition.");
160     }
161     for (int i = 0; i < rows(a); i++) {
162         for (int j = 0; j < columns(a); j++) {
163             a[i][j] -= b[i][j];

```

```
164     }
165 }
166 return a;
167 }
168
169 matrix operator+(const matrix &a, const matrix &b) {
170     matrix c(a);
171     return c += b;
172 }
173
174 matrix operator-(const matrix &a, const matrix &b) {
175     matrix c(a);
176     return c -= b;
177 }
178
179 template<class T>
180 matrix& operator*=(matrix &a, const std::vector<T> &v) {
181     if (columns(a) != (int)v.size() || v.empty()) {
182         throw std::runtime_error("Invalid dimensions for matrix multiplication.");
183     }
184     for (int i = 0; i < rows(a); i++) {
185         a[i][0] *= v[0];
186         for (int j = 1; j < columns(a); j++) {
187             a[i][0] += a[i][j]*v[j];
188         }
189     }
190     for (int i = 0; i < rows(a); i++) {
191         a[i].resize(1);
192     }
193     return a;
194 }
195
196 matrix operator*(const matrix &a, const matrix &b) {
197     if (columns(a) != rows(b)) {
198         throw std::runtime_error("Invalid dimensions for matrix multiplication.");
199     }
200     matrix res = make_matrix(rows(a), columns(b), 0);
201     for (int i = 0; i < rows(a); i++) {
202         for (int j = 0; j < columns(b); j++) {
203             for (int k = 0; k < rows(b); k++) {
204                 res[i][j] += a[i][k]*b[k][j];
205             }
206         }
207     }
208     return res;
209 }
210
211 matrix& operator*=(matrix &a, const matrix &b) {
212     return a = a*b;
213 }
214
215 template<class T>
216 matrix operator+(const matrix &a, const T &v) { matrix m(a); return m += v; }
217
218 template<class T>
219 matrix operator-(const matrix &a, const T &v) { matrix m(a); return m -= v; }
220
221 template<class T>
222 matrix operator*(const matrix &a, const T &v) { matrix m(a); return m *= v; }
```

```

223
224 template<class T>
225 matrix operator/(const matrix &a, const T &v) { matrix m(a); return m /= v; }
226
227 template<class T>
228 matrix operator+(const T &v, const matrix &a) { return a + v; }
229
230 template<class T>
231 matrix operator-(const T &v, const matrix &a) { return a - v; }
232
233 template<class T>
234 matrix operator*(const T &v, const matrix &a) { return a * v; }
235
236 template<class T>
237 matrix operator/(const T &v, const matrix &a) { return a / v; }
238
239 matrix operator^(const matrix &a, unsigned int p) {
240     if (rows(a) != columns(a)) {
241         throw std::runtime_error("Matrix must be square for exponentiation.");
242     }
243     if (p == 0) {
244         return identity_matrix(rows(a));
245     }
246     return (p % 2 == 0) ? (a*a)^(p/2) : a*(a^(p - 1));
247 }
248
249 matrix operator^=(matrix &a, unsigned int p) {
250     return a = a ^ p;
251 }
252
253 matrix power_sum(const matrix &a, unsigned int p) {
254     if (rows(a) != columns(a)) {
255         throw std::runtime_error("Matrix must be square for power_sum.");
256     }
257     if (p == 0) {
258         return make_matrix(rows(a), rows(a));
259     }
260     return (p % 2 == 0) ? power_sum(a, p/2)*(identity_matrix(rows(a)) + (a^(p/2)))
261                 : (a + a*power_sum(a, p - 1));
262 }
263
264 matrix transpose(const matrix &a) {
265     matrix res = make_matrix(columns(a), rows(a));
266     for (int i = 0; i < rows(res); i++) {
267         for (int j = 0; j < columns(res); j++) {
268             res[i][j] = a[j][i];
269         }
270     }
271     return res;
272 }
273
274 matrix& transpose_in_place(matrix &a) {
275     if (rows(a) != columns(a)) {
276         throw std::runtime_error("Matrix must be square for transpose_in_place.");
277     }
278     for (int i = 0; i < rows(a); i++) {
279         for (int j = i + 1; j < columns(a); j++) {
280             std::swap(a[i][j], a[j][i]);
281         }

```

```
282     }
283     return a;
284 }
285
286 matrix rotate(const matrix &a, int degrees = 90) {
287     if (degrees % 90 != 0) {
288         throw std::runtime_error("Rotation must be by a multiple of 90 degrees.");
289     }
290     if (degrees < 0) {
291         degrees = 360 - ((-degrees) % 360);
292     }
293     matrix res;
294     switch (degrees % 360) {
295         case 90: {
296             res = make_matrix(columns(a), rows(a));
297             for (int i = 0; i < columns(a); i++) {
298                 for (int j = 0; j < rows(a); j++) {
299                     res[i][j] = a[rows(a) - j - 1][i];
300                 }
301             }
302             break;
303         }
304         case 180: {
305             res = make_matrix(rows(a), columns(a));
306             for (int i = 0; i < rows(a); i++) {
307                 for (int j = 0; j < columns(a); j++) {
308                     res[i][j] = a[rows(a) - i - 1][columns(a) - j - 1];
309                 }
310             }
311             break;
312         }
313         case 270: {
314             res = make_matrix(columns(a), rows(a));
315             for (int i = 0; i < columns(a); i++) {
316                 for (int j = 0; j < rows(a); j++) {
317                     res[i][j] = a[j][columns(a) - i - 1];
318                 }
319             }
320             break;
321         }
322         default: {
323             res = a;
324         }
325     }
326     return res;
327 }
328
329 matrix& rotate_in_place(matrix &a, int degrees = 90) {
330     if (degrees % 90 != 0) {
331         throw std::runtime_error("Rotation must be by a multiple of 90 degrees.");
332     }
333     if (degrees % 180 != 0 && rows(a) != columns(a)) {
334         throw std::runtime_error("Matrix must be square for rotate_in_place.");
335     }
336     if (degrees < 0) {
337         degrees = 360 - ((-degrees) % 360);
338     }
339     int n = rows(a);
340     switch (degrees % 360) {
```

```

341     case 90: {
342         transpose_in_place(a);
343         for (int i = 0; i < n; i++) {
344             std::reverse(a[i].begin(), a[i].end());
345         }
346         break;
347     }
348     case 180: {
349         for (int i = 0; i < columns(a); i++) {
350             for (int j = 0, k = n - 1; j < k; j++, k--) {
351                 std::swap(a[i][j], a[i][k]);
352             }
353         }
354         for (int j = 0; j < n; j++) {
355             for (int i = 0, k = columns(a) - 1; i < k; i++, k--) {
356                 std::swap(a[i][j], a[k][j]);
357             }
358         }
359         break;
360     }
361     case 270: {
362         transpose_in_place(a);
363         for (int j = 0; j < n; j++) {
364             for (int i = 0, k = columns(a) - 1; i < k; i++, k--) {
365                 std::swap(a[i][j], a[k][j]);
366             }
367         }
368         break;
369     }
370 }
371 return a;
372 }

373 /**
374  * Example Usage */
375

376 #include <cassert>
377 #include <iostream>
378 using namespace std;
379
380 int main() {
381     int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
382     int a90[3][2] = {{4, 1}, {5, 2}, {6, 3}};
383     int a180[2][3] = {{6, 5, 4}, {3, 2, 1}};
384     int a270[3][2] = {{3, 6}, {2, 5}, {1, 4}};
385     cout << make_matrix(a) << endl;
386     assert(rotate(make_matrix(a), -270) == make_matrix(a90));
387     assert(rotate(make_matrix(a), -180) == make_matrix(a180));
388     assert(rotate(make_matrix(a), -90) == make_matrix(a270));
389     assert(rotate(make_matrix(a), 0) == make_matrix(a));
390     assert(rotate(make_matrix(a), 90) == make_matrix(a90));
391     assert(rotate(make_matrix(a), 180) == make_matrix(a180));
392     assert(rotate(make_matrix(a), 270) == make_matrix(a270));
393     assert(rotate(make_matrix(a), 360) == make_matrix(a));
394
395     int b[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
396     for (int d = -360; d <= 360; d += 90) {
397         matrix m = make_matrix(b);
398         assert(rotate_in_place(m, d) == rotate(make_matrix(b), d));
399     }
}

```

```

400
401 matrix m = make_matrix(5, 5, 10) + 10;
402 int v[] = {1, 2, 3, 4, 5}, mv[5][1] = {{300}, {300}, {300}, {300}, {300}};
403 assert(m*vector<int>(v, v + 5) == make_matrix(mv));
404
405 m[0][0] += 5;
406 assert(m[0][0] == 25 && m[1][1] == 20);
407 assert(power_sum(m, 3) == m + m*m + (m^3));
408 return 0;
409 }
```

5.5.2 Row Reduction

```

1 /*
2
3 Converts a matrix to reduced row echelon form using Gaussian elimination to
4 solve a system of linear equations as well as compute the determinant. In
5 practice, this method is prone to rounding error on certain matrices. For a more
6 accurate algorithm for solving systems of linear equations, LU decomposition
7 with row partial pivoting should be used.
8
9 - row_reduce(a) assigns the matrix a to its reduced row echelon form, returning
10 a reference to the modified argument itself.
11 - solve_system(a, b, &x) solves the system of linear equations a*x = b given an
12 r by c matrix a of real values, and a length r vector b, returning 0 if there
13 is one solution, -1 if there are zero solutions, or -2 if there are infinite
14 solutions. If there is exactly one solution, then the vector pointed to by x
15 is populated with the solution vector of length c.
16
17 Time Complexity:
18 - O(r^2*c) per call to row_reduce(a) and solve_system(a), where r and c are the
19 number of rows and columns of a respectively.
20
21 Space Complexity:
22 - O(1) auxiliary for row_reduce(a).
23 - O(r*c) auxiliary heap space for solve_system(a).
24 */
25
26 #include <cmath>
27 #include <cstddef>
28 #include <stdexcept>
29 #include <vector>
30
31
32 const double EPS = 1e-9;
33
34 template<class Matrix>
35 Matrix& row_reduce(Matrix &a) {
36     if (a.empty()) {
37         return a;
38     }
39     int r = a.size(), c = a[0].size(), lead = 0;
40     for (int row = 0; row < r && lead < c; row++) {
41         int i = row;
42         while (fabs(a[i][lead]) < EPS) {
```

```

43     if (++i == r) {
44         i = row;
45         if (++lead == c) {
46             return a;
47         }
48     }
49 }
50 std::swap(a[i], a[row]);
51 typename Matrix::value_type::value_type lv = a[row][lead];
52 for (int j = 0; j < c; j++) {
53     a[row][j] /= lv;
54 }
55 for (int i = 0; i < r; i++) {
56     if (i != row) {
57         lv = a[i][lead];
58         for (int j = 0; j < c; j++) {
59             a[i][j] -= lv*a[row][j];
60         }
61     }
62 }
63 for (int j = 0; j < lead; j++) {
64     a[row][j] = 0;
65 }
66 a[row][lead++] = 1;
67 }
68 return a;
69 }

70
71 template<class Matrix, class T>
72 int solve_system(const Matrix &a, const std::vector<T> &b, std::vector<T> *x) {
73     if (x == NULL || a.empty() || a.size() != b.size()) {
74         return -1;
75     }
76     int r = a.size(), c = a[0].size();
77     if (r < c) {
78         return -2;
79     }
80     Matrix m(a);
81     for (int i = 0; i < r; i++) {
82         m[i].push_back(b[i]);
83     }
84     row_reduce(m);
85     for (int i = 0; i < r; i++) {
86         int lead = -1;
87         for (int j = 0; j < c && lead < 0; j++) {
88             if (fabs(m[i][j]) > EPS) {
89                 lead = j;
90             }
91         }
92         if (lead < 0 && fabs(m[i][c]) > EPS) {
93             return -1;
94         }
95         if (lead > i) {
96             return -2;
97         }
98     }
99     x->resize(c);
100    for (int i = 0; i < c; i++) {
101        (*x)[i] = m[i][c];

```

```

102     }
103     return 0;
104 }
105
106 /*** Example Usage ***/
107
108 #include <cassert>
109 using namespace std;
110
111 int main() {
112     const int equations = 3, unknowns = 3;
113     const int a[equations][unknowns] = {{-1, 2, 5}, {1, 0, -6}, {-4, 2, 2}};
114     const int b[equations] = {3, 1, -2};
115     vector<vector<double>> m(equations);
116     for (int i = 0; i < equations; i++) {
117         m[i].assign(a[i], a[i] + unknowns);
118     }
119     vector<double> x;
120     assert(solve_system(m, vector<double>(b, b + equations), &x) == 0);
121     for (int i = 0; i < equations; i++) {
122         double sum = 0;
123         for (int j = 0; j < unknowns; j++) {
124             sum += a[i][j]*x[j];
125         }
126         assert(fabs(sum - b[i]) < EPS);
127     }
128     return 0;
129 }
```

5.5.3 Determinant and Inverse

```

1 /*
2
3 Computes the determinant and inverse of a square matrix using Gaussian
4 elimination. The inverse of a matrix a is another matrix b such that a*b equals
5 the identity matrix. The inverse of a exists if and only if the determinant of a
6 is zero. In this case, a is called invertible or non-singular. In practice,
7 simple Gaussian elimination is prone to rounding error on certain matrices. For
8 a more accurate algorithm for solving systems of linear equations, see LU
9 decomposition with row partial pivoting should be.
10
11 - det_naive(a) returns the determinant of an n by n matrix a, using the classic
12 divide-and-conquer algorithm by Laplace expansions.
13 - det(a) returns the determinant of an n by n matrix a using Gaussian
14 elimination.
15 - invert(a) assigns the n by n matrix a to its inverse (if it exists), returning
16 a reference to the modified argument itself. If a is not invertible, then its
17 assigned values after the function call will be undefined (+/-Inf or +/-NaN).
18
19 Time Complexity:
20 - O(n!) per call to det_naive(), where n is the dimension of the matrix.
21 - O(n^3) per call to det() and invert() where n is the dimension of the matrix.
22
23 Space Complexity:
24 - O(n) auxiliary stack space and O(n!*n) auxiliary heap space for det_naive(),
```

```

25     where n is the dimension of the matrix.
26 - O(n^2) auxiliary heap space for det() and invert().
27
28 */
29
30 #include <cmath>
31 #include <map>
32 #include <vector>
33
34 template<class SquareMatrix>
35 double det_naive(const SquareMatrix &a) {
36     int n = a.size();
37     if (n == 1) {
38         return a[0][0];
39     }
40     if (n == 2) {
41         return a[0][0]*a[1][1] - a[0][1]*a[1][0];
42     }
43     double res = 0;
44     SquareMatrix temp(n - 1, typename SquareMatrix::value_type(n - 1));
45     for (int p = 0; p < n; p++) {
46         int h = 0, k = 0;
47         for (int i = 1; i < n; i++) {
48             for (int j = 0; j < n; j++) {
49                 if (j == p) {
50                     continue;
51                 }
52                 temp[h][k++] = a[i][j];
53                 if (k == n - 1) {
54                     h++;
55                     k = 0;
56                 }
57             }
58         }
59         res += (p % 2 == 0 ? 1 : -1)*a[0][p]*det_naive(temp);
60     }
61     return res;
62 }
63
64 template<class SquareMatrix>
65 double det(const SquareMatrix &a, double EPS = 1e-10) {
66     SquareMatrix b(a);
67     int n = a.size();
68     double res = 1.0;
69     std::vector<bool> used(n, false);
70     for (int i = 0; i < n; i++) {
71         int p;
72         for (p = 0; p < n; p++) {
73             if (!used[p] && fabs(b[p][i]) > EPS) {
74                 break;
75             }
76         }
77         if (p >= n) {
78             return 0;
79         }
80         res *= b[p][i];
81         used[p] = true;
82         double z = 1.0/b[p][i];
83         for (int j = 0; j < n; j++) {

```

```

84     b[p][j] *= z;
85 }
86 for (int j = 0; j < n; j++) {
87     if (j != p) {
88         z = b[j][i];
89         for (int k = 0; k < n; k++) {
90             b[j][k] -= z*b[p][k];
91         }
92     }
93 }
94 return res;
95 }
96 }

97 template<class SquareMatrix>
98 SquareMatrix& invert(SquareMatrix &a) {
99     int n = a.size();
100    for (int i = 0; i < n; i++) {
101        a[i].resize(2*n);
102        for (int j = n; j < n*2; j++) {
103            a[i][j] = (i == j - n ? 1 : 0);
104        }
105    }
106    for (int i = 0; i < n; i++) {
107        double z = a[i][i];
108        for (int j = i; j < n*2; j++) {
109            a[i][j] /= z;
110        }
111        for (int j = 0; j < n; j++) {
112            if (i != j) {
113                double z = a[j][i];
114                for (int k = 0; k < n*2; k++) {
115                    a[j][k] -= z*a[i][k];
116                }
117            }
118        }
119    }
120 }
121 for (int i = 0; i < n; i++) {
122     a[i].erase(a[i].begin(), a[i].begin() + n);
123 }
124 return a;
125 }

126 /**
127  * Example Usage */
128
129 #include <cassert>
130 using namespace std;
131
132 int main() {
133     const int n = 3, a[n][n] = {{6, 1, 1}, {4, -2, 5}, {2, 8, 7}};
134     vector<vector<double>> m(n), inv, res(n, vector<double>(n, 0));
135     for (int i = 0; i < n; i++) {
136         m[i] = vector<double>(a[i], a[i] + n);
137     }
138     double d = det(m);
139     assert(fabs(d - det_naive(m)) < 1e-10);
140     invert(inv = m);
141     for (int i = 0; i < n; i++) {
142         for (int j = 0; j < n; j++) {

```

```

143     for (int k = 0; k < n; k++) {
144         res[i][j] += a[i][k]*inv[k][j];
145     }
146 }
147 }
148 for (int i = 0; i < n; i++) {
149     for (int j = 0; j < n; j++) {
150         assert(fabs(res[i][j] - (i == j ? 1 : 0)) < 1e-10);
151     }
152 }
153 return 0;
154 }
```

5.5.4 LU Decomposition

```

1 /*
2
3 The LU decomposition of a matrix a with row-partial pivoting is a factorization
4 of a (after some rows are possibly permuted by a permutation matrix p) as a
5 product of a lower triangular matrix l and an upper triangular matrix u. This
6 factorization can be used to tackle many common problems in linear algebra such
7 as solving systems of linear equations and computing determinants. An
8 improvement on basic row reduction, LU decomposition by row-partial pivoting
9 keeps the relative magnitude of matrix values small, thus reducing the relative
10 error due to rounding in computed solutions.
11
12 - lu_decompose(a, &picol) assigns the r by c matrix a to merged LU decomposition
13 matrix lu, returning either 0 or 1 denoting the "sign" of the permutation
14 parity (0 if the number of overall row swaps performed is even, or 1 if it is
15 odd), or -1 denoting a degenerate matrix (i.e. singular for square matrices).
16 The merged matrix lu has lu[i][j] = l[i][j] for i > j and lu[i][j] = u[i][j]
17 for i <= j. Note that the algorithm always yields an atomic lower triangular
18 matrix for which the diagonal entries l[i][i] are always equal to 1, so this
19 is not explicitly stored in the resulting merged matrix. For general i and j,
20 the values of the lower and upper triangular matrices should be accessed via
21 the getl(lu, i, j) and getu(lu, i, j) functions. Optionally, a vector<int>
22 pointer picol may be passed to return the permutation vector picol where
23 picol[i] stores the only column that is equal to 1 in row i of the permutation
24 matrix p (all other columns in row i of p are implicitly 0). The resulting
25 permutation matrix p corresponding to picol will satisfy p*a = l*u.
26 - solve_system(a, b, &x) solves the system of linear equations a*x = b given an
27 r by c matrix a of real values, and a length r vector b, returning 0 if there
28 is one solution or -1 if there are zero or infinite solutions. If there is
29 exactly one solution, then the vector pointed to by x is populated with the
30 solution vector of length c.
31 - det(a) returns the determinant of an n by n matrix a using LU decomposition.
32 - invert(a) assigns the n by n matrix a to its inverse (if it exists), returning
33 0 if the inversion was successful or -1 if a has no inverse.
34
35 Time Complexity:
36 - O(r^2*c) per call to lu_decompose(a) and solve_system(a, b), where r and c are
37 the number of rows and columns respectively, in accordance to the functions'
38 descriptions above.
39 - O(n^3) per call to det(a) and inverse(a), where n is the dimension of a.
40
```

```

41 Space Complexity:
42 - O(1) auxiliary for lu_decompose().
43 - O(n^2) for det(a) and inverse(a).
44 - O(r*c) auxiliary heap space for solve_system(a, b).
45 */
46
47
48 #include <algorithm>
49 #include <cmath>
50 #include <cstddef>
51 #include <limits>
52 #include <vector>
53
54 template<class Matrix>
55 int lu_decompose(Matrix &a, std::vector<int> *p1col = NULL,
56                   const double EPS = 1e-10) {
57     int r = a.size(), c = a[0].size(), parity = 0;
58     if (p1col != NULL) {
59         p1col->resize(r);
60         for (int i = 0; i < r; i++) {
61             (*p1col)[i] = i;
62         }
63     }
64     for (int i = 0; i < r && i < c; i++) {
65         int pi = i;
66         for (int k = i + 1; k < r; k++) {
67             if (fabs(a[k][i]) > fabs(a[pi][i])) {
68                 pi = k;
69             }
70         }
71         if (fabs(a[pi][i]) < EPS) {
72             return -1;
73         }
74         if (pi != i) {
75             if (p1col != NULL) {
76                 std::iter_swap(p1col->begin() + i, p1col->begin() + pi);
77             }
78             std::iter_swap(a.begin() + i, a.begin() + pi);
79             parity = 1 - parity;
80         }
81         for (int j = i + 1; j < r; j++) {
82             a[j][i] /= a[i][i];
83             for (int k = i + 1; k < c; k++) {
84                 a[j][k] -= a[j][i]*a[i][k];
85             }
86         }
87     }
88     return parity;
89 }
90
91 template<class Matrix>
92 double getl(const Matrix &lu, int i, int j) {
93     return i > j ? lu[i][j] : (i < j ? 0 : 1);
94 }
95
96 template<class Matrix>
97 double getu(const Matrix &lu, int i, int j) {
98     return i <= j ? lu[i][j] : 0;
99 }
```

```

100
101 template<class Matrix, class T>
102 int solve_system(const Matrix &a, const std::vector<T> &b, std::vector<T> *x,
103                  const double EPS = 1e-10) {
104     int r = a.size(), c = a[0].size();
105     if (x == NULL || a.empty() || a.size() != b.size() || r < c) {
106         return -1;
107     }
108     x->resize(c);
109     std::vector<int> p1col;
110     Matrix lu;
111     int status = lu_decompose(lu = a, &p1col, EPS);
112     if (status < 0) {
113         return status;
114     }
115     for (int i = 0; i < c; i++) {
116         (*x)[i] = b[p1col[i]];
117         for (int k = 0; k < i; k++) {
118             (*x)[i] -= getl(lu, i, k)*(*x)[k];
119         }
120     }
121     for (int i = c - 1; i >= 0; i--) {
122         for (int k = i + 1; k < c; k++) {
123             (*x)[i] -= getu(lu, i, k)*(*x)[k];
124         }
125         (*x)[i] /= getu(lu, i, i);
126     }
127     for (int i = 0; i < r; i++) {
128         double val = 0;
129         for (int j = 0; j < c; j++) {
130             val += a[i][j]*(*x)[j];
131         }
132         if (fabs(val - b[i])/b[i] > EPS) {
133             return -1;
134         }
135     }
136     return 0;
137 }
138
139 template<class SquareMatrix>
140 double det(const SquareMatrix &a) {
141     int n = a.size();
142     SquareMatrix lu;
143     int status = lu_decompose(lu = a);
144     if (status < 0) {
145         return 0;
146     }
147     double res = 1;
148     for (int i = 0; i < n; i++) {
149         res *= lu[i][i];
150     }
151     return status == 0 ? res : -res;
152 }
153
154 template<class SquareMatrix>
155 int invert(SquareMatrix &a) {
156     int n = a.size();
157     std::vector<int> p1col;
158     int status = lu_decompose(a, &p1col);

```

```

159     if (status < 0) {
160         return status;
161     }
162     SquareMatrix ia(n, typename SquareMatrix::value_type(n, 0));
163     for (int j = 0; j < n; j++) {
164         for (int i = 0; i < n; i++) {
165             if (pcol[i] == j) {
166                 ia[i][j] = 1.0;
167             } else {
168                 ia[i][j] = 0.0;
169             }
170             for (int k = 0; k < i; k++) {
171                 ia[i][j] -= getl(a, i, k)*ia[k][j];
172             }
173         }
174         for (int i = n - 1; i >= 0; i--) {
175             for (int k = i + 1; k < n; k++) {
176                 ia[i][j] -= getu(a, i, k)*ia[k][j];
177             }
178             ia[i][j] /= getu(a, i, i);
179         }
180     }
181     a.swap(ia);
182     return 0;
183 }
184
185 /** Example Usage **/
186
187 #include <cassert>
188 using namespace std;
189
190 int main() {
191     // Solve a system.
192     const int equations = 3, unknowns = 3;
193     const int a[equations][unknowns] = {{-1, 2, 5}, {1, 0, -6}, {-4, 2, 2}};
194     const int b[equations] = {3, 1, -2};
195     vector<vector<double>> m(equations);
196     for (int i = 0; i < equations; i++) {
197         m[i].assign(a[i], a[i] + unknowns);
198     }
199     vector<double> x;
200     assert(solve_system(m, vector<double>(b, b + equations), &x) == 0);
201     for (int i = 0; i < equations; i++) {
202         double sum = 0;
203         for (int j = 0; j < unknowns; j++) {
204             sum += a[i][j]*x[j];
205         }
206         assert(fabs(sum - b[i]) < 1e-10);
207     }
208 }
209 // Find the determinant.
210 const int n = 3, a[n][n] = {{1, 3, 5}, {2, 4, 7}, {1, 1, 0}};
211 vector<vector<double>> m(n);
212 for (int i = 0; i < n; i++) {
213     m[i] = vector<double>(a[i], a[i] + n);
214 }
215 assert(fabs(det(m) - 4) < 1e-10);
216
217 // Find the inverse.

```

```

218     const int n = 3, a[n][n] = {{6, 1, 1}, {4, -2, 5}, {2, 8, 7}};
219     vector<vector<double>> m(n), res(n, vector<double>(n, 0));
220     for (int i = 0; i < n; i++) {
221         m[i] = vector<double>(a[i], a[i] + n);
222     }
223     assert(invert(m) == 0);
224     for (int i = 0; i < n; i++) {
225         for (int j = 0; j < n; j++) {
226             for (int k = 0; k < n; k++) {
227                 res[i][j] += a[i][k]*m[k][j];
228             }
229         }
230     }
231     for (int i = 0; i < n; i++) {
232         for (int j = 0; j < n; j++) {
233             assert(fabs(res[i][j] - (i == j ? 1 : 0)) < 1e-10);
234         }
235     }
236 }
237     return 0;
238 }
```

5.5.5 Linear Programming (Simplex)

```

1 /*
2
3 Solves a linear programming problem using Dantzig's simplex algorithm. The
4 canonical form of a linear programming problem is to maximize (or minimize) the
5 dot product c*x, subject to a*x <= b and x >= 0, where x is a vector of unknowns
6 to be solved, c is a vector of coefficients, a is a matrix of linear equation
7 coefficients, and b is a vector of boundary coefficients.
8
9 - simplex_solve(a, b, c, &x) solves the linear programming problem for an m by n
10 matrix a of real values, a length m vector b, a length n vector c, returning 0
11 if a solution was found or -1 if there are no solutions. If a solution is
12 found, then the vector pointed to by x is populated with the solution vector
13 of length n.
14
15 Time Complexity:
16 - Polynomial (average) on the number of equations and unknowns, but exponential
17 in the worst case.
18
19 Space Complexity:
20 - O(m*n) auxiliary heap space.
21
22 */
23
24 #include <cmath>
25 #include <limits>
26 #include <vector>
27
28 template<class Matrix>
29 int simplex_solve(const Matrix &a, const std::vector<double> &b,
30                   const std::vector<double> &c, std::vector<double> *x,
31                   const bool MAXIMIZE = true, const double EPS = 1e-10) {
```

```

32     int m = a.size(), n = c.size();
33     Matrix t(m + 2, std::vector<double>(n + 2));
34     t[1][1] = 0;
35     for (int j = 1; j <= n; j++) {
36         t[1][j + 1] = MAXIMIZE ? c[j - 1] : -c[j - 1];
37     }
38     for (int i = 1; i <= m; i++) {
39         for (int j = 1; j <= n; j++) {
40             t[i + 1][j + 1] = -a[i - 1][j - 1];
41         }
42         t[i + 1][1] = b[i - 1];
43     }
44     for (int j = 1; j <= n; j++) {
45         t[0][j + 1] = j;
46     }
47     for (int i = n + 1; i <= m + n; i++) {
48         t[i - n + 1][0] = i;
49     }
50     double p1 = 0, p2 = 0;
51     bool done = true;
52     do {
53         double mn = std::numeric_limits<double>::max(), xmax = 0, v;
54         for (int j = 2; j <= n + 1; j++) {
55             if (t[1][j] > 0 && t[1][j] > xmax) {
56                 p2 = j;
57                 xmax = t[1][j];
58             }
59         }
60         for (int i = 2; i <= m + 1; i++) {
61             v = fabs(t[i][1] / t[i][p2]);
62             if (t[i][p2] < 0 && mn > v) {
63                 mn = v;
64                 p1 = i;
65             }
66         }
67         std::swap(t[p1][0], t[0][p2]);
68         for (int i = 1; i <= m + 1; i++) {
69             if (i != p1) {
70                 for (int j = 1; j <= n + 1; j++) {
71                     if (j != p2) {
72                         t[i][j] -= t[p1][j]*t[i][p2] / t[p1][p2];
73                     }
74                 }
75             }
76         }
77         t[p1][p2] = 1.0 / t[p1][p2];
78         for (int j = 1; j <= n + 1; j++) {
79             if (j != p2) {
80                 t[p1][j] *= fabs(t[p1][p2]);
81             }
82         }
83         for (int i = 1; i <= m + 1; i++) {
84             if (i != p1) {
85                 t[i][p2] *= t[p1][p2];
86             }
87         }
88         for (int i = 2; i <= m + 1; i++) {
89             if (t[i][1] < 0) {
90                 return -1;
91             }
92         }
93     } while (!done);
94 }
```

```

91         }
92     }
93     done = true;
94     for (int j = 2; j <= n + 1; j++) {
95         if (t[1][j] > 0) {
96             done = false;
97         }
98     }
99 } while (!done);
100 x->clear();
101 for (int j = 1; j <= n; j++) {
102     for (int i = 2; i <= m + 1; i++) {
103         if (fabs(t[i][0] - j) < EPS) {
104             x->push_back(t[i][1]);
105         }
106     }
107 }
108 return 0;
109 }
110
111 /** Example Usage and Output:
112
113 Solution = 33.3043 at (5.30435, 4.34783).
114
115 ***
116
117 #include <cassert>
118 #include <iostream>
119 using namespace std;
120
121 int main() {
122     // Solve [x, y] that maximizes 3x + 4y, subject to x, y >= 0 and:
123     // -2x + 1y <= 0
124     // 1x + 0.85y <= 9
125     // 1x + 2y <= 14
126     const int equations = 3, unknowns = 2;
127     double a[equations][unknowns] = {{-2, 1}, {1, 0.85}, {1, 2}};
128     double b[equations] = {0, 9, 14};
129     double c[unknowns] = {3, 4};
130     vector<vector<double>> va(equations, vector<double>(unknowns));
131     vector<double> vb(b, b + equations), vc(c, c + unknowns), x;
132     for (int i = 0; i < equations; i++) {
133         for (int j = 0; j < unknowns; j++) {
134             va[i][j] = a[i][j];
135         }
136     }
137     assert(simplex_solve(va, vb, vc, &x) == 0);
138     double maxval = 0;
139     for (int i = 0; i < (int)x.size(); i++) {
140         maxval += c[i]*x[i];
141     }
142     cout << "Solution = " << maxval << " at (" << x[0];
143     for (int i = 1; i < (int)x.size(); i++) {
144         cout << ", " << x[i];
145     }
146     cout << ")" . " << endl;
147     return 0;
148 }
```

5.6 Root Finding and Calculus

5.6.1 Root Finding (Bracketing)

```

1  /*
2
3 Finds an x in an interval [a, b] for a continuous function f such that f(x) = 0.
4 By the intermediate value theorem, a root must exist in [a, b] if the signs of
5 f(a) and f(b) differ. The answer is found with an absolute error of roughly
6 1/(2^n), where n is the number of iterations. Although it is possible to control
7 the error by looping while b - a is greater than an arbitrary epsilon, it is
8 simpler to let the loop run for a desired number of iterations until floating
9 point arithmetic break down. 100 iterations is usually sufficient, since the
10 search space will be reduced to 2^-100 (roughly 10^-30) times its original size.
11
12 - bisection_root(f, a, b) returns a root in an interval [a, b] for a continuous
13   function f where sgn(f(a)) != sgn(f(b)), using the bisection method.
14 - falsi_illinois_root(f, a, b) returns a root in an interval [a, b] for a continuous
15   function f where sgn(f(a)) != sgn(f(b)), using the Illinois algorithm variant
16   of the false position (a.k.a. regula falsi) method.
17
18 Time Complexity:
19 - O(n) calls will be made to f() in bisection_root() and falsi_illinois_root(),
20   where n is the number of iterations performed.
21
22 Space Complexity:
23 - O(1) auxiliary space for both operations.
24
25 */
26
27 #include <stdexcept>
28
29 template<class ContinuousFunction>
30 double bisection_root(ContinuousFunction f, double a, double b,
31                      const int ITERATIONS = 100) {
32     if (a > b || f(a)*f(b) > 0) {
33         throw std::runtime_error("Must give [a, b] where sgn(f(a)) != sgn(f(b)).");
34     }
35     double m;
36     for (int i = 0; i < ITERATIONS; i++) {
37         m = a + (b - a)/2;
38         if (f(a)*f(m) >= 0) {
39             a = m;
40         } else {
41             b = m;
42         }
43     }
44     return m;
45 }
46
47 template<class ContinuousFunction>
48 double falsi_illinois_root(ContinuousFunction f, double a, double b,
49                           const int ITERATIONS = 100) {
50     if (a > b || f(a)*f(b) > 0) {
51         throw std::runtime_error("Must give [a, b] where sgn(f(a)) != sgn(f(b)).");
52     }

```

```

53  double m, fm, fa = f(a), fb = f(b);
54  int side = 0;
55  for (int i = 0; i < ITERATIONS; i++) {
56      m = (fa*b - fb*a)/(fa - fb);
57      fm = f(m);
58      if (fb*fm > 0) {
59          b = m;
60          fb = fm;
61          if (side < 0) {
62              fa /= 2;
63          }
64          side = -1;
65      } else if (fa*fm > 0) {
66          a = m;
67          fa = fm;
68          if (side > 1) {
69              fb /= 2;
70          }
71          side = 1;
72      } else {
73          break;
74      }
75  }
76  return m;
77 }

79 /** Example Usage ***/
80
81 #include <cassert>
82 #include <cmath>
83
84 double f(double x) {
85     return x*x - 4*sin(x);
86 }
87
88 int main() {
89     assert(fabs(f(bisection_root(f, 1, 3))) < 1e-10);
90     assert(fabs(f(falsi_illinois_root(f, 1, 3))) < 1e-10);
91     return 0;
92 }
```

5.6.2 Root Finding (Iteration)

```

1 /*
2
3 Finds an x for a continuous function f such that f(x) = 0 using iterative
4 approximation by an initial guess that is close to the answer. Newton's method
5 requires an explicit definition of the function's derivative while the secant
6 method starts with two initial guesses and approximates the derivative using the
7 secant slope from the previous iteration. For n iterations and a good initial
8 guess, the methods below compute approximately  $2^n$  digits of precision, with the
9 secant method converging approximately 1.6 times slower than Newton's.
10
11 - newton_root(f, fprime, x0) returns a root x for a function f with derivative
12   fprime using an initial guess x0 which should be relatively close to x.
```

```
13 - secant_root(f, x0, x1) returns a root x for a function f using two initial
14 guesses x0 and x1 which should be relatively close to x.
15
16 Time Complexity:
17 - O(n) calls will be made to f() in newton_root() and secant_root(), where n is
18 the number of iterations performed.
19
20 Space Complexity:
21 - O(1) auxiliary space for both operations.
22
23 */
24
25 #include <cmath>
26 #include <stdexcept>
27
28 template<class ContinuousFunction>
29 double newton_root(ContinuousFunction f, ContinuousFunction fprime, double x0,
30                     const double EPS = 1e-15, const int ITERATIONS = 100) {
31     double x = x0, error = EPS + 1;
32     for (int i = 0; error > EPS && i < ITERATIONS; i++) {
33         double xnew = x - f(x)/fprime(x);
34         error = fabs(xnew - x);
35         x = xnew;
36     }
37     if (error > EPS) {
38         throw std::runtime_error("Newton's method failed to converge.");
39     }
40     return x;
41 }
42
43 template<class ContinuousFunction>
44 double secant_root(ContinuousFunction f, double x0, double x1,
45                     const double EPS = 1e-15, const int ITERATIONS = 100) {
46     double xold = x0, fxold = f(x0), x = x1, error = EPS + 1;
47     for (int i = 0; error > EPS && i < ITERATIONS; i++) {
48         double fx = f(x);
49         double xnew = x - fx*((x - xold)/(fx - fxold));
50         xold = x;
51         fxold = fx;
52         error = fabs(xnew - x);
53         x = xnew;
54     }
55     if (error > EPS) {
56         throw std::runtime_error("Secant method failed to converge.");
57     }
58     return x;
59 }
60
61 /** Example Usage ***/
62
63 #include <cassert>
64
65 double f(double x) {
66     return x*x - 4*sin(x);
67 }
68
69 double fprime(double x) {
70     return 2*x - 4*cos(x);
71 }
```

```

72
73 int main() {
74     assert(fabs(f(newton_root(f, fprime, 3))) < 1e-10);
75     assert(fabs(f(secant_root(f, 3, 2))) < 1e-10);
76     return 0;
77 }
```

5.6.3 Polynomial Root Finding (Differentiation)

```

1 /*
2
3 Finds every root x for a polynomial p such that p(x) = 0 by differentiation.
4 Each adjacent pair of local extrema is searched using the bisection method,
5 where local extrema are recursively found by finding the root of the derivative.
6
7 - horner_eval(p, x) evaluates the polynomial p of degree d (represented as a
8   vector of size d + 1 where p[i] stores the coefficient for the  $x^i$  term) at x,
9   using Horner's method.
10 - find_one_root(p, a, b, EPS) returns a root in the interval [a, b] for a
11   polynomial p where  $\text{sgn}(f(a)) \neq \text{sgn}(f(b))$ , using the bisection method. If this
12   precondition is not satisfied, then NaN is returned. The root is found to a
13   tolerance of EPS in absolute or relative error (whichever is reached first).
14 - find_all_roots(p, a, b, EPS) returns a vector of all roots in the interval
15   [a, b] for a polynomial p using the bisection method. The roots are found to a
16   tolerance of EPS in absolute or relative error (whichever is reached first).
17
18 Time Complexity:
19 -  $O(n)$  per call to horner_eval(), where n is the degree of the polynomial.
20 -  $O(n \log p)$  per call to find_one_root(), where n is the degree of the
21   polynomial and p = -log10(EPS) is the number of digits of absolute or relative
22   precision that is desired.
23 -  $O(n^3 \log p)$  per call to find_all_roots(), where n is the degree of the
24   polynomial and p = -log10(EPS) is the number of digits of absolute or relative
25   precision that is desired.
26
27 Space Complexity:
28 -  $O(1)$  auxiliary space for horner_eval() and find_one_root().
29 -  $O(n^2)$  auxiliary heap and  $O(n)$  auxiliary stack space for find_all_roots(),
30   where n is the degree of the polynomial.
31 */
32
33
34 #include <cmath>
35 #include <limits>
36 #include <utility>
37 #include <vector>
38
39 double horner_eval(const std::vector<double> &p, double x) {
40     double res = p.back();
41     for (int i = (int)p.size() - 2; i >= 0; i--) {
42         res = res*x + p[i];
43     }
44     return res;
45 }
```

```

47 double find_one_root(const std::vector<double> &p, double a, double b,
48                     const double EPS = 1e-15) {
49     double pa = horner_eval(p, a), pb = horner_eval(p, b);
50     bool paneg = pa < 0, pbneg = pb < 0;
51     if (paneg == pbneg) {
52         return std::numeric_limits<double>::quiet_NaN();
53     }
54     while (b - a > EPS && a*(1 + EPS) < b && a < b*(1 + EPS)) {
55         double m = a + (b - a)/2;
56         if ((horner_eval(p, m) < 0) == paneg) {
57             a = m;
58         } else {
59             b = m;
60         }
61     }
62     return a;
63 }
64
65 std::vector<double> find_all_roots(const std::vector<double> &p,
66                                     double a = -1e20, double b = 1e20,
67                                     const double EPS = 1e-15) {
68     std::vector<double> pprime;
69     for (int i = 1; i < (int)p.size(); i++) {
70         pprime.push_back(p[i]*i);
71     }
72     if (pprime.empty()) {
73         return std::vector<double>();
74     }
75     std::vector<double> res, r = find_all_roots(pprime, a, b, EPS);
76     r.push_back(b);
77     for (int i = 0; i < (int)r.size(); i++) {
78         double root = find_one_root(p, i == 0 ? a : r[i - 1], r[i], EPS);
79         if (!std::isnan(root) && (res.empty() || root != res.back())) {
80             res.push_back(root);
81         }
82     }
83     return res;
84 }
85
86 /*** Example Usage ***/
87
88 #include <cassert>
89 using namespace std;
90
91 int main() {
92     { // -1 + 2x - 6x^2 + 2x^3
93         int poly[] = {-1, 2, -6, 2};
94         vector<double> p(poly, poly + 4), roots = find_all_roots(p);
95         assert(roots.size() == 1 && fabs(horner_eval(p, roots[0])) < 1e-10);
96     }
97     { // -20 + 4x + 3x^2
98         int poly[] = {-20, 4, 3};
99         vector<double> p(poly, poly + 3), roots = find_all_roots(p);
100        assert(roots.size() == 2);
101        assert(fabs(horner_eval(p, roots[0])) < 1e-10);
102        assert(fabs(horner_eval(p, roots[1])) < 1e-10);
103    }
104    return 0;
105 }
```

5.6.4 Polynomial Root Finding (Laguerre)

```

1  /*
2
3 Finds every complex root x for a polynomial p with complex coefficients such
4 that p(x) = 0 using Laguerre's method.
5
6 - horner_eval(p, x) evaluates the complex polynomial p of degree d (represented
7 as a vector of size d + 1 where p[i] stores the complex coefficient for the
8  $x^i$  term) at x, using Horner's method, returning a pair where the first value
9 is a vector of sub-evaluations and the second value is the final result p(x).
10 - find_one_root(p, x0) returns a complex root x for a polynomial p (represented
11 as a vector of size d + 1 where p[i] stores the complex coefficient for the
12  $x^i$  term) using an initial guess x0 which should be relatively close to x. The
13 root is found to a tolerance of EPS in absolute or relative error (whichever
14 is reached first).
15 - find_all_roots(p) returns a vector of all complex roots for a complex
16 polynomial p. The roots are found to a tolerance of EPS in absolute or
17 relative error (whichever is reached first).
18
19 Time Complexity:
20 -  $O(n)$  per call to horner_eval(), where n is the degree of the polynomial.
21 -  $O(n \log p)$  per call to find_one_root(), where n is the degree of the
22 polynomial and p =  $-\log_{10}(\text{EPS})$  is the number of digits of absolute or relative
23 precision that is desired.
24 -  $O(n^2 \log p)$  per call to find_all_roots(), where n is the degree of the
25 polynomial and p =  $-\log_{10}(\text{EPS})$  is the number of digits of absolute or relative
26 precision that is desired.
27
28 Space Complexity:
29 -  $O(n)$  auxiliary heap space and  $O(1)$  auxiliary stack space for horner_eval() and
30 find_one_root(), where n is the degree of the polynomial.
31 -  $O(n)$  auxiliary heap and  $O(1)$  auxiliary stack space per for find_one_root() and
32 find_all_roots(), where n is the degree of the polynomial.
33
34 */
35
36 #include <complex>
37 #include <cstdlib>
38 #include <vector>
39
40 typedef std::complex<double> cdouble;
41 typedef std::vector<cdouble> cpoly;
42
43 std::pair<cdouble, cpoly> horner_eval(const cpoly &p, const cdouble &x) {
44     int n = p.size();
45     cpoly b(std::max(1, n - 1));
46     for (int i = n - 1; i > 0; i--) {
47         b[i - 1] = p[i] + (i < n - 1 ? b[i]*x : 0);
48     }
49     return std::make_pair(p[0] + b[0]*x, b);
50 }
51
52 cpoly derivative(const cpoly &p) {
53     int n = p.size();
54     cpoly res(std::max(1, n - 1));
55     for (int i = 1; i < n; i++) {

```

```

56     res[i - 1] = p[i]*cdouble(i);
57 }
58 return res;
59 }
60
61 int comp(const cdouble &a, const cdouble &b, const double EPS = 1e-15) {
62     double diff = std::abs(a) - std::abs(b);
63     return (diff < -EPS) ? -1 : (diff > EPS ? 1 : 0);
64 }
65
66 cdouble find_one_root(const cpoly &p, const cdouble &x0,
67                       const double EPS = 1e-15, const int ITERATIONS = 10000) {
68     cdouble x = x0;
69     int n = p.size() - 1;
70     cpoly p1 = derivative(p), p2 = derivative(p1);
71     for (int i = 0; i < ITERATIONS; i++) {
72         cdouble y0 = horner_eval(p, x).first;
73         if (comp(y0, 0, EPS) == 0) {
74             break;
75         }
76         cdouble g = horner_eval(p1, x).first/y0;
77         cdouble h = g*g - horner_eval(p2, x).first/y0;
78         cdouble r = std::sqrt(cdouble(n - 1)*(h*cdouble(n) - g*g));
79         cdouble d1 = g + r, d2 = g - r;
80         cdouble a = cdouble(n)/(comp(d1, d2, EPS) > 0 ? d1 : d2);
81         x -= a;
82         if (comp(a, 0, EPS) == 0) {
83             break;
84         }
85     }
86     return x;
87 }
88
89 std::vector<cdouble> find_all_roots(const cpoly &p, const double EPS = 1e-15,
90                                       const int ITERATIONS = 10000) {
91     std::vector<cdouble> res;
92     cpoly q = p;
93     while (q.size() > 2) {
94         cdouble z = cdouble(rand(), rand())/(double)RAND_MAX;
95         z = find_one_root(p, find_one_root(q, z, EPS, ITERATIONS), EPS, ITERATIONS);
96         q = horner_eval(q, z).second;
97         res.push_back(z);
98     }
99     res.push_back(-q[0] / q[1]);
100    return res;
101 }
102
103 /** Example Usage and Output:
104
105 Roots of 140 - 13x - 8x^2 + x^3:
106 (5.00000, 0.00000)
107 (-4.00000, -0.00000)
108 (7.00000, -0.00000)
109 Roots of ((2 + 3i)x + 6)(x + i)(2x + (6 + 4i))(xi + 1):
110 (0.00000, 1.00000)
111 (0.00000, -1.00000)
112 (-0.92308, 1.38462)
113 (-3.00000, -2.00000)
114

```

```

115 ***/
116
117 #include <iostream>
118 #include <iomanip>
119 using namespace std;
120
121 void print_roots(const vector<cdouble> &x) {
122     for (int i = 0; i < (int)x.size(); i++) {
123         printf("(%.5lf, %.5lf)\n", x[i].real(), x[i].imag());
124     }
125 }
126
127 int main() {
128     { // 140 - 13x - 8x^2 + x^3 = (x + 4)(x - 5)(x - 7)
129         printf("Roots of 140 - 13x - 8x^2 + x^3:\n");
130         cpoly p;
131         p.push_back(140);
132         p.push_back(-13);
133         p.push_back(-8);
134         p.push_back(1);
135         print_roots(find_all_roots(p));
136     }
137     { // (-24+36i) + (-26+12i)x + (-30+40i)x^2 + (-26+12i)x^3 + (-6+4i)x^4
138       // = ((2 + 3i)x + 6)(x + i)(2x + (6 + 4i))(xi + 1):
139         printf("Roots of ((2 + 3i)x + 6)(x + i)(2x + (6 + 4i))(xi + 1):\n");
140         cpoly p;
141         p.push_back(cdouble(-24, 36));
142         p.push_back(cdouble(-26, 12));
143         p.push_back(cdouble(-30, 40));
144         p.push_back(cdouble(-26, 12));
145         p.push_back(cdouble(-6, 4));
146         print_roots(find_all_roots(p));
147     }
148     return 0;
149 }
```

5.6.5 Polynomial Root Finding (RPOLY)

```

1 /*
2
3 Finds every complex root x for a polynomial p with real coefficients such that
4 p(x) = 0 using a variant of the Jenkins-Traub algorithm known as RPOLY. This
5 implementation is adapted from TOMS493 (www.netlib.org/toms/) with a simple
6 wrapper for the C++ <complex> class.
7
8 - find_all_roots(p) returns a vector of all complex roots for a polynomial p
9 with real coefficients.
10
11 Time Complexity:
12 - O(n) per call to find_all_roots(p), where n is the degree of the polynomial.
13
14 Space Complexity:
15 - O(n) auxiliary stack space, where n is the degree of the polynomial.
16
17 */
```

```

18
19 #include <cfloat>
20 #include <cmath>
21
22 typedef long double LD;
23 const int MAXN = 105;
24
25 void divide_quadratic(int n, LD u, LD v, LD p[], LD q[], LD *a, LD *b) {
26     q[0] = *b = p[0];
27     q[1] = *a = -((*b)*u) + p[1];
28     for (int i = 2; i < n; i++) {
29         q[i] = -((*a)*u + (*b)*v) + p[i];
30         *b = *a;
31         *a = q[i];
32     }
33 }
34
35 int get_flag(int n, LD a, LD b, LD *a1, LD *a3, LD *a7, LD *c, LD *d, LD *e,
36                 LD *f, LD *g, LD *h, LD k[], LD u, LD v, LD qk[]) {
37     divide_quadratic(n, u, v, k, qk, c, d);
38     if (fabsl(*c) <= 100.0*LDBL_EPSILON*fabsl(k[n - 1]) &&
39         fabsl(*d) <= 100.0*LDBL_EPSILON*fabsl(k[n - 2])) {
40         return 3;
41     }
42     *h = v*b;
43     if (fabsl(*d) >= fabsl(*c)) {
44         *e = a/(*d);
45         *f = (*c)/(*d);
46         *g = u*b;
47         *a1 = (*f) * b - a;
48         *a3 = (*e) * ((*g) + a) + (*h)*(b/(*d));
49         *a7 = (*h) + ((*f) + u) * a;
50         return 2;
51     }
52     *e = a/(*c);
53     *f = (*d)/(*c);
54     *g = (*e)*u;
55     *a1 = -(a*((*d) / (*c))) + b;
56     *a3 = (*e)*a + ((*g) + (*h)/(*c))*b;
57     *a7 = (*g)*(*d) + (*h)*(*f) + a;
58     return 1;
59 }
60
61 void find_polynomials(int n, int flag, LD a, LD b, LD a1, LD *a3, LD *a7,
62                         LD k[], LD qk[], LD qp[]) {
63     if (flag == 3) {
64         k[1] = k[0] = 0.0;
65         for (int i = 2; i < n; i++) {
66             k[i] = qk[i - 2];
67         }
68         return;
69     }
70     if (fabsl(a1) > 10.0*LDBL_EPSILON*fabsl(flag == 1 ? b : a)) {
71         *a7 /= a1;
72         *a3 /= a1;
73         k[0] = qp[0];
74         k[1] = qp[1] - (*a7)*qp[0];
75         for (int i = 2; i < n; i++) {
76             k[i] = qp[i] - ((*a7)*qp[i - 1]) + (*a3)*qk[i - 2];

```

```

77      }
78  } else {
79     k[0] = 0.0;
80     k[1] = -(*a7)*qp[0];
81     for (int i = 2; i < n; i++) {
82       k[i] = (*a3)*qk[i - 2] - (*a7)*qp[i - 1];
83     }
84   }
85 }
86
87 void estimate_coeff(int flag, LD *uu, LD *vv, LD a, LD a1, LD a3, LD a7, LD b,
88                      LD c, LD d, LD f, LD g, LD h, LD u, LD v, LD k[], int n,
89                      LD p[]) {
90   LD a4, a5, b1, b2, c1, c2, c3, c4, temp;
91   *vv = *uu = 0.0;
92   if (flag == 3) {
93     return;
94   }
95   if (flag != 2) {
96     a4 = a + u*b + h*f;
97     a5 = c + (u + v*f)*d;
98   } else {
99     a4 = (a + g)*f + h;
100    a5 = (f + u)*c + v*d;
101  }
102  b1 = -k[n - 1] / p[n];
103  b2 = -(k[n - 2] + b1*p[n - 1]) / p[n];
104  c1 = v*b2*a1;
105  c2 = b1*a7;
106  c3 = b1*b1*a3;
107  c4 = c1 - c2 - c3;
108  temp = b1*a4 - c4 + a5;
109  if (temp != 0.0) {
110    *uu = u - (u*(c3 + c2) + v*(b1*a1 + b2*a7)) / temp;
111    *vv = v*(1.0 + c4/temp);
112  }
113 }
114
115 void solve_quadratic(LD a, LD b1, LD c, LD *sr, LD *si, LD *lr, LD *li) {
116   LD b, d, e;
117   *sr = *si = *lr = *li = 0.0;
118   if (a == 0) {
119     *sr = (b1 != 0) ? -c / b1 : *sr;
120     return;
121   }
122   if (c == 0) {
123     *lr = -b1 / a;
124     return;
125   }
126   b = b1 / 2.0;
127   if (fabsl(b) < fabsl(c)) {
128     e = (c >= 0) ? a : -a;
129     e = b*(b / fabsl(c)) - e;
130     d = sqrtl(fabsl(e))*sqrtl(fabsl(c));
131   } else {
132     e = 1.0 - (a / b)*(c / b);
133     d = sqrtl(fabsl(e))*fabsl(b);
134   }
135   if (e >= 0) {

```

```

136     d = (b >= 0) ? -d : d;
137     *lr = (d - b) / a;
138     *sr = (*lr != 0) ? (c / *lr / a) : *sr;
139 } else {
140     *lr = *sr = -b / a;
141     *si = fabsl(d / a);
142     *li = -(*si);
143 }
144 }
145
146 void quadratic_iterate(int N, int * NZ, LD uu, LD vv, LD *szr, LD *szi, LD *lzs,
147                         LD *lzi, LD qp[], int n, LD *a, LD *b, LD p[], LD qk[],
148                         LD *a1, LD *a3, LD *a7, LD *c, LD *d, LD *e, LD *f,
149                         LD *g, LD *h, LD k[]) {
150     int steps = 0, flag, tried_flag = 0;
151     LD ee, mp, relstp = 0.0, t, u, ui, v, vi, zm;
152     *NZ = 0;
153     u = uu;
154     v = vv;
155     do {
156         solve_quadratic(1.0, u, v, szr, szi, lzs, lzi);
157         if (fabsl(fabsl(*szr) - fabsl(*lzs)) > 0.01*fabsl(*lzs)) {
158             break;
159         }
160         divide_quadratic(n, u, v, p, qp, a, b);
161         mp = fabsl(-((*szr)*(*b)) + *a) + fabsl((*szi)*(*b));
162         zm = sqrtl(fabsl(v));
163         ee = 2.0*fabsl(qp[0]);
164         t = -(*szr)*(*b);
165         for (int i = 1; i < N; i++) {
166             ee = ee*zm + fabsl(qp[i]);
167         }
168         ee = ee*zm + fabsl(*a + t);
169         ee = ee*9.0 + 2.0*fabsl(t) - 7.0*(fabsl(*a + t) + zm*fabsl(*b));
170         ee *= LDBL_EPSILON;
171         if (mp <= 20.0*ee) {
172             *NZ = 2;
173             break;
174         }
175         if (++steps > 20) {
176             break;
177         }
178         if (steps >= 2 && relstp <= 0.01 && mp >= omp && !tried_flag) {
179             relstp = (relstp < LDBL_EPSILON) ? sqrtl(LDBL_EPSILON) : sqrtl(relstp);
180             u -= u*relstp;
181             v += v*relstp;
182             divide_quadratic(n, u, v, p, qp, a, b);
183             for (int i = 0; i < 5; i++) {
184                 flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
185                 find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
186             }
187             tried_flag = 1;
188             steps = 0;
189         }
190        omp = mp;
191         flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
192         find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
193         flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
194         estimate_coeff(flag, &ui, &vi, *a, *a1, *a3, *a7, *b, *c, *d, *f, *g, *h, u,

```

```

195           v, k, N, p);
196     if (vi != 0) {
197       relstp = fabsl((-v + vi)/vi);
198       u = ui;
199       v = vi;
200     }
201   } while (vi != 0);
202 }
203
204 void real_iterate(int *flag, int *nz, LD *sss, int n, LD p[], int nn, LD qp[],
205                   LD *szr, LD *szi, LD k[], LD qk[]) {
206   int steps = 0;
207   LD ee, kv, mp, ms, omp = 0.0, pv, s, t = 0.0;
208   *flag = *nz = 0;
209   for (s = *sss; ; s += t) {
210     pv = p[0];
211     qp[0] = pv;
212     for (int i = 1; i < nn; i++) {
213       qp[i] = pv = pv * s + p[i];
214     }
215     mp = fabsl(pv);
216     ms = fabsl(s);
217     ee = 0.5*fabsl(qp[0]);
218     for (int i = 1; i < nn; i++) {
219       ee = ee*ms + fabsl(qp[i]);
220     }
221     if (mp <= 20.0*LDBL_EPSILON*(2.0*ee - mp)) {
222       *nz = 1;
223       *szr = s;
224       *szi = 0.0;
225       break;
226     }
227     if (++steps > 10) {
228       break;
229     }
230     if (steps >= 2 && fabsl(t) <= 0.001*fabsl(s - t) && mp > omp) {
231       *flag = 1;
232       *sss = s;
233       break;
234     }
235     omp = mp;
236     qk[0] = kv = k[0];
237     for (int i = 1; i < n; i++) {
238       qk[i] = kv = kv*s + k[i];
239     }
240     if (fabsl(kv) > fabsl(k[n - 1])*10.0*LDBL_EPSILON) {
241       t = -pv / kv;
242       k[0] = qp[0];
243       for (int i = 1; i < n; i++) {
244         k[i] = t*qk[i - 1] + qp[i];
245       }
246     } else {
247       k[0] = 0.0;
248       for (int i = 1; i < n; i++) {
249         k[i] = qk[i - 1];
250       }
251     }
252     kv = k[0];
253     for (int i = 1; i < n; i++) {

```

```

254     kv = kv*s + k[i];
255 }
256 t = (fabsl(k[n - 1])*10.0*LDBL_EPSILON < fabsl(kv)) ? (-pv / kv) : 0.0;
257 }
258 }
259
260 void solve_fixedshift(int l2, int *nz, LD sr, LD v, LD k[], int n, LD p[],
261                         int nn, LD qp[], LD u, LD qk[], LD svk[], LD *lzi,
262                         LD *lzs, LD *szr) {
263     int flag, _flag, __flag = 1, spass, stry, vpass, vtry;
264     LD a, a1, a3, a7, b, betas, betav, c, d, e, f, g, h;
265     LD oss, ots = 0.0, otv = 0.0, ovv, s, ss, ts, tss, tv, tvv, ui, vi, vv;
266     *nz = 0;
267     betav = betas = 0.25;
268     oss = sr;
269     ovv = v;
270     divide_quadratic(nn, u, v, p, qp, &a, &b);
271     flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h, k, u, v, qk);
272     for (int j = 0; j < l2; j++) {
273         _flag = 1;
274         find_polynomials(n, flag, a, b, a1, &a3, &a7, k, qk, qp);
275         flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h, k, u, v,
276                         qk);
277         estimate_coeff(flag, &ui, &vi, a, a1, a3, a7, b, c, d, f, g, h, u, v, k, n,
278                         p);
279         vv = vi;
280         ss = k[n - 1] != 0.0 ? (-p[n] / k[n - 1]) : 0.0;
281         ts = tv = 1.0;
282         if (j != 0 && flag != 3) {
283             tv = (vv != 0.0) ? fabsl((vv - ovv) / vv) : tv;
284             ts = (ss != 0.0) ? fabsl((ss - oss) / ss) : ts;
285             tvv = (tv < otv) ? tv*otv : 1.0;
286             tss = (ts < ots) ? ts*ots : 1.0;
287             vpass = (tvv < betav) ? 1 : 0;
288             spass = (tss < betas) ? 1 : 0;
289             if (spass || vpass) {
290                 for (int i = 0; i < n; i++) {
291                     svk[i] = k[i];
292                 }
293                 s = ss; stry = vtry = 0;
294                 for (;;) {
295                     if (!(_flag && spass && (!vpass || tss < tvv))) {
296                         quadratic_iterate(n, nz, ui, vi, szr, szi, lzs, lzi, qp, nn, &a, &b,
297                                         p, qk, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h, k);
298                         if (*nz > 0) return;
299                         __flag = vtry = 1;
300                         betav *= 0.25;
301                         if (stry || !spass) {
302                             __flag = 0;
303                         } else {
304                             for (int i = 0; i < n; i++) {
305                                 k[i] = svk[i];
306                             }
307                         }
308                     }
309                     _flag = 0;
310                     if (__flag != 0) {
311                         real_iterate(&__flag, nz, &s, n, p, nn, qp, szr, szi, k, qk);
312                         if (*nz > 0) {

```

```

313         return;
314     }
315     stry = 1;
316     betas *= 0.25;
317     if (_flag != 0) {
318         ui = -(s + s);
319         vi = s * s;
320         continue;
321     }
322 }
323 for (int i = 0; i < n; i++) k[i] = svk[i];
324 if (!vpass || vtry) {
325     break;
326 }
327 }
328 divide_quadratic(nn, u, v, p, qp, &a, &b);
329 flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h, k, u, v,
330                 qk);
331 }
332 }
333 ovv = vv;
334 oss = ss;
335 otv = tv;
336 ots = ts;
337 }
338 }
339
340 void find_all_roots(int degree, LD co[], LD re[], LD im[]) {
341     int j, jj, n, nm1, nn, nz, zero;
342     LD k[MAXN], p[MAXN], pt[MAXN], qp[MAXN], temp[MAXN], qk[MAXN], svk[MAXN];
343     LD bnd, df, dx, factor, ff, moduli_max, moduli_min, sc, x, xm;
344     LD aa, bb, cc, lzi, lsr, sr, sz, szr, t, u, xx, xxxx, yy;
345     n = degree;
346     xx = sqrtl(0.5);
347     yy = -xx;
348     for (j = 0; co[n] == 0; n--, j++) {
349         re[j] = im[j] = 0.0;
350     }
351     nn = n + 1;
352     for (int i = 0; i < nn; i++) p[i] = co[i];
353     while (n >= 1) {
354         if (n <= 2) {
355             if (n < 2) {
356                 re[degree - 1] = -p[1] / p[0];
357                 im[degree - 1] = 0.0;
358             } else {
359                 solve_quadratic(p[0], p[1], p[2], &re[degree - 2], &im[degree - 2],
360                                 &re[degree - 1], &im[degree - 1]);
361             }
362             break;
363         }
364         moduli_max = 0.0;
365         moduli_min = LDBL_MAX;
366         for (int i = 0; i < nn; i++) {
367             x = fabsl(p[i]);
368             if (x > moduli_max) {
369                 moduli_max = x;
370             }
371             if (x != 0 && x < moduli_min) {

```

```

372     moduli_min = x;
373 }
374 }
375 sc = LDBL_MIN / LDBL_EPSILON / moduli_min;
376 if ((sc < 2 && moduli_max >= 10) || (sc > 1 && LDBL_MAX/sc >= moduli_max)) {
377     sc = (sc == 0) ? LDBL_MIN : sc;
378     factor = powl(2.0, logl(sc) / logl(2.0));
379     if (factor != 1.0) {
380         for (int i = 0; i < nn; i++) {
381             p[i] *= factor;
382         }
383     }
384 }
385 for (int i = 0; i < nn; i++) {
386     pt[i] = fabsl(p[i]);
387 }
388 pt[n] = -pt[n];
389 nm1 = n - 1;
390 x = expl((logl(-pt[n]) - logl(pt[0])) / (LD)n);
391 if (pt[nm1] != 0) {
392     xm = -pt[n] / pt[nm1];
393     if (xm < x) {
394         x = xm;
395     }
396 }
397 xm = x;
398 do {
399     x = xm;
400     xm = 0.1 * x;
401     ff = pt[0];
402     for (int i = 1; i < nn; i++) {
403         ff = ff*xm + pt[i];
404     }
405 } while (ff > 0);
406 dx = x;
407 do {
408     df = ff = pt[0];
409     for (int i = 1; i < n; i++) {
410         ff = x*ff + pt[i];
411         df = x*df + ff;
412     }
413     ff = x*ff + pt[n];
414     dx = ff / df;
415     x -= dx;
416 } while (fabsl(dx / x) > 0.005);
417 bnd = x;
418 for (int i = 1; i < n; i++) {
419     k[i] = (LD)(n - i)*p[i] / (LD)n;
420 }
421 k[0] = p[0];
422 aa = p[n];
423 bb = p[nm1];
424 zero = (k[nm1] == 0) ? 1 : 0;
425 for (jj = 0; jj < 5; jj++) {
426     cc = k[nm1];
427     if (zero) {
428         for (int i = 0; i < nm1; i++) {
429             j = nm1 - i;
430             k[j] = k[j - 1];

```

```

431     }
432     k[0] = 0;
433     zero = (k[nm1] == 0) ? 1 : 0;
434 } else {
435     t = -aa / cc;
436     for (int i = 0; i < nm1; i++) {
437         j = nm1 - i;
438         k[j] = t*k[j - 1] + p[j];
439     }
440     k[0] = p[0];
441     zero = (fabsl(k[nm1]) <= fabsl(bb)*LDBL_EPSILON*10.0) ? 1 : 0;
442 }
443 }
444 for (int i = 0; i < n; i++) {
445     temp[i] = k[i];
446 }
447 static const LD DEG = 0.01745329251994329576923690768489L;
448 for (jj = 1; jj <= 20; jj++) {
449     xxx = -sinl(94*DEG)*yy + cosl(94*DEG)*xx;
450     yy = sinl(94*DEG)*xxx + cosl(94*DEG)*yy;
451     xx = xxx;
452     sr = bnd*xxx;
453     u = -2.0*sr;
454     for (int i = 0; i < nn; i++) {
455         qk[i] = svk[i] = 0.0;
456     }
457     solve_fixedshift(20*jj, &nz, sr, bnd, k, n, p, nn, qp, u, qk, svk, &lzi,
458                      &lzs, &szi, &szs);
459     if (nz != 0) {
460         j = degree - n;
461         re[j] = szs;
462         im[j] = szi;
463         nn = nn - nz;
464         n = nn - 1;
465         for (int i = 0; i < nn; i++) {
466             p[i] = qp[i];
467         }
468         if (nz != 1) {
469             re[j + 1] = lzs;
470             im[j + 1] = lzi;
471         }
472         break;
473     } else {
474         for (int i = 0; i < n; i++) {
475             k[i] = temp[i];
476         }
477     }
478 }
479 if (jj > 20) {
480     break;
481 }
482 }
483 }
484 /**
485 *** Example Usage and Output:
486 Roots of -1+2x^1-6x^2+2x^3:
487 (0.150976, 0.403144)
488 (0.150976, -0.403144)

```

```

490 (2.69805, 0)
491 Roots of -20+4x^1+3x^2:
492 (2, 0)
493 (-3.33333, 0)
494 */
495
496
497 #include <cassert>
498 #include <iostream>
499 #include <complex>
500 #include <vector>
501 using namespace std;
502
503 typedef std::complex<LD> cdouble;
504
505 vector<cdouble> find_all_roots(const vector<LD> &p) {
506     int degree = p.size() - 1;
507     LD c[MAXN], re[MAXN], im[MAXN];
508     copy(p.rbegin(), p.rend(), c);
509     find_all_roots(degree, c, re, im);
510     vector<cdouble> res;
511     for (int i = 0; i < (int)p.size() - 1; i++) {
512         res.push_back(complex<LD>(re[i], im[i]));
513     }
514     return res;
515 }
516
517 cdouble eval(const vector<LD> &p, cdouble x) {
518     cdouble res = p.back();
519     for (int i = p.size() - 2; i >= 0; i--) {
520         res = res*x + p[i];
521     }
522     return res;
523 }
524
525 void print_roots(const vector<LD> &p, const vector<cdouble> &x) {
526     cout << "Roots of ";
527     for (int i = 0; i < (int)p.size(); i++) {
528         cout << showpos << (double)p[i];
529         if (i > 0) {
530             cout << noshowpos << "x^" << i;
531         }
532     }
533     cout << ":" << endl;
534     for (int i = 0; i < (int)x.size(); i++) {
535         cout << "(" << (double)x[i].real() << ", " << (double)x[i].imag() << ")\\n";
536     }
537 }
538
539 int main() {
540     { // -1 + 2x - 6x^2 + 2x^3
541         int poly[] = {-1, 2, -6, 2};
542         vector<LD> p(poly, poly + 4);
543         print_roots(p, find_all_roots(p));
544     }
545     { // -20 + 4x + 3x^2
546         int poly[] = {-20, 4, 3};
547         vector<LD> p(poly, poly + 3);
548         print_roots(p, find_all_roots(p));

```

```

549     }
550     return 0;
551 }
```

5.6.6 Integration (Simpson)

```

1  /*
2
3 Computes the definite integral from a to b for a continuous function f using
4 Simpson's approximation: integral ~ [f(a) + 4*f((a + b)/2) + f(b)]*(b - a)/6.
5
6 - simpsons(f, a, b) returns the definite integral for a function f from a to b,
7 to a tolerance of EPS in absolute error.
8
9 Time Complexity:
10 - O(p) per call to integrate(), where p = -log10(EPS) is the number of digits
11 of absolute precision that is desired.
12
13 Space Complexity:
14 - O(p) auxiliary stack and O(1) auxiliary heap space, where p = -log10(EPS)
15 is the number of digits of absolute precision that is desired.
16
17 */
18
19 #include <cmath>
20
21 template<class ContinuousFunction>
22 double simpsons(ContinuousFunction f, double a, double b) {
23     return (f(a) + 4*f((a + b)/2) + f(b))*(b - a)/6;
24 }
25
26 template<class ContinuousFunction>
27 double integrate(ContinuousFunction f, double a, double b,
28                  const double EPS = 1e-15) {
29     double m = (a + b) / 2;
30     double am = simpsons(f, a, m);
31     double mb = simpsons(f, m, b);
32     double ab = simpsons(f, a, b);
33     if (fabs(am + mb - ab) < EPS) {
34         return ab;
35     }
36     return integrate(f, a, m) + integrate(f, m, b);
37 }
38
39 /** Example Usage ***/
40
41 #include <cstdio>
42 #include <cassert>
43 using namespace std;
44
45 double f(double x) {
46     return sin(x);
47 }
48
49 int main () {
```

```
50     double PI = acos(-1.0);
51     assert(fabs(integrate(f, 0.0, PI/2) - 1) < 1e-10);
52     return 0;
53 }
```

Chapter 6

Geometry

6.1 Geometry Library in One File

```
1  /*
2
3 This combines the next few subsections in this chapter in a cross-dependent
4 manner. All of these algorithms apply to a two-dimensional Cartesian plane.
5
6 Time Complexity:
7 - O(1) for all operations.
8
9 Space Complexity:
10 - O(1) for storage of all data types.
11 - O(1) auxiliary for all operations.
12
13 */
14
15 #include <algorithm>
16 #include <cmath>
17 #include <cstddef>
18 #include <limits>
19 #include <iostream>
20 #include <stdexcept>
21 #include <utility>
22 #include <vector>
23
24 const double M_NAN = std::numeric_limits<double>::quiet_NaN();
25 const double EPS = 1e-9;
26
27 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
28 #define NE(a, b) (fabs((a) - (b)) > EPS)
29 #define LT(a, b) ((a) < (b) - EPS)
30 #define GT(a, b) ((a) > (b) + EPS)
31 #define LE(a, b) ((a) <= (b) + EPS)
32 #define GE(a, b) ((a) >= (b) - EPS)
33
34 // A two-dimensional point class that behaves like std::complex while supporting
35 // epsilon comparisons.
36 struct point {
```

```

37     double x, y;
38
39     point() : x(0), y(0) {}
40     point(double x, double y) : x(x), y(y) {}
41     point(const point &p) : x(p.x), y(p.y) {}
42     point(const std::pair<double, double> &p) : x(p.first), y(p.second) {}
43
44     bool operator<(const point &p) const {
45         return EQ(x, p.x) ? LT(y, p.y) : LT(x, p.x);
46     }
47
48     bool operator>(const point &p) const {
49         return EQ(x, p.x) ? LT(p.y, y) : LT(p.x, x);
50     }
51
52     bool operator==(const point &p) const { return EQ(x, p.x) && EQ(y, p.y); }
53     bool operator!=(const point &p) const { return !(*this == p); }
54     bool operator<=(const point &p) const { return !(*this > p); }
55     bool operator>=(const point &p) const { return !(*this < p); }
56     point operator+(const point &p) const { return point(x + p.x, y + p.y); }
57     point operator-(const point &p) const { return point(x - p.x, y - p.y); }
58     point operator+(double v) const { return point(x + v, y + v); }
59     point operator-(double v) const { return point(x - v, y - v); }
60     point operator*(double v) const { return point(x * v, y * v); }
61     point operator/(double v) const { return point(x / v, y / v); }
62     point& operator+=(const point &p) { x += p.x; y += p.y; return *this; }
63     point& operator-=(const point &p) { x -= p.x; y -= p.y; return *this; }
64     point& operator+=(double v) { x += v; y += v; return *this; }
65     point& operator-=(double v) { x -= v; y -= v; return *this; }
66     point& operator*=(double v) { x *= v; y *= v; return *this; }
67     point& operator/=(double v) { x /= v; y /= v; return *this; }
68     friend point operator+(double v, const point &p) { return p + v; }
69     friend point operator*(double v, const point &p) { return p * v; }
70
71     double sqnorm() const { return x*x + y*y; }
72     double norm() const { return sqrt(x*x + y*y); }
73     double arg() const { return atan2(y, x); }
74     double dot(const point &p) const { return x*p.x + y*p.y; }
75     double cross(const point &p) const { return x*p.y - y*p.x; }
76     double proj(const point &p) const { return dot(p) / p.norm(); }
77
78     // Returns a proportional unit vector (p, q) = c(x, y) where p^2 + q^2 = 1.
79     point normalize() const {
80         return (EQ(x, 0) && EQ(y, 0)) ? point(0, 0) : (point(x, y) / norm());
81     }
82
83     // Returns (x, y) rotated 90 degrees clockwise about the origin.
84     point rotate90() const { return point(-y, x); }
85
86     // Returns (x, y) rotated t radians clockwise about the origin.
87     point rotateCW(double t) const {
88         return point(x*cos(t) + y*sin(t), y*cos(t) - x*sin(t));
89     }
90
91     // Returns (x, y) rotated t radians counter-clockwise about the origin.
92     point rotateCCW(double t) const {
93         return point(x*cos(t) - y*sin(t), x*sin(t) + y*cos(t));
94     }
95

```

```
96 // Returns (x, y) rotated t radians clockwise about point p.
97 point rotateCW(const point &p, double t) const {
98     return (*this - p).rotateCW(t) + p;
99 }
100
101 // Returns (x, y) rotated t radians counter-clockwise about the point p.
102 point rotateCCW(const point &p, double t) const {
103     return (*this - p).rotateCCW(t) + p;
104 }
105
106 // Returns (x, y) reflected across point p.
107 point reflect(const point &p) const {
108     return point(2*p.x - x, 2*p.y - y);
109 }
110
111 // Returns (x, y) reflected across the line containing points p and q.
112 point reflect(const point &p, const point &q) const {
113     if (p == q) {
114         return reflect(p);
115     }
116     point r(*this - p), s = q - p;
117     r = point(r.x*s.x + r.y*s.y, r.x*s.y - r.y*s.x) / s.sqnorm();
118     r = point(r.x*s.x - r.y*s.y, r.x*s.y + r.y*s.x) + p;
119     return r;
120 }
121
122 friend double sqnorm(const point &p) { return p.sqnorm(); }
123 friend double norm(const point &p) { return p.norm(); }
124 friend double arg(const point &p) { return p.arg(); }
125 friend double dot(const point &p, const point &q) { return p.dot(q); }
126 friend double cross(const point &p, const point &q) { return p.cross(q); }
127 friend double proj(const point &p, const point &q) { return p.proj(q); }
128 friend point normalize(const point &p) { return p.normalize(); }
129 friend point rotate90(const point &p) { return p.rotate90(); }
130
131 friend point rotateCW(const point &p, double t) {
132     return p.rotateCW(t);
133 }
134
135 friend point rotateCCW(const point &p, double t) {
136     return p.rotateCCW(t);
137 }
138
139 friend point rotateCW(const point &p, const point &q, double t) {
140     return p.rotateCW(q, t);
141 }
142
143 friend point rotateCCW(const point &p, const point &q, double t) {
144     return p.rotateCCW(q, t);
145 }
146
147 friend point reflect(const point &p, const point &q) {
148     return p.reflect(q);
149 }
150
151 friend point reflect(const point &p, const point &a, const point &b) {
152     return p.reflect(a, b);
153 }
154
```

```

155     friend std::ostream& operator<<(std::ostream &out, const point &p) {
156         return out << "(" << (fabs(p.x) < EPS ? 0 : p.x) << ","
157                                         << (fabs(p.y) < EPS ? 0 : p.y) << ")";
158     }
159 };
160
161 // A two-dimensional line class stored of the form ax + by + c = 0, normalized
162 // such that b is always either 1 (for normal line) or 0 (for vertical lines).
163 struct line {
164     double a, b, c;
165
166     line() : a(0), b(0), c(0) {} // Invalid or uninitialized line.
167
168     line(double a, double b, double c) {
169         if (!EQ(b, 0)) {
170             this->a = a / b;
171             this->c = c / b;
172             this->b = 1;
173         } else {
174             this->c = c / a;
175             this->a = 1;
176             this->b = 0;
177         }
178     }
179
180     line(double slope, const point &p) {
181         a = -slope;
182         b = 1;
183         c = slope * p.x - p.y;
184     }
185
186     line(const point &p, const point &q) : a(0), b(0), c(0) {
187         if (EQ(p.x, q.x)) {
188             if (NE(p.y, q.y)) { // Vertical line.
189                 a = 1;
190                 b = 0;
191                 c = -p.x;
192             } // Else, invalid line.
193         } else {
194             a = -(p.y - q.y) / (p.x - q.x);
195             b = 1;
196             c = -(a*p.x) - (b*p.y);
197         }
198     }
199
200     bool operator==(const line &l) const {
201         return EQ(a, l.a) && EQ(b, l.b) && EQ(c, l.c);
202     }
203
204     bool operator!=(const line &l) const {
205         return !(*this == l);
206     }
207
208     // Returns whether the line is initialized and normalized.
209     bool valid() const {
210         if (EQ(a, 0)) {
211             return !EQ(b, 0);
212         }
213         return EQ(b, 1) || (EQ(b, 0) && EQ(a, 1));

```

```

214     }
215
216     bool horizontal() const { return valid() && EQ(a, 0); }
217     bool vertical() const { return valid() && EQ(b, 0); }
218     double slope() const { return (!valid() || EQ(b, 0)) ? M_NAN : -a; }
219
220     // Solve for x at a given y. If the line is horizontal, then either -INF, INF,
221     // or NAN is returned based on whether y is below, above, or on the line.
222     double x(double y) const {
223         if (!valid() || EQ(a, 0)) {
224             return M_NAN; // Invalid or horizontal line.
225         }
226         return (-c - b*y) / a;
227     }
228
229     // Solve for y at a given x. If the line is vertical, then either -INF, INF,
230     // or NAN is returned based on whether x is left of, right of, or on the line.
231     double y(double x) const {
232         if (!valid() || EQ(b, 0)) {
233             return M_NAN; // Invalid or vertical line.
234         }
235         return (-c - a*x) / b;
236     }
237
238     bool contains(const point &p) const { return EQ(a*p.x + b*p.y + c, 0); }
239     bool is_parallel(const line &l) const { return EQ(a, l.a) && EQ(b, l.b); }
240     bool is_perpendicular(const line &l) const { return EQ(-a*l.a, b*l.b); }
241
242     // Return the parallel line passing through point p.
243     line parallel(const point &p) const {
244         return line(a, b, -a*p.x - b*p.y);
245     }
246
247     // Return the perpendicular line passing through point p.
248     line perpendicular(const point &p) const {
249         return line(-b, a, b*p.x - a*p.y);
250     }
251
252     friend std::ostream& operator<<(std::ostream &out, const line &l) {
253         return out << (fabs(l.a) < EPS ? 0 : l.a) << "x" << std::showpos
254             << (fabs(l.b) < EPS ? 0 : l.b) << "y"
255             << (fabs(l.c) < EPS ? 0 : l.c) << "=0" << std::noshowpos;
256     }
257 };
258
259 const double PI = acos(-1.0), DEG = PI/180, RAD = 180/PI;
260
261 // Returns t degrees reduced to the range [0, 360). E.g. -630 becomes 90.
262 double reduce_deg(double t) {
263     if (t < -360) {
264         return reduce_deg(fmod(t, 360));
265     }
266     if (t < 0) {
267         return t + 360;
268     }
269     return (t >= 360) ? fmod(t, 360) : t;
270 }
271
272 // Returns t radians reduced to the range [0, 2*pi). E.g. 720.5 becomes 0.5.

```

```
273 double reduce_rad(double t) {
274     if (t < -2*PI) {
275         return reduce_rad(fmod(t, 2*PI));
276     }
277     if (t < 0) {
278         return t + 2*PI;
279     }
280     return (t >= 2*PI) ? fmod(t, 2*PI) : t;
281 }
282
283 // Returns a two-dimensional Cartesian point given radius r and angle t radians
284 // in polar coordinates, analogous to std::polar().
285 point polar_point(double r, double t) {
286     return point(r*cos(t), r*sin(t));
287 }
288
289 // Returns the angle in radians of the segment from (0, 0) to point p, relative
290 // counterclockwise to the positive x-axis.
291 double polar_angle(const point &p) {
292     double t = arg(p);
293     return (t < 0) ? (t + 2*PI) : t;
294 }
295
296 // Returns the smallest angle in radians formed by the points a, o, b with
297 // vertex at point o.
298 double angle(const point &a, const point &o, const point &b) {
299     point u(o - a), v(o - b);
300     return acos(u.dot(v) / (norm(u)*norm(v)));
301 }
302
303 // Returns the angle in radians of segment from point a to point b, relative
304 // counterclockwise to the positive x-axis.
305 double angle_between(const point &a, const point &b) {
306     double t = atan2(a.cross(b), a.dot(b)); // Equivalently, b.arg() - a.arg().
307     return (t < 0) ? (t + 2*PI) : t;
308 }
309
310 // Returns the smaller angle in radians between two lines, limited to [0, PI/2].
311 double angle_between(const line &l1, const line &l2) {
312     double t = atan2(l1.a*l2.b - l2.a*l1.b, l1.a*l2.a + l1.b*l2.b);
313     if (t < 0) {
314         t += PI;
315     }
316     return GT(t, PI / 2) ? (PI - t) : t;
317 }
318
319 // Returns the magnitude (Euclidean norm) of the three-dimensional cross product
320 // between points a and b where the z-component is implicitly zero and the
321 // origin is implicitly shifted to point o. This operation is also equal to
322 // double the signed area of the triangle from these three points.
323 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
324     return (a - o).cross(b - o);
325 }
326
327 // Returns -1 if the path a->o->b forms a left turn on the plane, 0 if the path
328 // forms a straight line segment (collinear), or 1 if it forms a right turn.
329 int turn(const point &a, const point &o, const point &b) {
330     double c = cross(a, b, o);
331     return LT(c, 0) ? -1 : (GT(c, 0) ? 1 : 0);
```

```

332 }
333
334 // Returns the distance and squared distance between points a and b.
335 double dist(const point &a, const point &b) { return norm(b - a); }
336 double sqdist(const point &a, const point &b) { return sqnorm(b - a); }
337
338 // Returns the distance from point p to line l. If l is invalid (l.a = l.b = 0),
339 // then -INF, INF, or NaN is returned based on the sign of l.c.
340 double line_dist(const point &p, const line &l) {
341     return fabs(l.a*p.x + l.b*p.y + l.c) / sqrt(l.a*l.a + l.b*l.b);
342 }
343
344 // Returns the distance from point p to the line (not segment) containing points
345 // a and b. If a = b, then the distance from p to the single point is returned.
346 double line_dist(const point &p, const point &a, const point &b) {
347     return (a == b) ? dist(p, a)
348                 : norm(a + (p - a).dot(b - a)*(b - a) / sqdist(a, b) - p);
349 }
350
351 // Returns the distance between two lines. If the lines are non-parallel then
352 // the distance is considered to be 0. Otherwise, the distance is considered to
353 // be the perpendicular distance from any point on one line to the other line.
354 double line_dist(const line &l1, const line &l2) {
355     if (EQ(l1.a*l1.b, l2.a*l2.b)) {
356         double factor = EQ(l1.b, 0) ? (l1.a / l2.a) : (l1.b / l2.b);
357         return EQ(l1.c, l2.c*factor) ? 0
358             : fabs(l2.c*factor - l1.c) / sqrt(l1.a*l1.a + l1.b*l1.b);
359     }
360     return 0;
361 }
362
363 // Returns the distance from point p to the line segment ab.
364 double seg_dist(const point &p, const point &a, const point &b) {
365     if (a == b) {
366         return dist(p, a);
367     }
368     point ab(b - a), ap(p - a);
369     double n = sqnorm(ab), d = ab.dot(ap);
370     if (LE(d, 0) || EQ(n, 0)) {
371         return norm(ap);
372     }
373     return GE(d, n) ? norm(ap - ab) : norm(ap - ab*(d / n));
374 }
375
376 // Determines whether lines l1 and l2 intersect. Returns -1 if there is no
377 // intersection because the lines are parallel, 0 if there is exactly one
378 // intersection (in which case the intersection point is stored into pointer p
379 // if it's not NULL), or 1 if there are infinite intersections because the lines
380 // are identical.
381 int line_intersection(const line &l1, const line &l2, point *p = NULL) {
382     if (l1.is_parallel(l2)) {
383         return (l1 == l2) ? 1 : -1;
384     }
385     if (p != NULL) {
386         p->x = (l1.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);
387         if (!EQ(l1.b, 0)) {
388             p->y = -(l1.a*p->x + l1.c) / l1.b;
389         } else {
390             p->y = -(l2.a*p->x + l2.c) / l2.b;

```

```

391     }
392 }
393 return 0;
394 }
395
396 // Determines whether the infinite lines (not segments) through points p1, p2
397 // and through points p3, p4 intersect. Returns -1 if there is no intersection
398 // because the lines are parallel, 0 if there is exactly one intersection (in
399 // which case the intersection point is stored into pointer p if it's not NULL),
400 // or 1 if there are infinite intersections because the lines are identical.
401 int line_intersection(const point &p1, const point &p2, const point &p3,
402                      const point &p4, point *p = NULL) {
403     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
404     double c1 = -(p1.x*p2.y - p2.x*p1.y);
405     double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
406     double c2 = -(p3.x*p4.y - p4.x*p3.y);
407     double x = -(c1*b2 - c2*b1), y = -(a1*c2 - a2*c1);
408     double det = a1*b2 - a2*b1;
409     if (EQ(det, 0)) {
410         return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
411     }
412     if (p != NULL) {
413         *p = point(x / det, y / det);
414     }
415     return 0;
416 }
417
418 // Determines whether the line segment ab intersects the line segment cd.
419 // Returns -1 if the segments do not intersect, 0 if there is exactly one
420 // intersection point (in which case it is stored into pointer p if it's not
421 // NULL), or 1 if the intersection is another line segment (in which case the
422 // two endpoints are stored into pointers p and q if they are not NULL). If the
423 // segments are barely touching (close within EPS), then the result will depend
424 // on the setting of TOUCH_IS_INTERSECT.
425 int seg_intersection(const point &a, const point &b, const point &c,
426                      const point &d, point *p = NULL, point *q = NULL) {
427     static const bool TOUCH_IS_INTERSECT = true;
428     point ab(b - a), ac(c - a), cd(d - c);
429     double c1 = ab.cross(cd), c2 = ac.cross(ab);
430     if (EQ(c1, 0) && EQ(c2, 0)) { // Collinear.
431         double t0 = ac.dot(ab) / sqnorm(ab);
432         double t1 = t0 + cd.dot(ab) / sqnorm(ab);
433         double mint = std::min(t0, t1), maxt = std::max(t0, t1);
434         bool overlap = TOUCH_IS_INTERSECT ? (LE(mint, 1) && LE(0, maxt))
435                                         : (LT(mint, 1) && LT(0, maxt));
436         if (overlap) {
437             point res1 = std::max(std::min(a, b), std::min(c, d));
438             point res2 = std::min(std::max(a, b), std::max(c, d));
439             if (res1 == res2) {
440                 if (p != NULL) {
441                     *p = res1;
442                 }
443                 return 0; // Collinear and meeting at an endpoint.
444             }
445             if (p != NULL && q != NULL) {
446                 *p = res1;
447                 *q = res2;
448             }
449             return 1; // Collinear and overlapping.

```

```

450     } else {
451         return -1; // Collinear and disjoint.
452     }
453 }
454 if (EQ(c1, 0)) {
455     return -1; // Parallel and disjoint.
456 }
457 double t = ac.cross(cd)/c1, u = c2/c1;
458 bool t_between_01 = TOUCH_IS_INTERSECT ? (LE(0, t) && LE(t, 1))
459                                         : (LT(0, t) && LT(t, 1));
460 bool u_between_01 = TOUCH_IS_INTERSECT ? (LE(0, u) && LE(u, 1))
461                                         : (LT(0, u) && LT(u, 1));
462 if (t_between_01 && u_between_01) {
463     if (p != NULL) {
464         *p = a + t*ab;
465     }
466     return 0; // Non-parallel with one intersection.
467 }
468 return -1; // Non-parallel with no intersections.
469 }
470
471 // Returns the minimum distance from any point on the line segment ab to any
472 // point on the line segment cd. This is 0 if the segments touch or intersect.
473 double seg_dist(const point &a, const point &b, const point &c,
474                  const point &d) {
475     return (seg_intersection(a, b, c, d) >= 0) ? 0
476           : std::min(std::min(seg_dist(a, c, d), seg_dist(b, c, d)),
477                      std::min(seg_dist(c, a, b), seg_dist(d, a, b)));
478 }
479
480 // Returns the point on line l that is closest to point p. Note that the result
481 // always lies on the line through p that is perpendicular to l.
482 point closest_point(const line &l, const point &p) {
483     if (EQ(l.a, 0)) {
484         return point(p.x, -l.c); // Horizontal line.
485     }
486     if (EQ(l.b, 0)) {
487         return point(-l.c, p.y); // Vertical line.
488     }
489     point res;
490     line_intersection(l, l.perpendicular(p), &res);
491     return res;
492 }
493
494 // Returns the point on line segment ab that is closest to point p.
495 point closest_point(const point &a, const point &b, const point &p) {
496     if (a == b) {
497         return a;
498     }
499     point ap(p - a), ab(b - a);
500     double t = ap.dot(ab) / sqnorm(ab);
501     return (t <= 0) ? a : ((t >= 1) ? b : point(a.x + t*ab.x, a.y + t*ab.y));
502 }
503
504 // A two-dimensional circle class represented by (x - h)^2 + (y - k)^2 = r^2,
505 // where (h, k) is the center and r is the radius.
506 struct circle {
507     double h, k, r;
508

```

```

509     circle() : h(0), k(0), r(0) {}
510     circle(double r) : h(0), k(0), r(fabs(r)) {}
511     circle(const point &o, double r) : h(o.x), k(o.y), r(fabs(r)) {}
512     circle(double h, double k, double r) : h(h), k(k), r(fabs(r)) {}
513
514 // Circle with the line segment ab as a diameter.
515 circle(const point &a, const point &b) {
516     h = (a.x + b.x)/2.0;
517     k = (a.y + b.y)/2.0;
518     r = norm(a - point(h, k));
519 }
520
521 // Circumcircle of three points.
522 circle(const point &a, const point &b, const point &c) {
523     double an = sqnorm(b - c), bn = sqnorm(a - c), cn = sqnorm(a - b);
524     double wa = an*(bn + cn - an);
525     double wb = bn*(an + cn - bn);
526     double wc = cn*(an + bn - cn);
527     double w = wa + wb + wc;
528     if (EQ(w, 0)) {
529         throw std::runtime_error("No circle from collinear points.");
530     }
531     h = (wa*a.x + wb*b.x + wc*c.x)/w;
532     k = (wa*a.y + wb*b.y + wc*c.y)/w;
533     r = norm(a - point(h, k));
534 }
535
536 // Circle of radius r that contains points a and b. In the general case, there
537 // will be two possible circles and only one is chosen arbitrarily. However if
538 // the diameter is equal to dist(a, b) = 2*r, then there is only one possible
539 // center. If points a and b are identical, then there are infinite circles.
540 // If the points are too far away relative to the radius, then there is no
541 // possible circle. In the latter two cases, an exception is thrown.
542 circle(const point &a, const point &b, double r) : r(fabs(r)) {
543     if (LE(r, 0) && a == b) { // Circle with zero area.
544         h = a.x;
545         k = a.y;
546         return;
547     }
548     double d = norm(b - a);
549     if (EQ(d, 0)) {
550         throw std::runtime_error("Identical points, infinite circles.");
551     }
552     if (LT(r*2.0, d)) {
553         throw std::runtime_error("Points too far away to make circle.");
554     }
555     double v = sqrt(r*r - d*d/4.0) / d;
556     point m = (a + b) / 2.0;
557     h = m.x + v*(a.y - b.y);
558     k = m.y + v*(b.x - a.x);
559     // The other answer is (h, k) = (m.x - v*(a.y - b.y), m.y - v*(b.x - a.x)).
560 }
561
562 bool operator==(const circle &c) const {
563     return EQ(h, c.h) && EQ(k, c.k) && EQ(r, c.r);
564 }
565
566 bool operator!=(const circle &c) const {
567     return !(*this == c);

```

```

568     }
569
570     point center() const { return point(h, k); }
571     bool contains(const point &p) const { return LE(sqnorm(p - center()), r*r); }
572     bool on_edge(const point &p) const { return EQ(sqnorm(p - center()), r*r); }
573
574     friend std::ostream& operator<<(std::ostream &out, const circle &c) {
575         return out << std::showpos << "(x" << -(fabs(c.h) < EPS ? 0 : c.h) << ")^2+"
576             << "(y" << -(fabs(c.k) < EPS ? 0 : c.k) << ")^2"
577             << std::noshowpos << "=" << (fabs(c.r) < EPS ? 0 : c.r*c.r);
578     }
579 };
580
581 // Returns the circle inscribed inside the triangle abc.
582 circle incircle(const point &a, const point &b, const point &c) {
583     double al = norm(b - c), bl = norm(a - c), cl = norm(a - b), p = al + bl + cl;
584     return EQ(p, 0) ? circle(a.x, a.y, 0)
585                     : circle((al*a.x + bl*b.x + cl*c.x) / p,
586                               (al*a.y + bl*b.y + cl*c.y) / p,
587                               fabs(cross(a, b, c)) / p);
588 }
589
590 // Determines the line(s) tangent to circle c that passes through point p.
591 // Returns -1 if there is no tangent line because p is strictly inside c, 0 if
592 // there is exactly one tangent line because p is on the boundary of c (in which
593 // case the line will be stored into pointer l1 if it's not NULL), or 1 if there
594 // are two tangent lines because p is strictly outside of c (in which case the
595 // lines will be stored into pointers l1 and l2 if they are not NULL).
596 int tangent(const circle &c, const point &p, line *l1 = NULL, line *l2 = NULL) {
597     if (c.on_edge(p)) {
598         if (l1 != NULL) {
599             *l1 = line(point(c.h, c.k), p).perpendicular(p);
600         }
601         return 0;
602     }
603     if (c.contains(p)) {
604         return -1;
605     }
606     point q = (p - c.center()) / c.r;
607     double n = sqnorm(q), d = q.y*sqrt(sqnorm(q) - 1.0);
608     point t1((q.x - d) / n, c.k), t2((q.x + d) / n, c.k);
609     if (NE(q.y, 0)) {
610         t1.y += c.r*(1.0 - t1.x*q.x) / q.y;
611         t2.y += c.r*(1.0 - t2.x*q.x) / q.y;
612     } else {
613         d = c.r*sqrt(1.0 - t1.x*t1.x);
614         t1.y += d;
615         t2.y -= d;
616     }
617     t1.x = t1.x*c.r + c.h;
618     t2.x = t2.x*c.r + c.h;
619     if (l1 != NULL && l2 != NULL) {
620         *l1 = line(p, t1);
621         *l2 = line(p, t2);
622     }
623     return 1;
624 }
625
626 // Determines the intersection between the circle c and line l. Returns -1 if

```

```

627 // there is no intersection, 0 if the line is one intersection point because the
628 // line is tangent (in which case it will be stored into pointer p if it's not
629 // NULL), or 1 if there are two intersection points because the line crosses
630 // through the circle (in which case they will be stored into pointers p and q
631 // if they are not NULL).
632 int intersection(const circle &c, const line &l,
633                 point *p = NULL, point *q = NULL) {
634     if (!l.valid()) {
635         throw std::runtime_error("Invalid line for intersection.");
636     }
637     double v = c.h*l.a + c.k*l.b + l.c;
638     double aabb = l.a*l.a + l.b*l.b;
639     double disc = v*v / aabb - c.r*c.r;
640     if (disc > EPS) {
641         return -1;
642     }
643     double x0 = -l.a*l.c / aabb, y0 = -l.b*v / aabb;
644     if (disc > -EPS) {
645         if (p != NULL) {
646             *p = point(x0 + c.h, y0 + c.k);
647         }
648         return 0;
649     }
650     double k = sqrt(std::max(0.0, disc / -aabb));
651     if (p != NULL && q != NULL) {
652         *p = point(x0 + k*l.b + c.h, y0 - k*l.a + c.k);
653         *q = point(x0 - k*l.b + c.h, y0 + k*l.a + c.k);
654     }
655     return 1;
656 }
657
658 // Determines the intersection points between two circles c1 and c2.
659 // Returns: -2 if circle c2 completely encloses circle c1,
660 //          -1 if circle c1 completely encloses circle c2,
661 //          0 if the circles are completely disjoint,
662 //          1 if the circles are tangent with one intersection (stored in p),
663 //          2 if the circles intersect at two points (stored in p and q),
664 //          3 if the circles are equal and intersect at infinite points.
665 int intersection(const circle &c1, const circle &c2, point *p = NULL,
666                   point *q = NULL) {
667     if (EQ(c1.h, c2.h) && EQ(c1.k, c2.k)) {
668         return EQ(c1.r, c2.r) ? 3 : (c1.r > c2.r ? -1 : -2);
669     }
670     point d12(c2.center() - c1.center());
671     double d = norm(d12);
672     if (GT(d, c1.r + c2.r)) {
673         return 0;
674     }
675     if (LT(d, fabs(c1.r - c2.r))) {
676         return c1.r > c2.r ? -1 : -2;
677     }
678     double a = (c1.r*c1.r - c2.r*c2.r + d*d) / (2*d);
679     double x0 = c1.h + (d12.x*a / d), y0 = c1.k + (d12.y*a / d);
680     double s = sqrt(c1.r*c1.r - a*a), rx = -d12.y*s / d, ry = d12.x*s / d;
681     if (EQ(rx, 0) && EQ(ry, 0)) {
682         if (p != NULL) {
683             *p = point(x0, y0);
684         }
685         return 1;

```

```

686     }
687     if (p != NULL && q != NULL) {
688         *p = point(x0 - rx, y0 - ry);
689         *q = point(x0 + rx, y0 + ry);
690     }
691     return 2;
692 }
693
694 // Returns the intersection area of circles c1 and c2.
695 double intersection_area(const circle &c1, const circle &c2) {
696     double r = std::min(c1.r, c2.r), R = std::max(c1.r, c2.r);
697     double d = norm(c2.center() - c1.center());
698     if (LE(d, R - r)) {
699         return PI*r*r;
700     }
701     if (GE(d, R + r)) {
702         return 0;
703     }
704     return r*r*acos((d*d + r*r - R*R) / 2 / d / r) +
705             R*R*acos((d*d + R*R - r*r) / 2 / d / R) -
706             0.5*sqrt((-d + r + R)*(d + r - R)*(d - r + R)*(d + r + R));
707 }
708
709 // Returns the area of the triangle abc.
710 double triangle_area(const point &a, const point &b, const point &c) {
711     return fabs(cross(a, b, c)) / 2.0;
712 }
713
714 // Returns the area of a triangle with side lengths s1, s2, and s3. The given
715 // lengths must be non-negative and form a valid triangle.
716 double triangle_area_sides(double s1, double s2, double s3) {
717     double s = (s1 + s2 + s3) / 2.0;
718     return sqrt(s*(s - s1)*(s - s2)*(s - s3));
719 }
720
721 // Returns the area of a triangle with medians of lengths m1, m2, and m3. The
722 // median of a triangle is a line segment joining a vertex to the midpoint of
723 // the opposing edge.
724 double triangle_area_medians(double m1, double m2, double m3) {
725     return 4.0*triangle_area_sides(m1, m2, m3) / 3.0;
726 }
727
728 // Returns the area of a triangle with altitudes h1, h2, and h3. An altitude of
729 // a triangle is the shortest line between a vertex and the infinite line that
730 // is extended from its opposite edge.
731 double triangle_area_altitudes(double h1, double h2, double h3) {
732     if (EQ(h1, 0) || EQ(h2, 0) || EQ(h3, 0)) {
733         return 0;
734     }
735     double x = h1*h1, y = h2*h2, z = h3*h3;
736     double v = 2.0/(x*y) + 2.0/(x*z) + 2.0/(y*z);
737     return 1.0/sqrt(v - 1.0/(x*x) - 1.0/(y*y) - 1.0/(z*z));
738 }
739
740 // Returns whether points p1 and p2 lie on the same side of the line containing
741 // points a and b. If one or both points lie exactly on the line, then the
742 // result will depend on the setting of EDGE_IS_SAME_SIDE.
743 bool same_side(const point &p1, const point &p2, const point &a,
744                 const point &b) {

```

```

745     static const bool EDGE_IS_SAME_SIDE = true;
746     point ab(b - a);
747     double c1 = ab.cross(p1 - a), c2 = ab.cross(p2 - a);
748     return EDGE_IS_SAME_SIDE ? GE(c1*c2, 0) : GT(c1*c2, 0);
749 }
750
751 // Returns whether point p lies within the triangle abc. If the point lies on or
752 // close to an edge (by roughly EPS), then the result will depend on the setting
753 // of EDGE_IS_SAME_SIDE in the function above.
754 bool point_in_triangle(const point &p, const point &a, const point &b,
755                         const point &c) {
756     return same_side(p, a, b, c) &&
757            same_side(p, b, a, c) &&
758            same_side(p, c, a, b);
759 }
760
761 // Returns the area of a rectangle with opposing vertices a and b.
762 double rectangle_area(const point &a, const point &b) {
763     return fabs((a.x - b.x)*(a.y - b.y));
764 }
765
766 // Returns whether point p lies within the rectangle defined by a vertex at v
767 // (x, y), a width of w, and a height of h. Note that negative widths and
768 // heights are supported. If the point lies on or close to an edge (by roughly
769 // EPS), then the result will depend on the setting of EDGE_IS_INSIDE.
770 bool point_in_rectangle(const point &p, const point &v, double w, double h) {
771     static const bool EDGE_IS_INSIDE = true;
772     if (w < 0) {
773         return point_in_rectangle(p, point(v.x + w, v.y), -w, h);
774     }
775     if (h < 0) {
776         return point_in_rectangle(p, point(v.x, v.y + h), w, -h);
777     }
778     return EDGE_IS_INSIDE
779         ? (GE(p.x, v.x) && LE(p.x, v.x + w) && GE(p.y, v.y) && LE(p.y, v.y + h))
780         : (GT(p.x, v.x) && LT(p.x, v.x + w) && GT(p.y, v.y) && LT(p.y, v.y + h));
781 }
782
783 // Returns whether point p lies within the rectangle with opposing vertices a
784 // and b. If the point lies on or close to an edge (by roughly EPS), then the
785 // result will depend on the setting of EDGE_IS_INSIDE in the function above.
786 bool point_in_rectangle(const point &p, const point &a, const point &b) {
787     double xl = std::min(a.x, b.x), yl = std::min(a.y, b.y);
788     double xh = std::max(a.x, b.x), yh = std::max(a.y, b.y);
789     return point_in_rectangle(p, point(xl, yl), xh - xl, yh - yl);
790 }
791
792 // Determines the intersection region of the rectangle with opposing vertices a1
793 // and b1 and the rectangle with opposing vertices a2 and b2. Returns -1 if the
794 // rectangles are completely disjoint, 0 if the rectangles partially intersect,
795 // 1 if the first rectangle is completely inside the second, and 2 if the second
796 // rectangle is completely inside the first. If there is an intersection, the
797 // opposing vertices of the intersection rectangle will be stored into pointers
798 // p and q if they are not NULL. If the intersection is a single point or line
799 // segment, then the result will depend on the setting of EDGE_IS_INSIDE in the
800 // point_in_rectangle function above.
801 int rectangle_intersection(const point &a1, const point &b1, const point &a2,
802                           const point &b2, point *p = NULL, point *q = NULL) {
803     bool a1in2 = point_in_rectangle(a1, a2, b2);

```

```

804     bool b1in2 = point_in_rectangle(b1, a2, b2);
805     if (a1in2 && b1in2) {
806         if (p != NULL && q != NULL) {
807             *p = std::min(a1, b1);
808             *q = std::max(a1, b1);
809         }
810         return 1; // Rectangle 1 completely inside 2.
811     }
812     if (!a1in2 && !b1in2) {
813         if (point_in_rectangle(a2, a1, b1)) {
814             if (p != NULL && q != NULL) {
815                 *p = std::min(a2, b2);
816                 *q = std::max(a2, b2);
817             }
818             return 2; // Rectangle 2 completely inside 1.
819         }
820         return -1; // Completely disjoint.
821     }
822     if (p != NULL && q != NULL) {
823         if (a1in2) {
824             *p = a1;
825             *q = (a1 < b1) ? std::max(a2, b2) : std::min(a2, b2);
826         } else {
827             *p = b1;
828             *q = (b1 < a1) ? std::max(a2, b2) : std::min(a2, b2);
829         }
830         if (*p > *q) {
831             std::swap(p, q);
832         }
833     }
834     return 0;
835 }
836
837 /** Example Usage ***/
838
839 #include <cassert>
840 #include <iostream>
841 using namespace std;
842
843 #define pt point
844 #define EQP(p, q) (EQ((p).x, (q).x) && EQ((p).y, (q).y))
845
846 int main() {
847     pt p, q;
848
849     p = point(-10, 3);
850     assert(pt(-18, 29) == p + pt(-3, 9)*6 / 2 - pt(-1, 1));
851     assert(EQ(109, p.sqnorm()));
852     assert(EQ(10.44030650891, p.norm()));
853     assert(EQ(2.850135859112, p.arg()));
854     assert(EQ(0, p.dot(pt(3, 10))));
855     assert(EQ(0, p.cross(pt(10, -3))));
856     assert(EQ(10, p.proj(pt(-10, 0))));
857     assert(EQ(1, p.normalize().norm()));
858     assert(pt(-3, -10) == p.rotate90());
859     assert(pt(3, 12) == p.rotateCW(pt(1, 1), PI / 2));
860     assert(pt(1, -10) == p.rotateCCW(pt(2, 2), PI / 2));
861     assert(pt(10, -3) == p.reflect(pt(0, 0)));
862     assert(pt(-10, -3) == p.reflect(pt(-2, 0), pt(5, 0)));

```

```

863
864     line l(2, -5, -8);
865     line para = line(2, -5, -8).parallel(pt(-6, -2));
866     line perp = line(2, -5, -8).perpendicular(pt(-6, -2));
867     assert(l.is_parallel(para) && l.is_perpendicular(perp));
868     assert(l.slope() == 0.4);
869     assert(para == line(-0.4, 1, -0.4)); // -0.4x + y - 0.4 = 0.
870     assert(perp == line(2.5, 1, 17)); // 2.5x + y + 17 = 0.
871
872     assert(EQ(angle_between(l, perp), 90*DEG));
873
874     assert(EQ(123, reduce_deg(-8*360 + 123)));
875     assert(EQ(1.2345, reduce_rad(2*PI*8 + 1.2345)));
876     assert(polar_point(4, PI) == pt(-4, 0));
877     assert(polar_point(4, -PI/2) == pt(0, -4));
878     assert(EQ(45, polar_angle(pt(5, 5))*RAD));
879     assert(EQ(135, polar_angle(pt(-4, 4))*RAD));
880     assert(EQ(90, angle(pt(5, 0), pt(0, 5), pt(-5, 0))*RAD));
881     assert(EQ(225, angle_between(pt(0, 5), pt(5, -5))*RAD));
882     assert(-1 == cross(pt(0, 1), pt(1, 0), pt(0, 0)));
883     assert(1 == turn(pt(0, 1), pt(0, 0), pt(-5, -5)));
884
885     assert(EQ(5, dist(pt(-1, -1), pt(2, 3))));
886     assert(EQ(25, sqdist(pt(-1, -1), pt(2, 3))));
887     assert(EQ(1.2, line_dist(pt(2, 1), line(-4, 3, -1))));
888     assert(EQ(0.8, line_dist(pt(3, 3), pt(-1, -1), pt(2, 3))));
889     assert(EQ(1.2, line_dist(pt(2, 1), pt(-1, -1), pt(2, 3))));
890     assert(EQ(0.0, line_dist(line(-4, 3, -1), line(8, 6, 2))));
891     assert(EQ(0.8, line_dist(line(-4, 3, -1), line(-8, 6, -10))));
892     assert(EQ(1.0, seg_dist(pt(3, 3), pt(-1, -1), pt(2, 3))));
893     assert(EQ(1.2, seg_dist(pt(2, 1), pt(-1, -1), pt(2, 3))));
894     assert(EQ(0.0, seg_dist(pt(0, 2), pt(3, 3), pt(-1, -1), pt(2, 3))));
895     assert(EQ(0.6, seg_dist(pt(-1, 0), pt(-2, 2), pt(-1, -1), pt(2, 3))));
896
897     assert(line_intersection(line(-1, 1, 0), line(1, 1, -3), &p) == 0);
898     assert(p == pt(1.5, 1.5));
899     assert(line_intersection(pt(0, 0), pt(1, 1), pt(0, 4), pt(4, 0), &p) == 0);
900     assert(p == pt(2, 2));
901
902 {
903     #define test(a, b, c, d, e, f, g, h) \
904         seg_intersection(pt(a, b), pt(c, d), pt(e, f), pt(g, h), &p, &q)
905
906     // Intersection is a point.
907     assert(0 == test(-4, 0, 4, 0, 0, -4, 0, 4) && p == pt(0, 0));
908     assert(0 == test(0, 0, 10, 10, 2, 2, 16, 4) && p == pt(2, 2));
909     assert(0 == test(-2, 2, -2, -2, -2, 0, 0, 0) && p == pt(-2, 0));
910     assert(0 == test(0, 4, 4, 4, 4, 0, 4, 8) && p == pt(4, 4));
911
912     // Intersection is a segment.
913     assert(1 == test(10, 10, 0, 0, 2, 2, 6, 6));
914     assert(p == pt(2, 2) && q == pt(6, 6));
915     assert(1 == test(6, 8, 14, -2, 14, -2, 6, 8));
916     assert(p == pt(6, 8) && q == pt(14, -2));
917
918     // No intersection.
919     assert(-1 == test(6, 8, 8, 10, 12, 12, 4, 4));
920     assert(-1 == test(-4, 2, -8, 8, 0, 0, -4, 6));
921     assert(-1 == test(4, 4, 4, 6, 0, 2, 0, 0));

```

```

922     assert(-1 == test(4, 4, 6, 4, 0, 2, 0, 0));
923     assert(-1 == test(-2, -2, 4, 4, 10, 10, 6, 6));
924     assert(-1 == test(0, 0, 2, 2, 4, 0, 1, 4));
925     assert(-1 == test(2, 2, 2, 8, 4, 4, 6, 4));
926     assert(-1 == test(4, 2, 4, 4, 0, 8, 10, 0));
927 }
928
929 assert(pt(2.5, 2.5) == closest_point(line(-1, -1, 5), pt(0, 0)));
930 assert(pt(3, 0) == closest_point(line(1, 0, -3), pt(0, 0)));
931 assert(pt(0, 3) == closest_point(line(0, 1, -3), pt(0, 0)));
932
933 assert(pt(3, 0) == closest_point(pt(3, 0), pt(3, 3), pt(0, 0)));
934 assert(pt(2, -1) == closest_point(pt(2, -1), pt(4, -1), pt(0, 0)));
935 assert(pt(4, -1) == closest_point(pt(2, -1), pt(4, -1), pt(5, 0)));
936
937 circle c(-2, 5, sqrt(10));
938 assert(c == circle(point(-2, 5), sqrt(10)));
939 assert(c == circle(point(1, 6), point(-5, 4)));
940 assert(c == circle(point(-3, 2), point(-3, 8), point(-1, 8)));
941 assert(c == incircle(point(-12, 5), point(3, 0), point(0, 9)));
942 assert(c.contains(point(-2, 8)) && !c.contains(point(-2, 9)));
943 assert(c.on_edge(point(-1, 2)) && !c.on_edge(point(-1.01, 2)));
944
945 line l1, l2;
946 assert(-1 == tangent(circle(0, 0, 4), pt(1, 1), &l1, &l2));
947 assert(0 == tangent(circle(0, 0, sqrt(2)), pt(1, 1), &l1, &l2));
948 assert(l1 == line(-1, -1, 2));
949 assert(1 == tangent(circle(0, 0, 2), pt(2, 2), &l1, &l2));
950 assert(l1 == line(0, -2, 4));
951 assert(l2 == line(2, 0, -4));
952
953 assert(-1 == intersection(circle(1, 1, 3), line(5, 3, -30), &p, &q));
954 assert(0 == intersection(circle(1, 1, 3), line(0, 1, -4), &p, &q));
955 assert(p == pt(1, 4));
956 assert(1 == intersection(circle(1, 1, 3), line(0, 1, -1), &p, &q));
957 assert(p == pt(4, 1));
958 assert(q == pt(-2, 1));
959
960 assert(-2 == intersection(circle(1, 1, 1), circle(0, 0, 3), &p, &q));
961 assert(-1 == intersection(circle(0, 0, 3), circle(1, 1, 1), &p, &q));
962 assert(0 == intersection(circle(5, 0, 4), circle(-5, 0, 4), &p, &q));
963 assert(1 == intersection(circle(-5, 0, 5), circle(5, 0, 5), &p, &q));
964 assert(p == pt(0, 0));
965 assert(2 == intersection(circle(-0.5, 0, 1), circle(0.5, 0, 1), &p, &q));
966 assert(p == pt(0, -sqrt(3) / 2));
967 assert(q == pt(0, sqrt(3) / 2));
968
// Each circle passes through the other's center.
969 double r = 3, a = intersection_area(circle(-r/2, 0, r), circle(r/2, 0, r));
970 assert(EQ(a, r*r*(2*PI / 3 - sqrt(3) / 2)));
971
972
973 assert(EQ(6, triangle_area(pt(0, -1), pt(4, -1), pt(0, -4))));
974 assert(EQ(6, triangle_area_sides(3, 4, 5)));
975 assert(EQ(6, triangle_area_mediants(3.605551275, 2.5, 4.272001873)));
976 assert(EQ(6, triangle_area_altitudes(3, 4, 2.4)));
977
978 assert(point_in_triangle(pt(0, 0), pt(-1, 0), pt(0, -2), pt(4, 0)));
979 assert(!point_in_triangle(pt(0, 1), pt(-1, 0), pt(0, -2), pt(4, 0)));
980 assert(point_in_triangle(pt(-2.44, 0.82), pt(-1, 0), pt(-3, 1), pt(4, 0)));

```

```

981 assert(!point_in_triangle(pt(-2.44, 0.7), pt(-1, 0), pt(-3, 1), pt(4, 0)));
982
983 assert(EQ(20, rectangle_area(pt(1, 1), pt(5, 6))));
984
985 assert(point_in_rectangle(pt(0, -1), pt(0, -3), 3, 2));
986 assert(point_in_rectangle(pt(2, -2), pt(3, -3), -3, 2));
987 assert(!point_in_rectangle(pt(0, 0), pt(3, -1), -3, -2));
988 assert(point_in_rectangle(pt(2, -2), pt(3, -3), pt(0, -1)));
989 assert(!point_in_rectangle(pt(-1, -2), pt(3, -3), pt(0, -1)));
990
991 assert(-1 == rectangle_intersection(pt(0, 0), pt(1, 1), pt(2, 2), pt(3, 3)));
992 assert(0 == rectangle_intersection(pt(1, 1), pt(7, 7), pt(5, 5), pt(0, 0),
993                                &p, &q));
994 assert(EQP(p, pt(1, 1)) && EQP(q, pt(5, 5)));
995 assert(1 == rectangle_intersection(pt(1, 1), pt(0, 0), pt(0, 0), pt(1, 10),
996                                &p, &q));
997 assert(EQP(p, pt(0, 0)) && EQP(q, pt(1, 1)));
998 assert(2 == rectangle_intersection(pt(0, 5), pt(5, 7), pt(1, 6), pt(2, 5),
999                                &p, &q));
1000 assert(EQP(p, pt(1, 6)) && EQP(q, pt(2, 5)));
1001
1002 return 0;
1003 }
```

6.1 Geometric Classes

6.1.1 Point

```

1 /*
2
3 A two-dimensional real-valued point class supporting epsilon comparisons.
4 Operations include element-wise arithmetic, norm, arg, dot product, cross
5 product, projection, rotation, and reflection. See also std::complex.
6
7 Time Complexity:
8 - O(1) per call to the constructor and all other operations.
9
10 Space Complexity:
11 - O(1) for storage of the point.
12 - O(1) auxiliary for all operations.
13
14 */
15
16 #include <cmath>
17 #include <iostream>
18 #include <utility>
19
20 const double EPS = 1e-9;
21
22 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
23 #define LT(a, b) ((a) < (b) - EPS)
24
25 struct point {
```

```

26     double x, y;
27
28     point() : x(0), y(0) {}
29     point(double x, double y) : x(x), y(y) {}
30     point(const point &p) : x(p.x), y(p.y) {}
31     point(const std::pair<double, double> &p) : x(p.first), y(p.second) {}
32
33     bool operator<(const point &p) const {
34         return EQ(x, p.x) ? LT(y, p.y) : LT(x, p.x);
35     }
36
37     bool operator>(const point &p) const {
38         return EQ(x, p.x) ? LT(p.y, y) : LT(p.x, x);
39     }
40
41     bool operator==(const point &p) const { return EQ(x, p.x) && EQ(y, p.y); }
42     bool operator!=(const point &p) const { return !(*this == p); }
43     bool operator<=(const point &p) const { return !(*this > p); }
44     bool operator>=(const point &p) const { return !(*this < p); }
45     point operator+(const point &p) const { return point(x + p.x, y + p.y); }
46     point operator-(const point &p) const { return point(x - p.x, y - p.y); }
47     point operator+(double v) const { return point(x + v, y + v); }
48     point operator-(double v) const { return point(x - v, y - v); }
49     point operator*(double v) const { return point(x * v, y * v); }
50     point operator/(double v) const { return point(x / v, y / v); }
51     point& operator+=(const point &p) { x += p.x; y += p.y; return *this; }
52     point& operator-=(const point &p) { x -= p.x; y -= p.y; return *this; }
53     point& operator+=(double v) { x += v; y += v; return *this; }
54     point& operator-=(double v) { x -= v; y -= v; return *this; }
55     point& operator*=(double v) { x *= v; y *= v; return *this; }
56     point& operator/=(double v) { x /= v; y /= v; return *this; }
57     friend point operator+(double v, const point &p) { return p + v; }
58     friend point operator*(double v, const point &p) { return p * v; }
59
60     double sqnorm() const { return x*x + y*y; }
61     double norm() const { return sqrt(x*x + y*y); }
62     double arg() const { return atan2(y, x); }
63     double dot(const point &p) const { return x*p.x + y*p.y; }
64     double cross(const point &p) const { return x*p.y - y*p.x; }
65     double proj(const point &p) const { return dot(p) / p.norm(); }
66
67     // Returns a proportional unit vector (p, q) = c(x, y) where p^2 + q^2 = 1.
68     point normalize() const {
69         return (EQ(x, 0) && EQ(y, 0)) ? point(0, 0) : (point(x, y) / norm());
70     }
71
72     // Returns (x, y) rotated 90 degrees clockwise about the origin.
73     point rotate90() const { return point(-y, x); }
74
75     // Returns (x, y) rotated t radians clockwise about the origin.
76     point rotateCW(double t) const {
77         return point(x*cos(t) + y*sin(t), y*cos(t) - x*sin(t));
78     }
79
80     // Returns (x, y) rotated t radians counter-clockwise about the origin.
81     point rotateCCW(double t) const {
82         return point(x*cos(t) - y*sin(t), x*sin(t) + y*cos(t));
83     }
84

```

```
85 // Returns (x, y) rotated t radians clockwise about point p.
86 point rotateCW(const point &p, double t) const {
87     return (*this - p).rotateCW(t) + p;
88 }
89
90 // Returns (x, y) rotated t radians counter-clockwise about the point p.
91 point rotateCCW(const point &p, double t) const {
92     return (*this - p).rotateCCW(t) + p;
93 }
94
95 // Returns (x, y) reflected across point p.
96 point reflect(const point &p) const {
97     return point(2*p.x - x, 2*p.y - y);
98 }
99
100 // Returns (x, y) reflected across the line containing points p and q.
101 point reflect(const point &p, const point &q) const {
102     if (p == q) {
103         return reflect(p);
104     }
105     point r(*this - p), s = q - p;
106     r = point(r.x*s.x + r.y*s.y, r.x*s.y - r.y*s.x) / s.sqnorm();
107     r = point(r.x*s.x - r.y*s.y, r.x*s.y + r.y*s.x) + p;
108     return r;
109 }
110
111 friend double sqnorm(const point &p) { return p.sqnorm(); }
112 friend double norm(const point &p) { return p.norm(); }
113 friend double arg(const point &p) { return p.arg(); }
114 friend double dot(const point &p, const point &q) { return p.dot(q); }
115 friend double cross(const point &p, const point &q) { return p.cross(q); }
116 friend double proj(const point &p, const point &q) { return p.proj(q); }
117 friend point normalize(const point &p) { return p.normalize(); }
118 friend point rotate90(const point &p) { return p.rotate90(); }
119
120 friend point rotateCW(const point &p, double t) {
121     return p.rotateCW(t);
122 }
123
124 friend point rotateCCW(const point &p, double t) {
125     return p.rotateCCW(t);
126 }
127
128 friend point rotateCW(const point &p, const point &q, double t) {
129     return p.rotateCW(q, t);
130 }
131
132 friend point rotateCCW(const point &p, const point &q, double t) {
133     return p.rotateCCW(q, t);
134 }
135
136 friend point reflect(const point &p, const point &q) {
137     return p.reflect(q);
138 }
139
140 friend point reflect(const point &p, const point &a, const point &b) {
141     return p.reflect(a, b);
142 }
143
```

```

144     friend std::ostream& operator<<(std::ostream &out, const point &p) {
145         return out << "(" << (fabs(p.x) < EPS ? 0 : p.x) << ","
146                           << (fabs(p.y) < EPS ? 0 : p.y) << ")";
147     }
148 };
149
150 /** Example Usage ***/
151
152 #include <cassert>
153 #define pt point
154
155 const double PI = acos(-1.0);
156
157 int main() {
158     pt p(-10, 3), q;
159     assert(pt(-18, 29) == p + pt(-3, 9)*6 / 2 - pt(-1, 1));
160     assert(EQ(109, p.sqnorm()));
161     assert(EQ(10.44030650891, p.norm()));
162     assert(EQ(2.850135859112, p.arg()));
163     assert(EQ(0, p.dot(pt(3, 10))));
164     assert(EQ(0, p.cross(pt(10, -3))));
165     assert(EQ(10, p.proj(pt(-10, 0))));
166     assert(EQ(1, p.normalize().norm()));
167     assert(pt(-3, -10) == p.rotate90());
168     assert(pt(3, 12) == p.rotateCW(pt(1, 1), PI / 2));
169     assert(pt(1, -10) == p.rotateCCW(pt(2, 2), PI / 2));
170     assert(pt(10, -3) == p.reflect(pt(0, 0)));
171     assert(pt(-10, -3) == p.reflect(pt(-2, 0), pt(5, 0)));
172     return 0;
173 }
```

6.1.2 Line

```

1 /*
2
3 A straight line in two dimensions supporting epsilon comparisons. The line is
4 represented by the form  $a*x + b*y + c = 0$ , where the coefficients are normalized
5 so that  $b$  is always 1 except for when the line is vertical, in which case  $b = 0$ .
6 Operations include checking if the line is horizontal or vertical, finding the
7 slope, evaluating  $y$  at some  $x$  (and vice versa), checking if a point falls on the
8 line, checking if another line is parallel or perpendicular, and finding the
9 parallel or perpendicular line through a point.
10
11 Time Complexity:
12 -  $O(1)$  per call to the constructor and all other operations.
13
14 Space Complexity:
15 -  $O(1)$  for storage of the line.
16 -  $O(1)$  auxiliary for all operations.
17
18 */
19
20 #include <cmath>
21 #include <limits>
22 #include <iostream>
```

```
23
24 const double EPS = 1e-9;
25 const double M_NAN = std::numeric_limits<double>::quiet_NaN();
26
27 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
28 #define LT(a, b) ((a) < (b) - EPS)
29
30 struct line {
31     double a, b, c;
32
33     line() : a(0), b(0), c(0) {} // Invalid or uninitialized line.
34
35     line(double a, double b, double c) {
36         if (!EQ(b, 0)) {
37             this->a = a / b;
38             this->c = c / b;
39             this->b = 1;
40         } else {
41             this->c = c / a;
42             this->a = 1;
43             this->b = 0;
44         }
45     }
46
47     template<class Point>
48     line(double slope, const Point &p) {
49         a = -slope;
50         b = 1;
51         c = slope * p.x - p.y;
52     }
53
54     template<class Point>
55     line(const Point &p, const Point &q) : a(0), b(0), c(0) {
56         if (EQ(p.x, q.x)) {
57             if (NE(p.y, q.y)) { // Vertical line.
58                 a = 1;
59                 b = 0;
60                 c = -p.x;
61             } // Else, invalid line.
62         } else {
63             a = -(p.y - q.y) / (p.x - q.x);
64             b = 1;
65             c = -(a*p.x) - (b*p.y);
66         }
67     }
68
69     bool operator==(const line &l) const {
70         return EQ(a, l.a) && EQ(b, l.b) && EQ(c, l.c);
71     }
72
73     bool operator!=(const line &l) const {
74         return !(*this == l);
75     }
76
77     // Returns whether the line is initialized and normalized.
78     bool valid() const {
79         if (EQ(a, 0)) {
80             return !EQ(b, 0);
81         }
```

```

82     return EQ(b, 1) || (EQ(b, 0) && EQ(a, 1));
83 }
84
85 bool horizontal() const { return valid() && EQ(a, 0); }
86 bool vertical() const { return valid() && EQ(b, 0); }
87 double slope() const { return (!valid() || EQ(b, 0)) ? M_NAN : -a; }
88
89 // Solve for x at a given y. If the line is horizontal, then either -INF, INF,
90 // or NAN is returned based on whether y is below, above, or on the line.
91 double x(double y) const {
92     if (!valid() || EQ(a, 0)) {
93         return M_NAN; // Invalid or horizontal line.
94     }
95     return (-c - b*y) / a;
96 }
97
98 // Solve for y at a given x. If the line is vertical, then either -INF, INF,
99 // or NAN is returned based on whether x is left of, right of, or on the line.
100 double y(double x) const {
101     if (!valid() || EQ(b, 0)) {
102         return M_NAN; // Invalid or vertical line.
103     }
104     return (-c - a*x) / b;
105 }
106
107 template<class Point>
108 bool contains(const Point &p) const { return EQ(a*p.x + b*p.y + c, 0); }
109
110 bool is_parallel(const line &l) const { return EQ(a, l.a) && EQ(b, l.b); }
111 bool is_perpendicular(const line &l) const { return EQ(-a*l.a, b*l.b); }
112
113 // Return the parallel line passing through point p.
114 template<class Point>
115 line parallel(const Point &p) const {
116     return line(a, b, -a*p.x - b*p.y);
117 }
118
119 // Return the perpendicular line passing through point p.
120 template<class Point>
121 line perpendicular(const Point &p) const {
122     return line(-b, a, b*p.x - a*p.y);
123 }
124
125 friend std::ostream& operator<<(std::ostream &out, const line &l) {
126     return out << (fabs(l.a) < EPS ? 0 : l.a) << "x" << std::showpos
127             << (fabs(l.b) < EPS ? 0 : l.b) << "y"
128             << (fabs(l.c) < EPS ? 0 : l.c) << "=0" << std::noshowpos;
129 }
130 };
131
132 /** Example Usage ***/
133
134 #include <cassert>
135
136 struct point {
137     double x, y;
138     point(double x, double y) : x(x), y(y) {}
139 };
140

```

```

141 int main() {
142     line l(2, -5, -8);
143     line para = line(2, -5, -8).parallel(point(-6, -2));
144     line perp = line(2, -5, -8).perpendicular(point(-6, -2));
145     assert(l.is_parallel(para) && l.is_perpendicular(perp));
146     assert(l.slope() == 0.4);
147     assert(para == line(-0.4, 1, -0.4)); // -0.4x + y - 0.4 = 0.
148     assert(perp == line(2.5, 1, 17)); // 2.5x + y + 17 = 0.
149     return 0;
150 }
```

6.1.3 Circle

```

1 /*
2
3 A circle in two dimensions supporting epsilon comparisons. The circle centered
4 at (h, k) is represented by the relation (x - h)^2 + (y - k)^2 = r^2, where the
5 radius r is normalized to a non-negative number. Operations include constructing
6 a circle from a line segment, constructing a circumcircle, checking if a point
7 falls inside the circle or on its edge, and constructing an incircle.
8
9 Time Complexity:
10 - O(1) per call to the constructors and all other operations.
11
12 Space Complexity:
13 - O(1) for storage of the circle.
14 - O(1) auxiliary for all operations.
15
16 */
17
18 #include <cmath>
19 #include <iostream>
20 #include <stdexcept>
21 #include <utility>
22
23 const double EPS = 1e-9;
24
25 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
26 #define LE(a, b) ((a) <= (b) + EPS)
27 #define LT(a, b) ((a) < (b) - EPS)
28
29 typedef std::pair<double, double> point;
30 #define x first
31 #define y second
32
33 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
34 double norm(const point &a) { return sqrt(sqnorm(a)); }
35
36 struct circle {
37     double h, k, r;
38
39     circle() : h(0), k(0), r(0) {}
40     circle(double r) : h(0), k(0), r(fabs(r)) {}
41     circle(const point &o, double r) : h(o.x), k(o.y), r(fabs(r)) {}
42     circle(double h, double k, double r) : h(h), k(k), r(fabs(r)) {}
```

```

43
44 // Circle with the line segment ab as a diameter.
45 circle(const point &a, const point &b) {
46     h = (a.x + b.x)/2.0;
47     k = (a.y + b.y)/2.0;
48     r = norm(point(a.x - h, a.y - k));
49 }
50
51 // Circumcircle of three points.
52 circle(const point &a, const point &b, const point &c) {
53     double an = sqnorm(point(b.x - c.x, b.y - c.y));
54     double bn = sqnorm(point(a.x - c.x, a.y - c.y));
55     double cn = sqnorm(point(a.x - b.x, a.y - b.y));
56     double wa = an*(bn + cn - an);
57     double wb = bn*(an + cn - bn);
58     double wc = cn*(an + bn - cn);
59     double w = wa + wb + wc;
60     if (EQ(w, 0)) {
61         throw std::runtime_error("No circumcircle from collinear points.");
62     }
63     h = (wa*a.x + wb*b.x + wc*c.x)/w;
64     k = (wa*a.y + wb*b.y + wc*c.y)/w;
65     r = norm(point(a.x - h, a.y - k));
66 }
67
68 // Circle of radius r that contains points a and b. In the general case, there
69 // will be two possible circles and only one is chosen arbitrarily. However if
70 // the diameter is equal to dist(a, b) = 2*r, then there is only one possible
71 // center. If points a and b are identical, then there are infinite circles.
72 // If the points are too far away relative to the radius, then there is no
73 // possible circle. In the latter two cases, an exception is thrown.
74 circle(const point &a, const point &b, double r) : r(fabs(r)) {
75     if (LE(r, 0) && a == b) { // Circle with zero area.
76         h = a.x;
77         k = a.y;
78         return;
79     }
80     double d = norm(point(b.x - a.x, b.y - a.y));
81     if (EQ(d, 0)) {
82         throw std::runtime_error("Identical points, infinite circles.");
83     }
84     if (LT(r*2.0, d)) {
85         throw std::runtime_error("Points too far away to make circle.");
86     }
87     double v = sqrt(r*r - d*d/4.0) / d;
88     point m((a.x + b.x)/2.0, (a.y + b.y)/2.0);
89     h = m.x + v*(a.y - b.y);
90     k = m.y + v*(b.x - a.x);
91     // The other answer is (h, k) = (m.x - v*(a.y - b.y), m.y - v*(b.x - a.x)).
92 }
93
94 bool operator==(const circle &c) const {
95     return EQ(h, c.h) && EQ(k, c.k) && EQ(r, c.r);
96 }
97
98 bool operator!=(const circle &c) const {
99     return !(*this == c);
100 }
101

```

```

102     point center() const { return point(h, k); }
103
104     bool contains(const point &p) const {
105         return LE(sqnorm(point(p.x - h, p.y - k)), r*r);
106     }
107
108     bool on_edge(const point &p) const {
109         return EQ(sqnorm(point(p.x - h, p.y - k)), r*r);
110     }
111
112     friend std::ostream& operator<<(std::ostream &out, const circle &c) {
113         return out << std::showpos << "(x" << -(fabs(c.h) < EPS ? 0 : c.h) << ")^2+"
114                                         << "(y" << -(fabs(c.k) < EPS ? 0 : c.k) << ")^2"
115                                         << std::noshowpos << "=" << (fabs(c.r) < EPS ? 0 : c.r*c.r);
116     }
117 };
118
119 // Returns the circle inscribed inside the triangle abc.
120 circle incircle(const point &a, const point &b, const point &c) {
121     double al = norm(point(b.x - c.x, b.y - c.y));
122     double bl = norm(point(a.x - c.x, a.y - c.y));
123     double cl = norm(point(a.x - b.x, a.y - b.y));
124     double l = al + bl + cl;
125     point p(a.x - c.x, a.y - c.y), q(b.x - c.x, b.y - c.y);
126     return EQ(l, 0) ? circle(a.x, a.y, 0)
127                      : circle((al*a.x + bl*b.x + cl*c.x) / l,
128                               (al*a.y + bl*b.y + cl*c.y) / l,
129                               fabs(p.x*q.y - p.y*q.x) / l);
130 }
131
132 /** Example Usage ***/
133
134 #include <cassert>
135
136 int main() {
137     circle c(-2, 5, sqrt(10));
138     assert(c == circle(point(-2, 5), sqrt(10)));
139     assert(c == circle(point(1, 6), point(-5, 4)));
140     assert(c == circle(point(-3, 2), point(-3, 8), point(-1, 8)));
141     assert(c == incircle(point(-12, 5), point(3, 0), point(0, 9)));
142     assert(c.contains(point(-2, 8)) && !c.contains(point(-2, 9)));
143     assert(c.on_edge(point(-1, 2)) && !c.on_edge(point(-1.01, 2)));
144     return 0;
145 }
```

6.1.4 Triangle

```

1 /*
2
3 Common triangle calculations in two dimensions.
4
5 - triangle_area(a, b, c) returns the area of the triangle abc.
6 - triangle_area_sides(s1, s2, s3) returns the area of a triangle with side
7   lengths s1, s2, and s3. The given lengths must be non-negative and form a
8   valid triangle.
```

```

9 - triangle_area_mediants(m1, m2, m3) returns the area of a triangle with medians
10 of lengths m1, m2, and m3. The median of a triangle is a line segment joining
11 a vertex to the midpoint of the opposing edge.
12 - triangle_area_altitudes(h1, h2, h3) returns the area of a triangle with
13 altitudes h1, h2, and h3. An altitude of a triangle is the shortest line
14 between a vertex and the infinite line that is extended from its opposite
15 edge.
16 - same_side(p1, p2, a, b) returns whether points p1 and p2 lie on the same side
17 of the line containing points a and b. If one or both points lie exactly on
18 the line, then the result will depend on the setting of EDGE_IS_SAME_SIDE.
19 - point_in_triangle(p, a, b, c) returns whether point p lies within the triangle
20 abc. If the point lies on or close to an edge (by roughly EPS), then the
21 result will depend on the setting of EDGE_IS_SAME_SIDE in the function above.
22
23 Time Complexity:
24 - O(1) for all operations.
25
26 Space Complexity:
27 - O(1) auxiliary for all operations.
28 */
29
30
31 #include <algorithm>
32 #include <cmath>
33 #include <utility>
34
35 const double EPS = 1e-9;
36
37 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
38 #define LT(a, b) ((a) < (b) - EPS)
39 #define GT(a, b) ((a) > (b) + EPS)
40 #define LE(a, b) ((a) <= (b) + EPS)
41 #define GE(a, b) ((a) >= (b) - EPS)
42
43 typedef std::pair<double, double> point;
44 #define x first
45 #define y second
46
47 double cross(const point &a, const point &b) { return a.x*b.y - a.y*b.x; }
48
49 double triangle_area(const point &a, const point &b, const point &c) {
50     point ac(a.x - c.x, a.y - c.y), bc(b.x - c.x, b.y - c.y);
51     return fabs(cross(ac, bc)) / 2.0;
52 }
53
54 double triangle_area_sides(double s1, double s2, double s3) {
55     double s = (s1 + s2 + s3) / 2.0;
56     return sqrt(s*(s - s1)*(s - s2)*(s - s3));
57 }
58
59 double triangle_area_mediants(double m1, double m2, double m3) {
60     return 4.0*triangle_area_sides(m1, m2, m3) / 3.0;
61 }
62
63 double triangle_area_altitudes(double h1, double h2, double h3) {
64     if (EQ(h1, 0) || EQ(h2, 0) || EQ(h3, 0)) {
65         return 0;
66     }
67     double x = h1*h1, y = h2*h2, z = h3*h3;

```

```

68     double v = 2.0/(x*y) + 2.0/(x*z) + 2.0/(y*z);
69     return 1.0/sqrt(v - 1.0/(x*x) - 1.0/(y*y) - 1.0/(z*z));
70 }
71
72 bool same_side(const point &p1, const point &p2, const point &a,
73                 const point &b) {
74     static const bool EDGE_IS_SAME_SIDE = true;
75     point ab(b.x - a.x, b.y - a.y);
76     point p1a(p1.x - a.x, p1.y - a.y), p2a(p2.x - a.x, p2.y - a.y);
77     double c1 = cross(ab, p1a), c2 = cross(ab, p2a);
78     return EDGE_IS_SAME_SIDE ? GE(c1*c2, 0) : GT(c1*c2, 0);
79 }
80
81 bool point_in_triangle(const point &p, const point &a, const point &b,
82                        const point &c) {
83     return same_side(p, a, b, c) &&
84            same_side(p, b, a, c) &&
85            same_side(p, c, a, b);
86 }
87
88 /** Example Usage ***/
89
90 #include <cassert>
91
92 int main() {
93     assert(EQ(6, triangle_area(point(0, -1), point(4, -1), point(0, -4))));
94     assert(EQ(6, triangle_area_sides(3, 4, 5)));
95     assert(EQ(6, triangle_area_mediants(3.605551275, 2.5, 4.272001873)));
96     assert(EQ(6, triangle_area_altitudes(3, 4, 2.4)));
97
98     assert(point_in_triangle(point(0, 0),
99                             point(-1, 0), point(0, -2), point(4, 0)));
100    assert(!point_in_triangle(point(0, 1),
101                            point(-1, 0), point(0, -2), point(4, 0)));
102    assert(point_in_triangle(point(-2.44, 0.82),
103                            point(-1, 0), point(-3, 1), point(4, 0)));
104    assert(!point_in_triangle(point(-2.44, 0.7),
105                            point(-1, 0), point(-3, 1), point(4, 0)));
106
107    return 0;
108 }
```

6.1.5 Rectangle

```

1 /*
2
3 Common rectangle calculations in two dimensions.
4
5 - rectangle_area(a, b) returns the area of a rectangle with opposing vertices a
6   and b.
7 - point_in_rectangle(p, x, y, w, h) returns whether point p lies within the
8   rectangle defined by a vertex at v (x, y), a width of w, and a height of h.
9   Note that negative widths and heights are supported. If the point lies on or
10  close to an edge (by roughly EPS), then the result will depend on the setting
11  of EDGE_IS_INSIDE.
12 - point_in_rectangle(p, a, b) returns whether point p lies within the rectangle
```

```

13 with opposing vertices a and b. If the point lies on or close to an edge (by
14 roughly EPS), then the result will depend on the setting of EDGE_IS_INSIDE.
15 - rectangle_intersection(a1, b1, a2, b2, &p, &q) determines the intersection
16 region of the rectangle with opposing vertices a1 and b1 and the rectangle
17 with opposing vertices a2 and b2. Returns -1 if the rectangles are completely
18 disjoint, 0 if the rectangles partially intersect, 1 if the first rectangle is
19 completely inside the second, and 2 if the second rectangle is completely
20 inside the first. If there is an intersection, the opposing vertices of the
21 intersection rectangle will be stored into pointers p and q if they are not
22 NULL. If the intersection is a single point or line segment, then the result
23 will depend on the setting of EDGE_IS_INSIDE within point_in_rectangle().
24
25 Time Complexity:
26 - O(1) for all operations.
27
28 Space Complexity:
29 - O(1) auxiliary for all operations.
30
31 */
32
33 #include <algorithm>
34 #include <cmath>
35 #include <cstddef>
36 #include <utility>
37
38 const double EPS = 1e-9;
39
40 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
41 #define LT(a, b) ((a) < (b) - EPS)
42 #define GT(a, b) ((a) > (b) + EPS)
43 #define LE(a, b) ((a) <= (b) + EPS)
44 #define GE(a, b) ((a) >= (b) - EPS)
45
46 typedef std::pair<double, double> point;
47 #define x first
48 #define y second
49
50 double rectangle_area(const point &a, const point &b) {
51     return fabs((a.x - b.x)*(a.y - b.y));
52 }
53
54 bool point_in_rectangle(const point &p, const point &v, double w, double h) {
55     static const bool EDGE_IS_INSIDE = true;
56     if (w < 0) {
57         return point_in_rectangle(p, point(v.x + w, v.y), -w, h);
58     }
59     if (h < 0) {
60         return point_in_rectangle(p, point(v.x, v.y + h), w, -h);
61     }
62     return EDGE_IS_INSIDE
63         ? (GE(p.x, v.x) && LE(p.x, v.x + w) && GE(p.y, v.y) && LE(p.y, v.y + h))
64         : (GT(p.x, v.x) && LT(p.x, v.x + w) && GT(p.y, v.y) && LT(p.y, v.y + h));
65 }
66
67 bool point_in_rectangle(const point &p, const point &a, const point &b) {
68     double xl = std::min(a.x, b.x), yl = std::min(a.y, b.y);
69     double xh = std::max(a.x, b.x), yh = std::max(a.y, b.y);
70     return point_in_rectangle(p, point(xl, yl), xh - xl, yh - yl);
71 }
```

```

72
73 int rectangle_intersection(const point &a1, const point &b1, const point &a2,
74                             const point &b2, point *p = NULL, point *q = NULL) {
75     bool a1in2 = point_in_rectangle(a1, a2, b2);
76     bool b1in2 = point_in_rectangle(b1, a2, b2);
77     if (a1in2 && b1in2) {
78         if (p != NULL && q != NULL) {
79             *p = std::min(a1, b1);
80             *q = std::max(a1, b1);
81         }
82         return 1; // Rectangle 1 completely inside 2.
83     }
84     if (!a1in2 && !b1in2) {
85         if (point_in_rectangle(a2, a1, b1)) {
86             if (p != NULL && q != NULL) {
87                 *p = std::min(a2, b2);
88                 *q = std::max(a2, b2);
89             }
90             return 2; // Rectangle 2 completely inside 1.
91         }
92         return -1; // Completely disjoint.
93     }
94     if (p != NULL && q != NULL) {
95         if (a1in2) {
96             *p = a1;
97             *q = (a1 < b1) ? std::max(a2, b2) : std::min(a2, b2);
98         } else {
99             *p = b1;
100            *q = (b1 < a1) ? std::max(a2, b2) : std::min(a2, b2);
101        }
102        if (*p > *q) {
103            std::swap(p, q);
104        }
105    }
106    return 0;
107 }

108 /**
109  *** Example Usage ***
110
111 #include <cassert>
112
113 bool EQP(const point &a, const point &b) {
114     return EQ(a.x, b.x) && EQ(a.y, b.y);
115 }
116
117 int main() {
118     assert(EQ(20, rectangle_area(point(1, 1), point(5, 6))));
119
120     assert(point_in_rectangle(point(0, -1), point(0, -3), 3, 2));
121     assert(point_in_rectangle(point(2, -2), point(3, -3), -3, 2));
122     assert(!point_in_rectangle(point(0, 0), point(3, -1), -3, -2));
123     assert(point_in_rectangle(point(2, -2), point(3, -3), point(0, -1)));
124     assert(!point_in_rectangle(point(-1, -2), point(3, -3), point(0, -1)));
125
126     point p, q;
127     assert(-1 == rectangle_intersection(point(0, 0), point(1, 1),
128                                         point(2, 2), point(3, 3)));
129     assert(0 == rectangle_intersection(point(1, 1), point(7, 7),
130                                         point(5, 5), point(0, 0), &p, &q));
130 }
```

```

131 assert(EQP(p, point(1, 1)) && EQP(q, point(5, 5)));
132 assert(1 == rectangle_intersection(point(1, 1), point(0, 0),
133                                     point(0, 0), point(1, 10), &p, &q));
134 assert(EQP(p, point(0, 0)) && EQP(q, point(1, 1)));
135 assert(2 == rectangle_intersection(point(0, 5), point(5, 7),
136                                     point(1, 6), point(2, 5), &p, &q));
137 assert(EQP(p, point(1, 6)) && EQP(q, point(2, 5)));
138
139 return 0;
140 }

```

6.2 Elementary Geometric Calculations

6.2.1 Angles

```

1 /*
2
3 Angle calculations in two dimensions. The constants DEG and RAD may be used as
4 multipliers to convert between degrees and radians. For example, if t is a value
5 in radians, then t*DEG is the equivalent angle in degrees.
6
7 - reduce_deg(t) takes an angle t degrees and returns an equivalent angle in the
8 range [0, 360) degrees. E.g. -630 becomes 90.
9 - reduce_rad(t) takes an angle t radians and returns an equivalent angle in the
10 range [0, 360) radians. E.g. 720.5 becomes 0.5.
11 - polar_point(r, t) returns a two-dimensional Cartesian point given radius r and
12 angle t radians in polar coordinates (see std::polar()).
13 - polar_angle(p) returns the angle in radians of the line segment from (0, 0) to
14 point p, relative counterclockwise to the positive x-axis.
15 - angle(a, o, b) returns the smallest angle in radians formed by the points a,
16 o, b with vertex at point o.
17 - angle_between(a, b) returns the angle in radians of segment from point a to
18 point b, relative counterclockwise to the positive x-axis.
19 - angle_between(a1, b1, a2, b2) returns the smaller angle in radians between
20 two lines a1*x + b1*y + c1 = 0 and a2*x + b2*y + c2 = 0, limited to [0, PI/2].
21 - cross(a, b, o) returns the magnitude (Euclidean norm) of the three-dimensional
22 cross product between points a and b where the z-component is implicitly zero
23 and the origin is implicitly shifted to point o. This operation is also equal
24 to double the signed area of the triangle from these three points.
25 - turn(a, o, b) returns -1 if the path a->o->b forms a left turn on the plane, 0
26 if the path forms a straight line segment, or 1 if it forms a right turn.
27
28 Time Complexity:
29 - O(1) for all operations.
30
31 Space Complexity:
32 - O(1) auxiliary for all operations.
33
34 */
35
36 #include <cmath>
37 #include <utility>
38

```

```
39 const double EPS = 1e-9;
40
41 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
42 #define LT(a, b) ((a) < (b) - EPS)
43 #define GT(a, b) ((a) > (b) + EPS)
44
45 typedef std::pair<double, double> point;
46 #define x first
47 #define y second
48
49 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
50 double norm(const point &a) { return sqrt(sqnorm(a)); }
51
52 const double PI = acos(-1.0), DEG = PI/180, RAD = 180/PI;
53
54 double reduce_deg(double t) {
55     if (t < -360) {
56         return reduce_deg(fmod(t, 360));
57     }
58     if (t < 0) {
59         return t + 360;
60     }
61     return (t >= 360) ? fmod(t, 360) : t;
62 }
63
64 double reduce_rad(double t) {
65     if (t < -2*PI) {
66         return reduce_rad(fmod(t, 2*PI));
67     }
68     if (t < 0) {
69         return t + 2*PI;
70     }
71     return (t >= 2*PI) ? fmod(t, 2*PI) : t;
72 }
73
74 point polar_point(double r, double t) {
75     return point(r*cos(t), r*sin(t));
76 }
77
78 double polar_angle(const point &p) {
79     double t = atan2(p.y, p.x);
80     return (t < 0) ? (t + 2*PI) : t;
81 }
82
83 double angle(const point &a, const point &o, const point &b) {
84     point u(o.x - a.x, o.y - a.y), v(o.x - b.x, o.y - b.y);
85     return acos((u.x*v.x + u.y*v.y) / (norm(u)*norm(v)));
86 }
87
88 double angle_between(const point &a, const point &b) {
89     double t = atan2(a.x*b.y - a.y*b.x, a.x*b.x + a.y*b.y);
90     return (t < 0) ? (t + 2*PI) : t;
91 }
92
93 double angle_between(const double &a1, const double &b1,
94                      const double &a2, const double &b2) {
95     double t = atan2(a1*b2 - a2*b1, a1*a2 + b1*b2);
96     if (t < 0) {
97         t += PI;
```

```

98     }
99     return GT(t, PI / 2) ? (PI - t) : t;
100 }
101
102 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
103     return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
104 }
105
106 int turn(const point &a, const point &o, const point &b) {
107     double c = cross(a, b, o);
108     return LT(c, 0) ? -1 : (GT(c, 0) ? 1 : 0);
109 }
110
111 /** Example Usage ***/
112
113 #include <cassert>
114
115 bool EQP(const point &a, const point &b) {
116     return EQ(a.x, b.x) && EQ(a.y, b.y);
117 }
118
119 int main() {
120     assert(EQ(123, reduce_deg(-8*360 + 123)));
121     assert(EQ(1.2345, reduce_rad(2*PI*8 + 1.2345)));
122     assert(EQP(polar_point(4, PI), point(-4, 0)));
123     assert(EQP(polar_point(4, -PI/2), point(0, -4)));
124     assert(EQ(45, polar_angle(point(5, 5))*RAD));
125     assert(EQ(135*DEG, polar_angle(point(-4, 4))));
126     assert(EQ(90*DEG, angle(point(5, 0), point(0, 5), point(-5, 0))));
127     assert(EQ(225*DEG, angle_between(point(0, 5), point(5, -5))));
128     assert(-1 == cross(point(0, 1), point(1, 0), point(0, 0)));
129     assert(1 == turn(point(0, 1), point(0, 0), point(-5, -5)));
130     return 0;
131 }
```

6.2.2 Distances

```

1 /*
2
3 Distance calculations in two dimensions for points, lines, and line segments.
4
5 - dist(a, b) and sqdist(a, b) respectively return the distance and squared
6   distance between points a and b.
7 - line_dist(p, a, b, c) returns the distance from point p to the line
8   a*x + b*y + c = 0. If the line is invalid (i.e. a = b = 0), then -INF, INF,
9   or NaN is returned based on the sign of c.
10 - line_dist(p, a, b) returns the distance from point p to the infinite line
11   containing points a and b. If the line is invalid (i.e. a = b), then the
12   distance from p to the single point is returned.
13 - line_dist(a1, b1, c1, a2, b2, c2) returns the distance between two lines. If
14   the lines are non-parallel then the distance is considered to be 0. Otherwise,
15   the distance is considered to be the perpendicular distance from any point on
16   one line to the other line.
17 - seg_dist(p, a, b) returns the distance from point p to the line segment ab.
18 - seg_dist(a, b, c, d) returns the minimum distance from any point on the line
```

```
19    segment ab to any point on the line segment cd. This is 0 if the segments
20    touch or intersect.
21
22 Time Complexity:
23 - O(1) for all operations.
24
25 Space Complexity:
26 - O(1) auxiliary for all operations.
27
28 */
29
30 #include <algorithm>
31 #include <cmath>
32 #include <utility>
33
34 const double EPS = 1e-9;
35
36 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
37 #define LE(a, b) ((a) <= (b) + EPS)
38 #define GE(a, b) ((a) >= (b) - EPS)
39
40 typedef std::pair<double, double> point;
41 #define x first
42 #define y second
43
44 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
45 double norm(const point &a) { return sqrt(sqnorm(a)); }
46 double dot(const point &a, const point &b) { return a.x*b.x + a.y*b.y; }
47 double cross(const point &a, const point &b) { return a.x*b.y - a.y*b.x; }
48
49 double dist(const point &a, const point &b) {
50     return norm(point(b.x - a.x, b.y - a.y));
51 }
52
53 double sqdist(const point &a, const point &b) {
54     return sqnorm(point(b.x - a.x, b.y - a.y));
55 }
56
57 double line_dist(const point &p, double a, double b, double c) {
58     return fabs(a*p.x + b*p.y + c) / sqrt(a*a + b*b);
59 }
60
61 double line_dist(const point &p, const point &a, const point &b) {
62     if (EQ(a.x, b.x) && EQ(a.y, b.y)) {
63         return dist(p, a);
64     }
65     double u = ((p.x - a.x)*(b.x - a.x) + (p.y - a.y)*(b.y - a.y)) / sqdist(a, b);
66     return norm(point(a.x + u*(b.x - a.x) - p.x, a.y + u*(b.y - a.y) - p.y));
67 }
68
69 double line_dist(double a1, double b1, double c1,
70                  double a2, double b2, double c2) {
71     if (EQ(a1*b2, a2*b1)) {
72         double factor = EQ(b1, 0) ? (a1 / a2) : (b1 / b2);
73         return EQ(c1, c2*factor) ? 0
74                               : fabs(c2*factor - c1) / sqrt(a1*a1 + b1*b1);
75     }
76     return 0;
77 }
```

```

78
79 double seg_dist(const point &p, const point &a, const point &b) {
80     if (EQ(a.x, b.x) && EQ(a.y, b.y)) {
81         return dist(p, a);
82     }
83     point ab(b.x - a.x, b.y - a.y), ap(p.x - a.x, p.y - a.y);
84     double n = sqnorm(ab), d = dot(ab, ap);
85     if (LE(d, 0) || EQ(n, 0)) {
86         return norm(ap);
87     }
88     return GE(d, n) ? norm(point(ap.x - ab.x, ap.y - ab.y))
89                      : norm(point(ap.x - ab.x*(d / n), ap.y - ab.y*(d / n)));
90 }
91
92 double seg_dist(const point &a, const point &b,
93                 const point &c, const point &d) {
94     point ab(b.x - a.x, b.y - a.y);
95     point ac(c.x - a.x, c.y - a.y);
96     point cd(d.x - c.x, d.y - c.y);
97     double c1 = cross(ab, cd), c2 = cross(ac, ab);
98     if (EQ(c1, 0) && EQ(c2, 0)) {
99         double t0 = dot(ac, ab) / norm(ab), t1 = t0 + dot(cd, ab) / norm(ab);
100        if (LE(std::min(t0, t1), 1) && LE(0, std::max(t0, t1))) {
101            return 0;
102        }
103    } else {
104        double t = cross(ac, cd) / c1, u = c2 / c1;
105        if (!EQ(c1, 0) && LE(0, t) && LE(t, 1) && LE(0, u) && LE(u, 1)) {
106            return 0;
107        }
108    }
109    return std::min(std::min(seg_dist(a, c, d), seg_dist(b, c, d)),
110                  std::min(seg_dist(c, a, b), seg_dist(d, a, b)));
111 }
112
113 point closest_point(const point &a, const point &b, const point &p) {
114     if (EQ(a.x, b.x) && EQ(a.y, b.y)) {
115         return a;
116     }
117     point ap(p.x - a.x, p.y - a.y), ab(b.x - a.x, b.y - a.y);
118     double t = dot(ap, ab) / sqnorm(ab);
119     return (t <= 0) ? a : ((t >= 1) ? b : point(a.x + t*ab.x, a.y + t*ab.y));
120 }
121
122 /* Example Usage */
123
124 #include <cassert>
125
126 int main() {
127     assert(EQ(5, dist(point(-1, -1), point(2, 3))));
128     assert(EQ(25, sqdist(point(-1, -1), point(2, 3))));
129     assert(EQ(1.2, line_dist(point(2, 1), -4, 3, -1)));
130     assert(EQ(0.8, line_dist(point(3, 3), point(-1, -1), point(2, 3))));
131     assert(EQ(1.2, line_dist(point(2, 1), point(-1, -1), point(2, 3))));
132     assert(EQ(0, line_dist(-4, 3, -1, 8, 6, 2)));
133     assert(EQ(0.8, line_dist(-4, 3, -1, -8, 6, -10)));
134     assert(EQ(1.0, seg_dist(point(3, 3), point(-1, -1), point(2, 3))));
135     assert(EQ(1.2, seg_dist(point(2, 1), point(-1, -1), point(2, 3))));
136     assert(EQ(0, seg_dist(point(0, 2), point(3, 3), point(-1, -1), point(2, 3))));

```

```

137     assert(EQ(0.6,
138             seg_dist(point(-1, 0), point(-2, 2), point(-1, -1), point(2, 3)));
139     return 0;
140 }
```

6.2.3 Line Intersection

```

1  /*
2
3  Intersection and closest point calculations in two dimensions for straight lines
4  and line segments.
5
6  - line_intersection(a1, b1, c1, a2, b2, c2, &p) determines whether the lines
7  a1*x + b1*y + c1 = 0 and a2*x + b2*y + c2 = 0 intersect, returning -1 if
8  there is no intersection because the lines are parallel, 0 if there is exactly
9  one intersection (in which case the intersection point is stored into pointer
10 p if it's not NULL), or 1 if there are infinite intersections because the
11 lines are identical.
12 - line_intersection(p1, p2, p3, p4, &p) determines whether the infinite lines
13 (not segments) through points p1, p2 and through points p3 and p4 intersect,
14 returning -1 if there is no intersection because the lines are parallel, 0 if
15 there is exactly one intersection (in which case the intersection point is
16 stored into pointer p if it's not NULL), or 1 if there are infinite
17 intersections because the lines are identical.
18 - seg_intersection(a, b, c, d, &p, &q) determines whether the line segment ab
19 intersects the line segment cd, returning -1 if the segments do not intersect,
20 0 if there is exactly one intersection point (in which case it is stored into
21 pointer p if it's not NULL), or 1 if the intersection is another line segment
22 (in which case the two endpoints are stored into pointers p and q if they are
23 not NULL). If the segments are barely touching (close within EPS), then the
24 result will depend on the setting of TOUCH_IS_INTERSECT.
25 - closest_point(a, b, c, p) returns the point on line a*x + b*y + c = 0 that is
26 closest to point p. Note that the result always lies on the line through p
27 which is perpendicular to the line a*x + b*y + c = 0.
28 - closest_point(a, b, p) returns the point on segment ab closest to point p.
29
30 Time Complexity:
31 - O(1) for all operations.
32
33 Space Complexity:
34 - O(1) auxiliary for all operations.
35
36 */
37
38 #include <algorithm>
39 #include <cmath>
40 #include <cstddef>
41 #include <utility>
42
43 const double EPS = 1e-9;
44
45 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
46 #define LT(a, b) ((a) < (b) - EPS)
47 #define LE(a, b) ((a) <= (b) + EPS)
48
```

```

49 typedef std::pair<double, double> point;
50 #define x first
51 #define y second
52
53 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
54 double norm(const point &a) { return sqrt(sqnorm(a)); }
55 double dot(const point &a, const point &b) { return a.x*b.x + a.y*b.y; }
56 double cross(const point &a, const point &b) { return a.x*b.y - a.y*b.x; }
57
58 int line_intersection(double a1, double b1, double c1, double a2, double b2,
59                      double c2, point *p = NULL) {
60     if (EQ(a1, a2) && EQ(b1, b2)) {
61         return EQ(c1, c2) ? 1 : -1;
62     }
63     if (p != NULL) {
64         p->x = (b1*c1 - b1*c2) / (a2*b1 - a1*b2);
65         if (!EQ(b1, 0)) {
66             p->y = -(a1*p->x + c1) / b1;
67         } else {
68             p->y = -(a2*p->x + c2) / b2;
69         }
70     }
71     return 0;
72 }
73
74 int line_intersection(const point &p1, const point &p2,
75                      const point &p3, const point &p4, point *p = NULL) {
76     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
77     double c1 = -(p1.x*p2.y - p2.x*p1.y);
78     double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
79     double c2 = -(p3.x*p4.y - p4.x*p3.y);
80     double x = -(c1*b2 - c2*b1), y = -(a1*c2 - a2*c1);
81     double det = a1*b2 - a2*b1;
82     if (EQ(det, 0)) {
83         return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
84     }
85     if (p != NULL) {
86         *p = point(x / det, y / det);
87     }
88     return 0;
89 }
90
91 int seg_intersection(const point &a, const point &b, const point &c,
92                      const point &d, point *p = NULL, point *q = NULL) {
93     static const bool TOUCH_IS_INTERSECT = true;
94     point ab(b.x - a.x, b.y - a.y);
95     point ac(c.x - a.x, c.y - a.y);
96     point cd(d.x - c.x, d.y - c.y);
97     double c1 = cross(ab, cd), c2 = cross(ac, ab);
98     if (EQ(c1, 0) && EQ(c2, 0)) { // Collinear.
99         double t0 = dot(ac, ab) / sqnorm(ab);
100        double t1 = t0 + dot(cd, ab) / sqnorm(ab);
101        double mint = std::min(t0, t1), maxt = std::max(t0, t1);
102        bool overlap = TOUCH_IS_INTERSECT ? (LE(mint, 1) && LE(0, maxt))
103                                         : (LT(mint, 1) && LT(0, maxt));
104        if (overlap) {
105            point res1 = std::max(std::min(a, b), std::min(c, d));
106            point res2 = std::min(std::max(a, b), std::max(c, d));
107            if (res1 == res2) {

```

```

108     if (p != NULL) {
109         *p = res1;
110     }
111     return 0; // Collinear and meeting at an endpoint.
112 }
113 if (p != NULL && q != NULL) {
114     *p = res1;
115     *q = res2;
116 }
117 return 1; // Collinear and overlapping.
118 } else {
119     return -1; // Collinear and disjoint.
120 }
121 }
122 if (EQ(c1, 0)) {
123     return -1; // Parallel and disjoint.
124 }
125 double t = cross(ac, cd)/c1, u = c2/c1;
126 bool t_between_01 = TOUCH_IS_INTERSECT ? (LE(0, t) && LE(t, 1))
127                                         : (LT(0, t) && LT(t, 1));
128 bool u_between_01 = TOUCH_IS_INTERSECT ? (LE(0, u) && LE(u, 1))
129                                         : (LT(0, u) && LT(u, 1));
130 if (t_between_01 && u_between_01) {
131     if (p != NULL) {
132         *p = point(a.x + t*ab.x, a.y + t*ab.y);
133     }
134     return 0; // Non-parallel with one intersection.
135 }
136 return -1; // Non-parallel with no intersections.
137 }
138
139 point closest_point(double a, double b, double c, const point &p) {
140     if (EQ(a, 0)) {
141         return point(p.x, -c); // Horizontal line.
142     }
143     if (EQ(b, 0)) {
144         return point(-c, p.y); // Vertical line.
145     }
146     point res;
147     line_intersection(a, b, c, -b, a, b*p.x - a*p.y, &res);
148     return res;
149 }
150
151 point closest_point(const point &a, const point &b, const point &p) {
152     if (a == b) return a;
153     point ap(p.x - a.x, p.y - a.y), ab(b.x - a.x, b.y - a.y);
154     double t = dot(ap, ab) / norm(ab);
155     if (t <= 0) return a;
156     if (t >= 1) return b;
157     return point(a.x + t * ab.x, a.y + t * ab.y);
158 }
159
160 /** Example Usage ***/
161
162 #include <cassert>
163 #define point point
164
165 bool EQP(const point &a, const point &b) {
166     return EQ(a.x, b.x) && EQ(a.y, b.y);

```

```

167 }
168
169 int main() {
170     point p, q;
171
172     assert(line_intersection(-1, 1, 0, 1, 1, -3, &p) == 0);
173     assert(EQP(p, point(1.5, 1.5)));
174     assert(line_intersection(point(0, 0), point(1, 1), point(0, 4), point(4, 0),
175                             &p) == 0);
176     assert(EQP(p, point(2, 2)));
177
178 {
179     #define test(a, b, c, d, e, f, g, h) seg_intersection( \
180         point(a, b), point(c, d), point(e, f), point(g, h), &p, &q)
181
182     // Intersection is a point.
183     assert(0 == test(-4, 0, 4, 0, 0, -4, 0, 4) && EQP(p, point(0, 0)));
184     assert(0 == test(0, 0, 10, 10, 2, 2, 16, 4) && EQP(p, point(2, 2)));
185     assert(0 == test(-2, 2, -2, -2, 0, 0, 0) && EQP(p, point(-2, 0)));
186     assert(0 == test(0, 4, 4, 4, 0, 4, 8) && EQP(p, point(4, 4)));
187
188     // Intersection is a segment.
189     assert(1 == test(10, 10, 0, 0, 2, 2, 6, 6));
190     assert(EQP(p, point(2, 2)) && EQP(q, point(6, 6)));
191     assert(1 == test(6, 8, 14, -2, 14, -2, 6, 8));
192     assert(EQP(p, point(6, 8)) && EQP(q, point(14, -2)));
193
194     // No intersection.
195     assert(-1 == test(6, 8, 8, 10, 12, 12, 4, 4));
196     assert(-1 == test(-4, 2, -8, 8, 0, 0, -4, 6));
197     assert(-1 == test(4, 4, 4, 6, 0, 2, 0, 0));
198     assert(-1 == test(4, 4, 6, 4, 0, 2, 0, 0));
199     assert(-1 == test(-2, -2, 4, 4, 10, 10, 6, 6));
200     assert(-1 == test(0, 0, 2, 2, 4, 0, 1, 4));
201     assert(-1 == test(2, 2, 2, 8, 4, 4, 6, 4));
202     assert(-1 == test(4, 2, 4, 4, 0, 8, 10, 0));
203 }
204
205 assert(EQP(point(2.5, 2.5), closest_point(-1, -1, 5, point(0, 0))));
206 assert(EQP(point(3, 0), closest_point(1, 0, -3, point(0, 0))));
207 assert(EQP(point(0, 3), closest_point(0, 1, -3, point(0, 0))));
208
209 assert(EQP(point(3, 0),
210             closest_point(point(3, 0), point(3, 3), point(0, 0))));
211 assert(EQP(point(2, -1),
212             closest_point(point(2, -1), point(4, -1), point(0, 0))));
213 assert(EQP(point(4, -1),
214             closest_point(point(2, -1), point(4, -1), point(5, 0))));
215 return 0;
216 }
```

6.2.4 Circle Intersection

```

1 /*
2
```



```

62     }
63 };
64
65 struct line {
66     double a, b, c;
67
68     line() : a(0), b(0), c(0) {}
69
70     line(double a, double b, double c) {
71         if (!EQ(b, 0)) {
72             this->a = a / b;
73             this->c = c / b;
74             this->b = 1;
75         } else {
76             this->c = c / a;
77             this->a = 1;
78             this->b = 0;
79         }
80     }
81
82     line(const point &p, const point &q) : a(0), b(0), c(0) {
83         if (EQ(p.x, q.x)) {
84             if (NE(p.y, q.y)) { // Vertical line.
85                 a = 1;
86                 b = 0;
87                 c = -p.x;
88             } // Else, invalid line.
89         } else {
90             a = -(p.y - q.y) / (p.x - q.x);
91             b = 1;
92             c = -(a*p.x) - (b*p.y);
93         }
94     }
95 };
96
97 int tangent(const circle &c, const point &p, line *l1 = NULL, line *l2 = NULL) {
98     point vop(p.x - c.h, p.y - c.k);
99     if (EQ(sqnorm(vop), c.r*c.r)) { // Point on an edge.
100         if (l1 != 0) { // Get perpendicular line through p.
101             *l1 = line(point(c.h, c.k), p);
102             *l1 = line(-l1->b, l1->a, l1->b*p.x - l1->a*p.y);
103         }
104         return 0;
105     }
106     if (LE(sqnorm(vop), c.r*c.r)) {
107         return -1; // Point inside circle, no intersection.
108     }
109     point q(vop.x / c.r, vop.y / c.r);
110     double n = sqnorm(q), d = q.y*sqrt(sqnorm(q) - 1.0);
111     point t1((q.x - d) / n, c.k), t2((q.x + d) / n, c.k);
112     if (NE(q.y, 0)) { // Common case.
113         t1.y += c.r*(1.0 - t1.x*q.x) / q.y;
114         t2.y += c.r*(1.0 - t2.x*q.x) / q.y;
115     } else { // Point at center horizontal, y = 0.
116         d = c.r*sqrt(1.0 - t1.x*t1.x);
117         t1.y += d;
118         t2.y -= d;
119     }
120     t1.x = t1.x*c.r + c.h;

```

```

121 t2.x = t2.x*c.r + c.h;
122 //note: here, t1 and t2 are the two points of tangencies
123 if (l1 != NULL && l2 != NULL) {
124     *l1 = line(p, t1);
125     *l2 = line(p, t2);
126 }
127 return 1;
128 }
129
130 int intersection(const circle &c, const line &l, point *p = NULL,
131                   point *q = NULL) {
132     double v = c.h*l.a + c.k*l.b + l.c;
133     double aabb = l.a*l.a + l.b*l.b;
134     double disc = v*v / aabb - c.r*c.r;
135     if (disc > EPS) {
136         return -1;
137     }
138     double x0 = -l.a*l.c / aabb, y0 = -l.b*v / aabb;
139     if (disc > -EPS) {
140         if (p != NULL) {
141             *p = point(x0 + c.h, y0 + c.k);
142         }
143         return 0;
144     }
145     double k = sqrt(std::max(0.0, disc / -aabb));
146     if (p != NULL && q != NULL) {
147         *p = point(x0 + k*l.b + c.h, y0 - k*l.a + c.k);
148         *q = point(x0 - k*l.b + c.h, y0 + k*l.a + c.k);
149     }
150     return 1;
151 }
152
153 int intersection(const circle &c1, const circle &c2, point *p = NULL,
154                   point *q = NULL) {
155     if (EQ(c1.h, c2.h) && EQ(c1.k, c2.k)) {
156         return EQ(c1.r, c2.r) ? 3 : (c1.r > c2.r ? -1 : -2);
157     }
158     point d12(point(c2.h - c1.h, c2.k - c1.k));
159     double d = norm(d12);
160     if (GT(d, c1.r + c2.r)) {
161         return 0;
162     }
163     if (LT(d, fabs(c1.r - c2.r))) {
164         return c1.r > c2.r ? -1 : -2;
165     }
166     double a = (c1.r*c1.r - c2.r*c2.r + d*d) / (2*d);
167     double x0 = c1.h + (d12.x*a / d), y0 = c1.k + (d12.y*a / d);
168     double s = sqrt(c1.r*c1.r - a*a), rx = -d12.y*s / d, ry = d12.x*s / d;
169     if (EQ(rx, 0) && EQ(ry, 0)) {
170         if (p != NULL) {
171             *p = point(x0, y0);
172         }
173         return 1;
174     }
175     if (p != NULL && q != NULL) {
176         *p = point(x0 - rx, y0 - ry);
177         *q = point(x0 + rx, y0 + ry);
178     }
179     return 2;

```

```

180 }
181
182 double intersection_area(const circle &c1, const circle &c2) {
183     double r = std::min(c1.r, c2.r), R = std::max(c1.r, c2.r);
184     double d = norm(point(c2.h - c1.h, c2.k - c1.k));
185     if (LE(d, R - r)) {
186         return PI*r*r;
187     }
188     if (GE(d, R + r)) {
189         return 0;
190     }
191     return r*r*acos((d*d + r*r - R*R) / 2 / d / r) +
192         R*R*acos((d*d + R*R - r*r) / 2 / d / R) -
193         0.5*sqrt((-d + r + R)*(d + r - R)*(d - r + R)*(d + r + R));
194 }
195
196 /** Example Usage **/
197
198 #include <cassert>
199
200 bool EQP(const point &a, const point &b) {
201     return EQ(a.x, b.x) && EQ(a.y, b.y);
202 }
203
204 bool EQL(const line &l1, const line &l2) {
205     return EQ(l1.a, l2.a) && EQ(l1.b, l2.b) && EQ(l1.c, l2.c);
206 }
207
208 int main() {
209     line l1, l2;
210     assert(-1 == tangent(circle(0, 0, 4), point(1, 1), &l1, &l2));
211     assert(0 == tangent(circle(0, 0, sqrt(2)), point(1, 1), &l1, &l2));
212     assert(EQL(l1, line(-1, -1, 2)));
213     assert(1 == tangent(circle(0, 0, 2), point(2, 2), &l1, &l2));
214     assert(EQL(l1, line(0, -2, 4)));
215     assert(EQL(l2, line(2, 0, -4)));
216
217     point p, q;
218     assert(-1 == intersection(circle(1, 1, 3), line(5, 3, -30), &p, &q));
219     assert(0 == intersection(circle(1, 1, 3), line(0, 1, -4), &p, &q));
220     assert(EQP(p, point(1, 4)));
221     assert(1 == intersection(circle(1, 1, 3), line(0, 1, -1), &p, &q));
222     assert(EQP(p, point(4, 1)));
223     assert(EQP(q, point(-2, 1)));
224
225     assert(-2 == intersection(circle(1, 1, 1), circle(0, 0, 3), &p, &q));
226     assert(-1 == intersection(circle(0, 0, 3), circle(1, 1, 1), &p, &q));
227     assert(0 == intersection(circle(5, 0, 4), circle(-5, 0, 4), &p, &q));
228     assert(1 == intersection(circle(-5, 0, 5), circle(5, 0, 5), &p, &q));
229     assert(EQP(p, point(0, 0)));
230     assert(2 == intersection(circle(-0.5, 0, 1), circle(0.5, 0, 1), &p, &q));
231     assert(EQP(p, point(0, -sqrt(3) / 2)));
232     assert(EQP(q, point(0, sqrt(3) / 2)));
233
234 // Each circle passes through the other's center.
235     double r = 3, a = intersection_area(circle(-r/2, 0, r), circle(r/2, 0, r));
236     assert(EQ(a, r*r*(2*PI / 3 - sqrt(3) / 2)));
237     return 0;
238 }
```

6.3 Intermediate Geometric Calculations

6.3.1 Polygon Sorting and Area

```

1  /*
2
3 Given a list of distinct points in two-dimensions, order them into a valid
4 polygon and determine the area.
5
6 - mean_center(lo, hi) returns the arithmetic mean of a range [lo, hi] of points,
7   where lo and hi must be random-access iterators. This point is mathematically
8   guaranteed to lie within the non-self-intersecting closed polygon constructed
9   by sorting all other points clockwise about it. Note that this is different
10  from the geometric centroid (a.k.a. barycenter) of a polygon.
11 - cw_comp(a, b, c) returns whether point a compares clockwise "before" point b
12  when using c as a central reference point.
13 - cw_comp_class(c) constructs a wrapper class of cw_comp() that may be passed to
14  std::sort() a range of points clockwise to produce a valid polygon.
15 - ccw_comp_class(c) constructs a wrapper class of cw_comp() that may be passed
16  to std::sort() a range of points counter-clockwise to produce a valid polygon.
17 - polygon_area(lo, hi) returns the area of the polygon specified by the range
18  [lo, hi] of points, where lo and hi must be BidirectionalIterators. The points
19  are interpreted as a polygon based on the order given in the range. The input
20  polygon does not have to be sorted using the methods above, but must be given
21  in some ordering that yields a valid non-self-intersecting closed polygon.
22  Optionally, the last point may be equal to the first point in the input
23  without affecting the result. The area is computed using the shoelace formula.
24
25 Time Complexity:
26 - O(n) per call to mean_center(lo, hi) and polygon_area(lo, hi), where n is the
27   distance between lo and hi.
28 - O(1) per call to cw_comp() and the related class comparators.
29
30 Space Complexity:
31 - O(1) auxiliary for all operations.
32
33 */
34
35 #include <algorithm>
36 #include <cmath>
37 #include <stdexcept>
38 #include <utility>
39
40 const double EPS = 1e-9;
41
42 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
43 #define LT(a, b) ((a) < (b) - EPS)
44 #define GE(a, b) ((a) >= (b) - EPS)
45
46 typedef std::pair<double, double> point;
47 #define x first
48 #define y second
49
50 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
51 double cross(const point &a, const point &b) { return a.x*b.y - a.y*b.x; }
52

```

```

53 template<class It>
54 point mean_center(It lo, It hi) {
55     if (lo == hi) {
56         throw std::runtime_error("Cannot get center of an empty range.");
57     }
58     double x_sum = 0, y_sum = 0, num_points = hi - lo;
59     for (; lo != hi; ++lo) {
60         x_sum += lo->x;
61         y_sum += lo->y;
62     }
63     return point(x_sum / num_points, y_sum / num_points);
64 }
65
66 bool cw_comp(const point &a, const point &b, const point &c) {
67     if (GE(a.x - c.x, 0) && LT(b.x - c.x, 0)) {
68         return true;
69     }
70     if (LT(a.x - c.x, 0) && GE(b.x - c.x, 0)) {
71         return false;
72     }
73     if (EQ(a.x - c.x, 0) && EQ(b.x - c.x, 0)) {
74         if (GE(a.y - c.y, 0) || GE(b.y - c.y, 0)) {
75             return a.y > b.y;
76         }
77         return b.y > a.y;
78     }
79     point ac(a.x - c.x, a.y - c.y), bc(b.x - c.x, b.y - c.y);
80     double det = cross(ac, bc);
81     if (EQ(det, 0)) {
82         return sqnorm(ac) > sqnorm(bc);
83     }
84     return det < 0;
85 }
86
87 struct cw_comp_class {
88     point c;
89     cw_comp_class(const point &c) : c(c) {}
90     bool operator()(const point &a, const point &b) const {
91         return cw_comp(a, b, c);
92     }
93 };
94
95 struct ccw_comp_class {
96     point c;
97     ccw_comp_class(const point &c) : c(c) {}
98     bool operator()(const point &a, const point &b) const {
99         return cw_comp(b, a, c);
100    }
101 };
102
103 template<class It>
104 double polygon_area(It lo, It hi) {
105     if (lo == hi) {
106         return 0;
107     }
108     double area = 0;
109     if (*lo != *hi) {
110         area += (lo->x - hi->x)*(lo->y + hi->y);
111     }

```

```

112     for (It i = hi, j = --hi; i != lo; --i, --j) {
113         area += (i->x - j->x)*(i->y + j->y);
114     }
115     return fabs(area / 2.0);
116 }
117
118 /** Example Usage ***/
119
120 #include <cassert>
121 #include <vector>
122 using namespace std;
123
124 int main() {
125     // Irregular pentagon with only the vertex (1, 2) not on its convex hull.
126     // The ordering here is already sorted in ccw order around their mean center,
127     // though we will shuffle them to verify our sorting comparator.
128     point points[] = {point(1, 3),
129                         point(1, 2),
130                         point(2, 1),
131                         point(0, 0),
132                         point(-1, 3)};
133     vector<point> v(points, points + 5);
134     std::random_shuffle(v.begin(), v.end());
135     point c = mean_center(v.begin(), v.end());
136     assert(EQ(c.x, 0.6) && EQ(c.y, 1.8));
137     sort(v.begin(), v.end(), cw_comp_class(c));
138     for (int i = 0; i < (int)v.size(); i++) {
139         assert(v[i] == points[i]);
140     }
141     assert(EQ(polygon_area(v.begin(), v.end()), 5));
142     return 0;
143 }
```

6.3.2 Point-in-Polygon (Ray Casting)

```

1 /*
2
3 Given a point p and a polygon in two dimensions, determine whether p lies inside
4 the polygon using a ray casting algorithm.
5
6 - point_in_polygon(p, lo, hi) returns whether p lies within the polygon defined
7   by the range [lo, hi) of points specifying the vertices in either clockwise
8   or counter-clockwise order, where lo and hi must be random-access iterators.
9   If p lies barely on an edge (within EPS), then the result will depend on the
10  setting of EDGE_IS_INSIDE.
11
12 Time Complexity:
13 - O(n) per call to point_in_polygon(lo, hi), where n is the distance between lo
14 and hi.
15
16 Space Complexity:
17 - O(1) auxiliary.
18
19 */
20
```

```

21 #include <cmath>
22 #include <utility>
23
24 const double EPS = 1e-9;
25
26 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
27 #define LE(a, b) ((a) <= (b) + EPS)
28 #define GT(a, b) ((a) > (b) + EPS)
29
30 typedef std::pair<double, double> point;
31 #define x first
32 #define y second
33
34 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
35     return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
36 }
37
38 template<class It>
39 bool point_in_polygon(const point &p, It lo, It hi) {
40     static const bool EDGE_IS_INSIDE = true;
41     bool ans = 0;
42     for (It i = lo, j = hi - 1; i != hi; j = i++) {
43         if (EQ(i->y, p.y) &&
44             (EQ(i->x, p.x) ||
45              (EQ(j->y, p.y) && (LE(i->x, p.x) || LE(j->x, p.x))))) {
46             return EDGE_IS_INSIDE;
47         }
48         if (GT(i->y, p.y) != GT(j->y, p.y)) {
49             double det = cross(*i, *j, p);
50             if (EQ(det, 0)) {
51                 return EDGE_IS_INSIDE;
52             }
53             if (GT(det, 0) != GT(j->y, i->y)) {
54                 ans = !ans;
55             }
56         }
57     }
58     return ans;
59 }
60
61 /** Example Usage ***/
62
63 #include <cassert>
64 using namespace std;
65
66 int main() {
67     // Irregular trapezoid.
68     point p[] = {point(-1, 3), point(1, 3), point(2, 1), point(0, 0)};
69     assert(point_in_polygon(point(1, 2), p, p + 4));
70     assert(point_in_polygon(point(0, 3), p, p + 4));
71     assert(!point_in_polygon(point(0, 3.01), p, p + 4));
72     assert(!point_in_polygon(point(2, 2), p, p + 4));
73     return 0;
74 }
```

6.3.3 Convex Hull and Diametral Pair

```

1  /*
2
3 Given a list of points in two dimensions, determine the convex hull using the
4 monotone chain algorithm, and the diameter of the points using the method of
5 rotating calipers. The convex hull is the smallest convex polygon (a polygon
6 such that every line crossing through it will only do so once) that contains all
7 of its points.
8
9 - convex_hull(lo, hi) returns the convex hull as a vector of polygon vertices in
10 clockwise order, given a range [lo, hi) of points where lo and hi must be
11 random-access iterators. The input range will be sorted lexicographically (by
12 x, then by y) after the function call. Note that to produce the hull points in
13 counter-clockwise order, replace every GE() comparison with LE(). To have the
14 first point on the hull repeated as the last in the resulting vector, the
15 final res.resize(k - 1) may be changed to res.resize(k).
16 - diametral_pair(lo, hi) returns a maximum diametral pair given a range [lo, hi)
17 of points where lo and hi must be random-access iterators. The input range
18 will be sorted lexicographically (by x, then by y) after the function call.
19
20 Time Complexity:
21 - O(n log n) per call to convex_hull(lo, hi) and diametral_pair(lo, hi), where n
22 is the distance between lo and hi.
23
24 Space Complexity:
25 - O(n) auxiliary for storage of the convex hull in both operations.
26
27 */
28
29 #include <algorithm>
30 #include <cmath>
31 #include <utility>
32 #include <vector>
33
34 const double EPS = 1e-9;
35
36 #define GE(a, b) ((a) >= (b) - EPS)
37
38 typedef std::pair<double, double> point;
39 #define x first
40 #define y second
41
42 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
43 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
44     return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
45 }
46
47 template<class It>
48 std::vector<point> convex_hull(It lo, It hi) {
49     int k = 0;
50     if (hi - lo <= 1) {
51         return std::vector<point>(lo, hi);
52     }
53     std::vector<point> res(2*(int)(hi - lo));
54     std::sort(lo, hi);
55     for (It it = lo; it != hi; ++it) {

```

```

56     while (k >= 2 && GE(cross(res[k - 1], *it, res[k - 2]), 0)) {
57         k--;
58     }
59     res[k++] = *it;
60 }
61 int t = k + 1;
62 for (It it = hi - 2; it != lo - 1; --it) {
63     while (k >= t && GE(cross(res[k - 1], *it, res[k - 2]), 0)) {
64         k--;
65     }
66     res[k++] = *it;
67 }
68 res.resize(k - 1);
69 return res;
70 }

71
72 template<class It>
73 std::pair<point, point> diametral_pair(It lo, It hi) {
74     std::vector<point> h = convex_hull(lo, hi);
75     int m = h.size();
76     if (m == 1) {
77         return std::make_pair(h[0], h[0]);
78     }
79     if (m == 2) {
80         return std::make_pair(h[0], h[1]);
81     }
82     int k = 1;
83     while (fabs(cross(h[0], h[(k + 1) % m], h[m - 1])) >
84            fabs(cross(h[0], h[k], h[m - 1]))) {
85         k++;
86     }
87     double maxdist = 0, d;
88     std::pair<point, point> res;
89     for (int i = 0, j = k; i <= k && j < m; i++) {
90         d = sqnorm(point(h[i].x - h[j].x, h[i].y - h[j].y));
91         if (d > maxdist) {
92             maxdist = d;
93             res = std::make_pair(h[i], h[j]);
94         }
95         while (j < m && fabs(cross(h[(i + 1) % m], h[(j + 1) % m], h[i])) >
96                fabs(cross(h[(i + 1) % m], h[j], h[i]))) {
97             d = sqnorm(point(h[i].x - h[(j + 1) % m].x, h[i].y - h[(j + 1) % m].y));
98             if (d > maxdist) {
99                 maxdist = d;
100                res = std::make_pair(h[i], h[(j + 1) % m]);
101            }
102            j++;
103        }
104    }
105    return res;
106 }

107
108 /** Example Usage ***/
109
110 #include <cassert>
111 using namespace std;
112
113 int main() {
114     { // Irregular pentagon with only the vertex (1, 2) not on the hull.

```

```

115     vector<point> v;
116     v.push_back(point(1, 3));
117     v.push_back(point(1, 2));
118     v.push_back(point(2, 1));
119     v.push_back(point(0, 0));
120     v.push_back(point(-1, 3));
121     std::random_shuffle(v.begin(), v.end());
122     vector<point> h;
123     h.push_back(point(-1, 3));
124     h.push_back(point(1, 3));
125     h.push_back(point(2, 1));
126     h.push_back(point(0, 0));
127     assert(convex_hull(v.begin(), v.end()) == h);
128 }
129 {
130     vector<point> v;
131     v.push_back(point(0, 0));
132     v.push_back(point(3, 0));
133     v.push_back(point(0, 3));
134     v.push_back(point(1, 1));
135     v.push_back(point(4, 4));
136     pair<point, point> res = diametral_pair(v.begin(), v.end());
137     assert(res.first == point(0, 0));
138     assert(res.second == point(4, 4));
139 }
140     return 0;
141 }
```

6.3.4 Minimum Enclosing Circle

```

1 /*
2
3 Given a list of points in two dimensions, find the circle with smallest area
4 which contains all the given points using a randomized algorithm.
5
6 - minimum_enclosing_circle(lo, hi) returns the minimum enclosing circle given a
7   range [lo, hi] of points, where lo and hi must be random-access iterators. The
8   input range will be shuffled after the function call, though this is only to
9   avoid the worst-case running time and is not necessary for correctness.
10
11 Time Complexity:
12 - O(n) on average per call to minimum_enclosing_circle(lo, hi), where n is the
13   distance between lo and hi.
14
15 Space Complexity:
16 - O(1) auxiliary.
17
18 */
19
20 #include <algorithm>
21 #include <cmath>
22 #include <stdexcept>
23 #include <utility>
24
25 const double EPS = 1e-9;
```

```

26
27 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
28 #define LE(a, b) ((a) <= (b) + EPS)
29
30 typedef std::pair<double, double> point;
31 define x first
32 define y second
33
34 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
35 double norm(const point &a) { return sqrt(sqnorm(a)); }
36
37 struct circle {
38     double h, k, r;
39
40     circle() : h(0), k(0), r(0) {}
41     circle(double h, double k, double r) : h(h), k(k), r(fabs(r)) {}
42
43     // Circle with the line segment ab as a diameter.
44     circle(const point &a, const point &b) {
45         h = (a.x + b.x)/2.0;
46         k = (a.y + b.y)/2.0;
47         r = norm(point(a.x - h, a.y - k));
48     }
49
50     // Circumcircle of three points.
51     circle(const point &a, const point &b, const point &c) {
52         double an = sqnorm(point(b.x - c.x, b.y - c.y));
53         double bn = sqnorm(point(a.x - c.x, a.y - c.y));
54         double cn = sqnorm(point(a.x - b.x, a.y - b.y));
55         double wa = an*(bn + cn - an);
56         double wb = bn*(an + cn - bn);
57         double wc = cn*(an + bn - cn);
58         double w = wa + wb + wc;
59         if (EQ(w, 0)) {
60             throw std::runtime_error("No circumcircle from collinear points.");
61         }
62         h = (wa*a.x + wb*b.x + wc*c.x)/w;
63         k = (wa*a.y + wb*b.y + wc*c.y)/w;
64         r = norm(point(a.x - h, a.y - k));
65     }
66
67     bool contains(const point &p) const {
68         return LE(sqnorm(point(p.x - h, p.y - k)), r*r);
69     }
70 };
71
72 template<class It>
73 circle minimum_enclosing_circle(It lo, It hi) {
74     if (lo == hi) {
75         return circle(0, 0, 0);
76     }
77     if (lo + 1 == hi) {
78         return circle(lo->x, lo->y, 0);
79     }
80     std::random_shuffle(lo, hi);
81     circle res(*lo, *(lo + 1));
82     for (It i = lo + 2; i != hi; ++i) {
83         if (res.contains(*i)) {
84             continue;

```

```

85     }
86     res = circle(*lo, *i);
87     for (It j = lo + 1; j != i; ++j) {
88         if (res.contains(*j)) {
89             continue;
90         }
91         res = circle(*i, *j);
92         for (It k = lo; k != j; ++k) {
93             if (!res.contains(*k)) {
94                 res = circle(*i, *j, *k);
95             }
96         }
97     }
98 }
99 return res;
100}
101
102/** Example Usage ***/
103
104#include <cassert>
105#include <vector>
106using namespace std;
107
108int main() {
109    vector<point> v;
110    v.push_back(point(0, 0));
111    v.push_back(point(0, 1));
112    v.push_back(point(1, 0));
113    v.push_back(point(1, 1));
114    circle res = minimum_enclosing_circle(v.begin(), v.end());
115    assert(EQ(res.h, 0.5) && EQ(res.k, 0.5) && EQ(res.r, 1/sqrt(2)));
116    return 0;
117}

```

6.3.5 Closest Pair

```

1 /*
2
3 Given a list of points in two dimensions, find the closest pair among them using
4 a divide and conquer algorithm.
5
6 - closest_pair(lo, hi, &res) returns the minimum Euclidean distance between any
7   two pair of points in the range [lo, hi), where lo and hi must be
8   random-access iterators. The input range will be sorted lexicographically (by
9   x, then by y) after the function call. If there is an answer, the closest pair
10  will be stored into pointer *res.
11
12 Time Complexity:
13 - O(n log^2 n) per call to closest_pair(lo, hi, &res), where n is the distance
14  between lo and hi.
15
16 Space Complexity:
17 - O(n log^2 n) auxiliary stack space for closest_pair(lo, hi, &res), where n is
18  the distance between lo and hi.
19

```

```

20 */
21
22 #include <algorithm>
23 #include <cmath>
24 #include <limits>
25 #include <utility>
26
27 const double EPS = 1e-9;
28
29 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
30 #define LT(a, b) ((a) < (b) - EPS)
31
32 typedef std::pair<double, double> point;
33 #define x first
34 #define y second
35
36 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
37 double norm(const point &a) { return sqrt(sqnorm(a)); }
38 bool cmp_x(const point &a, const point &b) { return LT(a.x, b.x); }
39 bool cmp_y(const point &a, const point &b) { return LT(a.y, b.y); }
40
41 template<class It>
42 double closest_pair(It lo, It hi, std::pair<point, point> *res = NULL,
43                     double mindist = std::numeric_limits<double>::max(),
44                     bool sort_x = true) {
45     if (lo == hi) {
46         return std::numeric_limits<double>::max();
47     }
48     if (sort_x) {
49         std::sort(lo, hi, cmp_x);
50     }
51     It mid = lo + (hi - lo)/2;
52     double midx = mid->x;
53     double d1 = closest_pair(lo, mid, res, mindist, false);
54     mindist = std::min(mindist, d1);
55     double d2 = closest_pair(mid + 1, hi, res, mindist, false);
56     mindist = std::min(mindist, d2);
57     std::sort(lo, hi, cmp_y);
58     int size = 0;
59     It t[hi - lo];
60     for (It it = lo; it != hi; ++it) {
61         if (fabs(it->x - midx) < mindist) {
62             t[size++] = it;
63         }
64     }
65     for (int i = 0; i < size; i++) {
66         for (int j = i + 1; j < size; j++) {
67             point a(*t[i]), b(*t[j]);
68             if (b.y - a.y >= mindist) {
69                 break;
70             }
71             double dist = norm(point(a.x - b.x, a.y - b.y));
72             if (mindist > dist) {
73                 mindist = dist;
74                 if (res) {
75                     *res = std::make_pair(a, b);
76                 }
77             }
78         }
    }
}

```

```

79     }
80     return mindist;
81 }
82
83 /** Example Usage **/
84
85 #include <cassert>
86 #include <vector>
87 using namespace std;
88
89 int main() {
90     vector<point> v;
91     v.push_back(point(2, 3));
92     v.push_back(point(12, 30));
93     v.push_back(point(40, 50));
94     v.push_back(point(5, 1));
95     v.push_back(point(12, 10));
96     v.push_back(point(3, 4));
97     pair<point, point> res;
98     assert(EQ(closest_pair(v.begin(), v.end(), &res), sqrt(2)));
99     assert(res.first == point(2, 3));
100    assert(res.second == point(3, 4));
101    return 0;
102 }
```

6.3.6 Segment Intersection Finding

```

1 /*
2
3 Given a list of line segments in two dimensions, determine whether any pair of
4 segments intersect using a sweep line algorithm.
5
6 - find_intersection(lo, hi, &res1, &res2) returns whether any pair of segments
7   intersect given a range [lo, hi) of segments, where lo and hi are
8   random-access iterators. If there an intersection is found, then one such pair
9   of segments will be stored into pointers res1 and res2. If some segments are
10  barely touching (close within EPS), then the result will depend on the setting
11  of TOUCH_IS_INTERSECT.
12
13 Time Complexity:
14 - O(n log n) per call to find_intersection(lo, hi, &res1, &res2), where n is
15   the distance between lo and hi.
16
17 Space Complexity:
18 - O(n) auxiliary heap space for find_intersection(lo, hi, &res1, &res2), where n
19   is the distance between lo and hi.
20
21 */
22
23 #include <algorithm>
24 #include <cmath>
25 #include <set>
26 #include <utility>
27
28 const double EPS = 1e-9;
```

```

29
30 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
31 #define LT(a, b) ((a) < (b) - EPS)
32 #define LE(a, b) ((a) <= (b) + EPS)
33
34 typedef std::pair<double, double> point;
35 #define x first
36 #define y second
37
38 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
39 double norm(const point &a) { return sqrt(sqnorm(a)); }
40 double dot(const point &a, const point &b) { return a.x*b.x + a.y*b.y; }
41 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
42     return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
43 }
44
45 int seg_intersection(const point &a, const point &b, const point &c,
46                      const point &d, point *p = NULL, point *q = NULL) {
47     static const bool TOUCH_IS_INTERSECT = true;
48     point ab(b.x - a.x, b.y - a.y);
49     point ac(c.x - a.x, c.y - a.y);
50     point cd(d.x - c.x, d.y - c.y);
51     double c1 = cross(ab, cd), c2 = cross(ac, ab);
52     if (EQ(c1, 0) && EQ(c2, 0)) { // Collinear.
53         double t0 = dot(ac, ab) / sqnorm(ab);
54         double t1 = t0 + dot(cd, ab) / sqnorm(ab);
55         double mint = std::min(t0, t1), maxt = std::max(t0, t1);
56         bool overlap = TOUCH_IS_INTERSECT ? (LE(mint, 1) && LE(0, maxt))
57                                         : (LT(mint, 1) && LT(0, maxt));
58         if (overlap) {
59             point res1 = std::max(std::min(a, b), std::min(c, d));
60             point res2 = std::min(std::max(a, b), std::max(c, d));
61             if (res1 == res2) {
62                 if (p != NULL) {
63                     *p = res1;
64                 }
65                 return 0; // Collinear and meeting at an endpoint.
66             }
67             if (p != NULL && q != NULL) {
68                 *p = res1;
69                 *q = res2;
70             }
71             return 1; // Collinear and overlapping.
72         } else {
73             return -1; // Collinear and disjoint.
74         }
75     }
76     if (EQ(c1, 0)) {
77         return -1; // Parallel and disjoint.
78     }
79     double t = cross(ac, cd)/c1, u = c2/c1;
80     bool t_between_01 = TOUCH_IS_INTERSECT ? (LE(0, t) && LE(t, 1))
81                                         : (LT(0, t) && LT(t, 1));
82     bool u_between_01 = TOUCH_IS_INTERSECT ? (LE(0, u) && LE(u, 1))
83                                         : (LT(0, u) && LT(u, 1));
84     if (t_between_01 && u_between_01) {
85         if (p != NULL) {
86             *p = point(a.x + t*ab.x, a.y + t*ab.y);
87         }

```

```

88     return 0; // Non-parallel with one intersection.
89 }
90     return -1; // Non-parallel with no intersections.
91 }
92
93 struct segment {
94     point p, q;
95
96     segment() {}
97     segment(const point &p, const point &q) : p(min(p, q)), q(max(p, q)) {}
98
99     bool operator<(const segment &rhs) const {
100         if (p.x < rhs.p.x) {
101             double c = cross(q, rhs.p, p);
102             if (c != 0) {
103                 return c > 0;
104             }
105         } else if (rhs.p.x < p.x) {
106             double c = cross(rhs.q, q, rhs.p);
107             if (c != 0) {
108                 return c < 0;
109             }
110         }
111         return p.y < rhs.p.y;
112     }
113 };
114
115 template<class SegIt>
116 struct event {
117     point p;
118     int type;
119     SegIt seg;
120
121     event() {}
122     event(const point &p, int type, SegIt seg) : p(p), type(type), seg(seg) {}
123
124     bool operator<(const event &rhs) const {
125         if (p.x != rhs.p.x) {
126             return p.x < rhs.p.x;
127         }
128         if (type != rhs.type) {
129             return rhs.type < type;
130         }
131         return p.y < rhs.p.y;
132     }
133 };
134
135 bool intersect(const segment &s1, const segment &s2) {
136     return seg_intersection(s1.p, s1.q, s2.p, s2.q) >= 0;
137 }
138
139 template<class It>
140 bool find_intersection(It lo, It hi, segment *res1, segment *res2) {
141     int cnt = 0;
142     event<It> e[2*(int)(hi - lo)];
143     for (It it = lo; it != hi; ++it) {
144         if (it->p > it->q) {
145             std::swap(it->p, it->q);
146         }

```

```

147     e[cnt++] = event<It>(it->p, 1, it);
148     e[cnt++] = event<It>(it->q, -1, it);
149 }
150 std::sort(e, e + cnt);
151 std::set<segment> s;
152 std::set<segment>::iterator it, next, prev;
153 for (int i = 0; i < cnt; i++) {
154     It seg = e[i].seg;
155     if (e[i].type == 1) {
156         it = s.lower_bound(*seg);
157         if (it != s.end() && intersect(*it, *seg)) {
158             *res1 = *it;
159             *res2 = *seg;
160             return true;
161         }
162         if (it != s.begin() && intersect(*--it, *seg)) {
163             *res1 = *it;
164             *res2 = *seg;
165             return true;
166         }
167         s.insert(*seg);
168     } else {
169         it = s.lower_bound(*seg);
170         next = prev = it;
171         prev = it;
172         if (it != s.begin() && it != --s.end()) {
173             if (intersect(*(++next), *(--prev))) {
174                 *res1 = *next;
175                 *res2 = *prev;
176                 return true;
177             }
178         }
179         s.erase(it);
180     }
181 }
182 return false;
183 }

184 /**
185  *** Example Usage ***
186
187 #include <vector>
188 using namespace std;
189
190 int main() {
191     vector<segment> v;
192     v.push_back(segment(point(0, 0), point(2, 2)));
193     v.push_back(segment(point(3, 0), point(0, -1)));
194     v.push_back(segment(point(0, 2), point(2, -2)));
195     v.push_back(segment(point(0, 3), point(9, 0)));
196     segment res1, res2;
197     assert(find_intersection(v.begin(), v.end(), &res1, &res2));
198     assert(res1.p == point(0, 0) && res1.q == point(2, 2));
199     assert(res2.p == point(0, 2) && res2.q == point(2, -2));
200     return 0;
201 }

```

6.4 Advanced Geometric Computations

6.4.1 Convex Polygon Cut

```

1  /*
2
3 Given a convex polygon (a polygon such that every line crossing through it will
4 only do so once) in two dimensions, and two points specifying an infinite line,
5 cut off the right part of the polygon, and return the resulting left part.
6
7 - convex_cut(lo, hi, p, q) returns the points of the left side of a polygon, in
8 clockwise order, after it has been cut by the line containing points p and q.
9 The original convex polygon is given by the range [lo, hi) of points in
10 clockwise order, where lo and hi must be random-access iterators.
11
12 Time Complexity:
13 - O(n) per call to convex_cut(lo, hi, p, q), where n is the distance between lo
14 and hi.
15
16 Space Complexity:
17 - O(n) auxiliary for storage of the resulting convex cut.
18
19 */
20
21 #include <cmath>
22 #include <cstddef>
23 #include <utility>
24 #include <vector>
25
26 const double EPS = 1e-9;
27
28 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
29 #define LT(a, b) ((a) < (b) - EPS)
30 #define GT(a, b) ((a) > (b) + EPS)
31
32 typedef std::pair<double, double> point;
33 #define x first
34 #define y second
35
36 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
37     return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
38 }
39
40 int turn(const point &a, const point &o, const point &b) {
41     double c = cross(a, b, o);
42     return LT(c, 0) ? -1 : (GT(c, 0) ? 1 : 0);
43 }
44
45 int line_intersection(const point &p1, const point &p2,
46                       const point &p3, const point &p4, point *p = NULL) {
47     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
48     double c1 = -(p1.x*p2.y - p2.x*p1.y);
49     double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
50     double c2 = -(p3.x*p4.y - p4.x*p3.y);
51     double x = -(c1*b2 - c2*b1), y = -(a1*c2 - a2*c1);
52     double det = a1*b2 - a2*b1;

```

```

53     if (EQ(det, 0)) {
54         return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
55     }
56     if (p != NULL) {
57         *p = point(x / det, y / det);
58     }
59     return 0;
60 }
61
62 template<class It>
63 std::vector<point> convex_cut(It lo, It hi, const point &p, const point &q) {
64     if (EQ(p.x, q.x) && EQ(p.y, q.y)) {
65         throw std::runtime_error("Cannot cut using line from identical points.");
66     }
67     std::vector<point> res;
68     for (It i = lo, j = hi - 1; i != hi; j = i++) {
69         int d1 = turn(q, p, *j), d2 = turn(q, p, *i);
70         if (d1 >= 0) {
71             res.push_back(*j);
72         }
73         if (d1*d2 < 0) {
74             point r;
75             line_intersection(p, q, *j, *i, &r);
76             res.push_back(r);
77         }
78     }
79     return res;
80 }
81
82 /** Example Usage **/
83
84 #include <cassert>
85 using namespace std;
86
87 int main() {
88     {
89         vector<point> v;
90         v.push_back(point(1, 3));
91         v.push_back(point(2, 2));
92         v.push_back(point(2, 1));
93         v.push_back(point(0, 0));
94         v.push_back(point(-1, 3));
95         // Cut using the vertical line through (0, 0).
96         vector<point> c;
97         c.push_back(point(-1, 3));
98         c.push_back(point(0, 3));
99         c.push_back(point(0, 0));
100        assert(convex_cut(v.begin(), v.end(), point(0, 0), point(0, 1)) == c);
101    }
102    { // On a non-convex input, the result may be multiple disjoint polygons!
103        vector<point> v;
104        v.push_back(point(0, 0));
105        v.push_back(point(2, 2));
106        v.push_back(point(0, 4));
107        v.push_back(point(3, 4));
108        v.push_back(point(3, 0));
109        vector<point> c;
110        c.push_back(point(1, 0));
111        c.push_back(point(0, 0));

```

```

112     c.push_back(point(1, 1));
113     c.push_back(point(1, 3));
114     c.push_back(point(0, 4));
115     c.push_back(point(1, 4));
116     assert(convex_cut(v.begin(), v.end(), point(1, 0), point(1, 4)) == c);
117 }
118 return 0;
119 }
```

6.4.2 Polygon Intersection and Union

```

1 /*
2
3 Given two polygons, determine the areas of their intersection and union using a
4 sweep line algorithm and the inclusion-exclusion principle.
5
6 - intersection_area(lo1, hi1, lo2, hi2) returns the intersection area of two
7 polygons respectively specified by two ranges [lo1, hi1) and [lo2, hi2) of
8 vertices in clockwise order, where lo1, hi1, lo2, and hi2 must be
9 random-access iterators.
10 - union_area(lo1, hi1, lo2, hi2) returns the union area of two polygons
11 respectively specified by two ranges [lo1, hi1) and [lo2, hi2) of vertices in
12 clockwise order, where lo1, hi1, lo2, and hi2 must be random-access iterators.
13
14 Time Complexity:
15 - O(n^2 log n) per call to intersection_area(lo1, hi1, lo2, hi2) and
16 union_area(lo1, hi1, lo2, hi2) where n is the sum of distances between lo1 and
17 hi1 and lo2 and hi2 respectively.
18
19 Space Complexity:
20 - O(n) auxiliary heap space for intersection_area(lo1, hi1, lo2, hi2) and
21 union_area(lo1, hi1, lo2, hi2), where n is the sum of distances between lo1
22 and hi1 and lo2 and hi2 respectively.
23 */
24
25
26 #include <algorithm>
27 #include <cmath>
28 #include <set>
29 #include <utility>
30 #include <vector>
31
32 const double EPS = 1e-9;
33
34 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
35 #define LT(a, b) ((a) < (b) - EPS)
36 #define LE(a, b) ((a) <= (b) + EPS)
37
38 typedef std::pair<double, double> point;
39 #define x first
40 #define y second
41
42 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
43 double dot(const point &a, const point &b) { return a.x*b.x + a.y*b.y; }
44 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
```

```

45     return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
46 }
47
48 int seg_intersection(const point &a, const point &b, const point &c,
49                      const point &d, point *p = NULL, point *q = NULL) {
50     static const bool TOUCH_IS_INTERSECT = true;
51     point ab(b.x - a.x, b.y - a.y);
52     point ac(c.x - a.x, c.y - a.y);
53     point cd(d.x - c.x, d.y - c.y);
54     double c1 = cross(ab, cd), c2 = cross(ac, ab);
55     if (EQ(c1, 0) && EQ(c2, 0)) { // Collinear.
56         double t0 = dot(ac, ab) / sqnorm(ab);
57         double t1 = t0 + dot(cd, ab) / sqnorm(ab);
58         double mint = std::min(t0, t1), maxt = std::max(t0, t1);
59         bool overlap = TOUCH_IS_INTERSECT ? (LE(mint, 1) && LE(0, maxt))
60                                         : (LT(mint, 1) && LT(0, maxt));
61         if (overlap) {
62             point res1 = std::max(std::min(a, b), std::min(c, d));
63             point res2 = std::min(std::max(a, b), std::max(c, d));
64             if (res1 == res2) {
65                 if (p != NULL) {
66                     *p = res1;
67                 }
68                 return 0; // Collinear and meeting at an endpoint.
69             }
70             if (p != NULL && q != NULL) {
71                 *p = res1;
72                 *q = res2;
73             }
74             return 1; // Collinear and overlapping.
75         } else {
76             return -1; // Collinear and disjoint.
77         }
78     }
79     if (EQ(c1, 0)) {
80         return -1; // Parallel and disjoint.
81     }
82     double t = cross(ac, cd)/c1, u = c2/c1;
83     bool t_between_01 = TOUCH_IS_INTERSECT ? (LE(0, t) && LE(t, 1))
84                                             : (LT(0, t) && LT(t, 1));
85     bool u_between_01 = TOUCH_IS_INTERSECT ? (LE(0, u) && LE(u, 1))
86                                             : (LT(0, u) && LT(u, 1));
87     if (t_between_01 && u_between_01) {
88         if (p != NULL) {
89             *p = point(a.x + t*ab.x, a.y + t*ab.y);
90         }
91         return 0; // Non-parallel with one intersection.
92     }
93     return -1; // Non-parallel with no intersections.
94 }
95
96 int line_intersection(const point &p1, const point &p2,
97                       const point &p3, const point &p4, point *p = NULL) {
98     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
99     double c1 = -(p1.x*p2.y - p2.x*p1.y);
100    double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
101    double c2 = -(p3.x*p4.y - p4.x*p3.y);
102    double x = -(c1*b2 - c2*b1), y = -(a1*c2 - a2*c1);
103    double det = a1*b2 - a2*b1;

```

```

104     if (EQ(det, 0)) {
105         return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
106     }
107     if (p != NULL) {
108         *p = point(x / det, y / det);
109     }
110     return 0;
111 }
112
113 struct event {
114     double y;
115     int mask_delta;
116
117     event(double y = 0, int mask_delta = 0) {
118         this->y = y;
119         this->mask_delta = mask_delta;
120     }
121
122     bool operator<(const event &e) const {
123         if (y != e.y) {
124             return y < e.y;
125         }
126         return mask_delta < e.mask_delta;
127     }
128 };
129
130 template<class It>
131 double intersection_area(It lo1, It hi1, It lo2, It hi2) {
132     It plo[2] = {lo1, lo2}, phi[] = {hi1, hi2};
133     std::set<double> xs;
134     for (It i1 = lo1; i1 != hi1; ++i1) {
135         xs.insert(i1->x);
136     }
137     for (It i2 = lo2; i2 != hi2; ++i2) {
138         xs.insert(i2->x);
139     }
140     for (It i1 = lo1, j1 = hi1 - 1; i1 != hi1; j1 = i1++) {
141         for (It i2 = lo2, j2 = hi2 - 1; i2 != hi2; j2 = i2++) {
142             point p;
143             if (seg_intersection(*i1, *j1, *i2, *j2, &p) == 0) {
144                 xs.insert(p.x);
145             }
146         }
147     }
148     std::vector<double> xsa(xs.begin(), xs.end());
149     double res = 0;
150     for (int k = 0; k < (int)xsa.size() - 1; k++) {
151         double x = (xsa[k] + xsa[k + 1])/2;
152         point sweep0(x, 0), sweep1(x, 1);
153         std::vector<event> events;
154         for (int poly = 0; poly < 2; poly++) {
155             It lo = plo[poly], hi = phi[poly];
156             double area = 0;
157             for (It i = lo, j = hi - 1; i != hi; j = i++) {
158                 area += (j->x - i->x)*(j->y + i->y);
159             }
160             for (It j = lo, i = hi - 1; j != hi; i = j++) {
161                 point p;
162                 if (line_intersection(*j, *i, sweep0, sweep1, &p) == 0) {

```

```

163     double y = p.y, x0 = i->x, x1 = j->x;
164     int sgn_area = (area < 0 ? -1 : (area > 0 ? 1 : 0));
165     if (x0 < x && x1 > x) {
166         events.push_back(event(y, sgn_area*(1 << poly)));
167     } else if (x0 > x && x1 < x) {
168         events.push_back(event(y, -sgn_area*(1 << poly)));
169     }
170 }
171 }
172 std::sort(events.begin(), events.end());
173 double a = 0;
174 int mask = 0;
175 for (int j = 0; j < (int)events.size(); j++) {
176     if (mask == 3) {
177         a += events[j].y - events[j - 1].y;
178     }
179     mask += events[j].mask_delta;
180 }
181 res += a*(xsa[k + 1] - xsa[k]);
182 }
183 return res;
184 }
185 }

186 template<class It>
187 double polygon_area(It lo, It hi) {
188     if (lo == hi) {
189         return 0;
190     }
191     double area = 0;
192     if (*lo != *--hi) {
193         area += (lo->x - hi->x)*(lo->y + hi->y);
194     }
195     for (It i = hi, j = --hi; i != lo; --i, --j) {
196         area += (i->x - j->x)*(i->y + j->y);
197     }
198     return fabs(area / 2.0);
199 }
200 }

201 template<class It>
202 double union_area(It lo1, It hi1, It lo2, It hi2) {
203     return polygon_area(lo1, hi1) + polygon_area(lo2, hi2) -
204         intersection_area(lo1, hi1, lo2, hi2);
205 }
206 }

207 /**
208  *** Example Usage ***
209 */

210 #include <cassert>
211 using namespace std;
212
213 int main() {
214     vector<point> p, s;
215     // Irregular pentagon a triangle of area 1.5 overlapping quadrant 2.
216     p.push_back(point(1, 3));
217     p.push_back(point(1, 2));
218     p.push_back(point(2, 1));
219     p.push_back(point(0, 0));
220     p.push_back(point(-1, 3));
221     // Square of area 12.5 in quadrant 2.

```

```

222     s.push_back(point(0, 0));
223     s.push_back(point(0, 3));
224     s.push_back(point(-3, 3));
225     s.push_back(point(-3, 0));
226     assert(EQ(1.5, intersection_area(p.begin(), p.end(), s.begin(), s.end())));
227     assert(EQ(12.5, union_area(p.begin(), p.end(), s.begin(), s.end())));
228     return 0;
229 }
```

6.4.3 Delaunay Triangulation (Simple)

```

1  /*
2
3 Given a set P of two dimensional points, the Delaunay triangulation of P is a
4 set of non-overlapping triangles that covers the entire convex hull of P such
5 that no point in P lies within the circumcircle of any of the resulting
6 triangles. For any point p in the convex hull of P (but not necessarily in P),
7 the nearest point is guaranteed to be a vertex of the enclosing triangle from
8 the triangulation.
9
10 The triangulation may not exist (e.g. for a set of collinear points), or may not
11 be unique if it does exists. The following program assumes its existence and
12 produces one such valid result using a simple algorithm which encases each
13 triangle in a circle and rejecting the triangle if another point in the
14 tessellation is within the generalized circle.
15
16 - delaunay_triangulation(lo, hi) returns a Delaunay triangulation for the input
17 range [lo, hi] of points, where lo and hi must be random-access iterators, or
18 an empty vector if a triangulation does not exist.
19
20 Time Complexity:
21 - O(n^4) per call to delaunay_triangulation(lo, hi), where n is the distance
22 between lo and hi.
23
24 Space Complexity:
25 - O(n) auxiliary heap space for storage of the Delaunay triangulation.
26
27 */
28
29 #include <algorithm>
30 #include <cmath>
31 #include <utility>
32 #include <vector>
33
34 const double EPS = 1e-9;
35
36 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
37 #define LT(a, b) ((a) < (b) - EPS)
38 #define LE(a, b) ((a) <= (b) + EPS)
39
40 typedef std::pair<double, double> point;
41 #define x first
42 #define y second
43
44 double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
```

```

45 double dot(const point &a, const point &b) { return a.x*b.x + a.y*b.y; }
46 double cross(const point &a, const point &b, const point &o = point(0, 0)) {
47     return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
48 }
49
50 int seg_intersection(const point &a, const point &b, const point &c,
51                      const point &d, point *p = NULL, point *q = NULL) {
52     static const bool TOUCH_IS_INTERSECT = false; // false is important!
53     point ab(b.x - a.x, b.y - a.y);
54     point ac(c.x - a.x, c.y - a.y);
55     point cd(d.x - c.x, d.y - c.y);
56     double c1 = cross(ab, cd), c2 = cross(ac, ab);
57     if (EQ(c1, 0) && EQ(c2, 0)) { // Collinear.
58         double t0 = dot(ac, ab) / sqnorm(ab);
59         double t1 = t0 + dot(cd, ab) / sqnorm(ab);
60         double mint = std::min(t0, t1), maxt = std::max(t0, t1);
61         bool overlap = TOUCH_IS_INTERSECT ? (LE(mint, 1) && LE(0, maxt))
62                                         : (LT(mint, 1) && LT(0, maxt));
63         if (overlap) {
64             point res1 = std::max(std::min(a, b), std::min(c, d));
65             point res2 = std::min(std::max(a, b), std::max(c, d));
66             if (res1 == res2) {
67                 if (p != NULL) {
68                     *p = res1;
69                 }
70                 return 0; // Collinear and meeting at an endpoint.
71             }
72             if (p != NULL && q != NULL) {
73                 *p = res1;
74                 *q = res2;
75             }
76             return 1; // Collinear and overlapping.
77         } else {
78             return -1; // Collinear and disjoint.
79         }
80     }
81     if (EQ(c1, 0)) {
82         return -1; // Parallel and disjoint.
83     }
84     double t = cross(ac, cd)/c1, u = c2/c1;
85     bool t_between_01 = TOUCH_IS_INTERSECT ? (LE(0, t) && LE(t, 1))
86                                             : (LT(0, t) && LT(t, 1));
87     bool u_between_01 = TOUCH_IS_INTERSECT ? (LE(0, u) && LE(u, 1))
88                                             : (LT(0, u) && LT(u, 1));
89     if (t_between_01 && u_between_01) {
90         if (p != NULL) {
91             *p = point(a.x + t*ab.x, a.y + t*ab.y);
92         }
93         return 0; // Non-parallel with one intersection.
94     }
95     return -1; // Non-parallel with no intersections.
96 }
97
98 struct triangle {
99     point a, b, c;
100
101    triangle(const point &a, const point &b, const point &c) : a(a), b(b), c(c) {}
102
103    bool operator==(const triangle &t) const {

```

```

104     return EQ(a.x, t.a.x) && EQ(a.y, t.a.y) &&
105         EQ(b.x, t.b.x) && EQ(b.y, t.b.y) &&
106         EQ(c.x, t.c.x) && EQ(c.y, t.c.y);
107     }
108 };
109
110 template<class It>
111 std::vector<triangle> delaunay_triangulation(It lo, It hi) {
112     int n = hi - lo;
113     std::vector<double> x, y, z;
114     for (It it = lo; it != hi; ++it) {
115         x.push_back(it->x);
116         y.push_back(it->y);
117         z.push_back(sqnorm(*it));
118     }
119     std::vector<triangle> res;
120     for (int i = 0; i < n - 2; i++) {
121         for (int j = i + 1; j < n; j++) {
122             for (int k = i + 1; k < n; k++) {
123                 if (j == k) {
124                     continue;
125                 }
126                 double nx = (y[j] - y[i])*(z[k] - z[i]) - (y[k] - y[i])*(z[j] - z[i]);
127                 double ny = (x[k] - x[i])*(z[j] - z[i]) - (x[j] - x[i])*(z[k] - z[i]);
128                 double nz = (x[j] - x[i])*(y[k] - y[i]) - (x[k] - x[i])*(y[j] - y[i]);
129                 if (LE(0, nz)) {
130                     continue;
131                 }
132                 point s1[] = {lo[i], lo[j], lo[k], lo[i]};
133                 for (int m = 0; m < n; m++) {
134                     if (nx*(x[m] - x[i]) + ny*(y[m] - y[i]) + nz*(z[m] - z[i]) > 0) {
135                         goto skip;
136                     }
137                 }
138                 // Handle four points on a circle.
139                 for (int t = 0; t < (int)res.size(); t++) {
140                     point s2[] = {res[t].a, res[t].b, res[t].c, res[t].a};
141                     for (int u = 0; u < 3; u++) {
142                         for (int v = 0; v < 3; v++) {
143                             if (seg_intersection(s1[u], s1[u + 1], s2[v], s2[v + 1]) == 0) {
144                                 goto skip;
145                             }
146                         }
147                     }
148                 }
149                 res.push_back(triangle(lo[i], lo[j], lo[k]));
150                 skip:;
151             }
152         }
153     }
154     return res;
155 }
156
157 /** Example Usage **/
158
159 #include <cassert>
160 using namespace std;
161
162 int main() {

```

```

163     vector<point> v;
164     v.push_back(point(1, 3));
165     v.push_back(point(1, 2));
166     v.push_back(point(2, 1));
167     v.push_back(point(0, 0));
168     v.push_back(point(-1, 3));
169     vector<triangle> t;
170     t.push_back(triangle(point(1, 3), point(1, 2), point(-1, 3)));
171     t.push_back(triangle(point(1, 3), point(2, 1), point(1, 2)));
172     t.push_back(triangle(point(1, 2), point(2, 1), point(0, 0)));
173     t.push_back(triangle(point(1, 2), point(0, 0), point(-1, 3)));
174     assert(delaunay_triangulation(v.begin(), v.end()) == t);
175     return 0;
176 }
```

6.4.4 Delaunay Triangulation (Fast)

```

1  /*
2
3 Given a set P of two dimensional points, the Delaunay triangulation of P is a
4 set of non-overlapping triangles that covers the entire convex hull of P such
5 that no point in P lies within the circumcircle of any of the resulting
6 triangles. For any point p in the convex hull of P (but not necessarily in P),
7 the nearest point is guaranteed to be a vertex of the enclosing triangle from
8 the triangulation.
9
10 The triangulation may not exist (e.g. for a set of collinear points), or may not
11 be unique if it does exists. The following program assumes its existence and
12 produces one such valid result using TABLE_DELAUNAY, a divide and conquer
13 algorithm with linear merging. Its fully documented version along with debugging
14 messages for the current asserts() may be found at the following link:
15 http://people.sc.fsu.edu/~jburkardt/f_src/table_delaunay/table_delaunay.html
16
17 - delaunay_triangulation(lo, hi) returns a Delaunay triangulation for the input
18 range [lo, hi) of points, where lo and hi must be random-access iterators, or
19 an empty vector if a triangulation does not exist.
20
21 Time Complexity:
22 - O(n log n) per call to delaunay_triangulation(lo, hi), where n is the distance
23 between lo and hi.
24
25 Space Complexity:
26 - O(n) auxiliary heap space for storage of the Delaunay triangulation.
27
28 */
29
30 #include <algorithm>
31 #include <cassert>
32 #include <cmath>
33 #include <cstddef>
34 #include <limits>
35 #include <utility>
36 #include <vector>
37
38 int wrap(int ival, int ilo, int ihi) {
```

```

39     int jlo = std::min(ilo, ihi), jhi = std::max(ilo, ihi);
40     int wide = jhi + 1 - jlo, res = jlo;
41     if (wide != 1) {
42         assert(wide != 0);
43         int tmp = (ival - jlo) % wide;
44         if (tmp < 0) {
45             res += std::abs(wide);
46         }
47         res += tmp;
48     }
49     return res;
50 }
51
52 void permute(int n, double a[][2], int p[]) {
53     for (int istart = 1; istart <= n; istart++) {
54         if (p[istart - 1] < 0)
55             continue;
56         if (p[istart - 1] == istart) {
57             p[istart - 1] = -p[istart - 1];
58             continue;
59         }
60         double tmp0 = a[istart - 1][0], tmp1 = a[istart - 1][1];
61         int igit = istart;
62         for (;;) {
63             int iput = igit;
64             igit = p[igit - 1];
65             p[iput - 1] = -p[iput - 1];
66             assert(!(igit < 1 || n < igit));
67             if (igit == istart) {
68                 a[iput - 1][0] = tmp0;
69                 a[iput - 1][1] = tmp1;
70                 break;
71             }
72             a[iput - 1][0] = a[igit - 1][0];
73             a[iput - 1][1] = a[igit - 1][1];
74         }
75     }
76 }
77 for (int i = 0; i < n; i++) {
78     p[i] = -p[i];
79 }
80 }
81
82 int* sort_heap(int n, double a[][2]) {
83     double aval[2];
84     int i, ir, j, l, idxt;
85     int *idx;
86     if (n < 1)
87         return NULL;
88     if (n == 1) {
89         idx = new int[1];
90         idx[0] = 1;
91         return idx;
92     }
93     idx = new int[n];
94     for (int i = 0; i < n; i++) {
95         idx[i] = i + 1;
96     }
97 }
```

```

98     l = n/2 + 1;
99     ir = n;
100    for (;;) {
101        if (l < 1) {
102            l--;
103            idxt = idx[l - 1];
104            aval[0] = a[idxt - 1][0];
105            aval[1] = a[idxt - 1][1];
106        } else {
107            idxt = idx[ir - 1];
108            aval[0] = a[idxt - 1][0];
109            aval[1] = a[idxt - 1][1];
110            idx[ir - 1] = idx[0];
111            if (--ir == 1) {
112                idx[0] = idxt;
113                break;
114            }
115        }
116        i = l;
117        j = 2*l;
118        while (j <= ir) {
119            if (j < ir && (a[idx[j - 1] - 1][0] < a[idx[j] - 1][0] ||
120                            (a[idx[j - 1] - 1][0] == a[idx[j] - 1][0] &&
121                             a[idx[j - 1] - 1][1] < a[idx[j] - 1][1]))) {
122                j++;
123            }
124            if (aval[0] < a[idx[j - 1] - 1][0] ||
125                (aval[0] == a[idx[j - 1] - 1][0] &&
126                 aval[1] < a[idx[j - 1] - 1][1])) {
127                idx[i - 1] = idx[j - 1];
128                i = j;
129                j *= 2;
130            } else {
131                j = ir + 1;
132            }
133        }
134        idx[i - 1] = idxt;
135    }
136    return idx;
137 }
138
139 int lrline(double xu, double yu, double xv1, double yv1,
140             double xv2, double yv2, double dv) {
141     static const double tol = 1e-7;
142     double dx = xv2 - xv1, dy = yv2 - yv1;
143     double dxu = xu - xv1, dyu = yu - yv1;
144     double t = dy*dxu - dx*dyu + dv*sqrt(dx*dx + dy*dy);
145     double tolabs = tol*std::max(std::max(fabs(dx), fabs(dy)),
146                                 std::max(fabs(dxu), std::max(fabs(dyu), fabs(dv))));
147     return tolabs < t ? 1 : (-tolabs <= t ? 0 : -1);
148 }
149
150 void vbedg(double x, double y, int point_num, double point_xy[][2],
151             int tri_num, int tri_nodes[][][3], int tri_neigh[][][3],
152             int *ltri, int *ledg, int *rtri, int *redg) {
153     int a, b;
154     double ax, ay, bx, by;
155     bool done;
156     int e, l, t;

```

```

157  if (*ltri == 0) {
158      done = false;
159      *ltri = *rtri;
160      *ledg = *redg;
161  } else {
162      done = true;
163  }
164  for (;;) {
165      l = -tri_neigh[*rtri - 1][*redg - 1];
166      t = l / 3;
167      e = l % 3 + 1;
168      a = tri_nodes[t - 1][e - 1];
169      if (e <= 2) {
170          b = tri_nodes[t - 1][e];
171      } else {
172          b = tri_nodes[t - 1][0];
173      }
174      ax = point_xy[a - 1][0];
175      ay = point_xy[a - 1][1];
176      bx = point_xy[b - 1][0];
177      by = point_xy[b - 1][1];
178      if (lrline(x, y, ax, ay, bx, by, 0.0) <= 0) {
179          break;
180      }
181      *rtri = t;
182      *redg = e;
183  }
184  if (done) {
185      return;
186  }
187  t = *ltri;
188  e = *ledg;
189  for (;;) {
190      b = tri_nodes[t - 1][e - 1];
191      e = wrap(e - 1, 1, 3);
192      while (0 < tri_neigh[t - 1][e - 1]) {
193          t = tri_neigh[t - 1][e - 1];
194          if (tri_nodes[t - 1][0] == b) {
195              e = 3;
196          } else if (tri_nodes[t - 1][1] == b) {
197              e = 1;
198          } else {
199              e = 2;
200          }
201      }
202      a = tri_nodes[t - 1][e - 1];
203      ax = point_xy[a - 1][0];
204      ay = point_xy[a - 1][1];
205      bx = point_xy[b - 1][0];
206      by = point_xy[b - 1][1];
207      if (lrline(x, y, ax, ay, bx, by, 0.0) <= 0) {
208          break;
209      }
210  }
211  *ltri = t;
212  *ledg = e;
213  return;
214 }
215

```

```

216 int diaedg(double x0, double y0, double x1, double y1,
217             double x2, double y2, double x3, double y3) {
218     double ca, cb, s, tol, tola, tolb;
219     int value;
220     tol = 100.0*std::numeric_limits<double>::epsilon();
221     double dx10 = x1 - x0, dy10 = y1 - y0;
222     double dx12 = x1 - x2, dy12 = y1 - y2;
223     double dx30 = x3 - x0, dy30 = y3 - y0;
224     double dx32 = x3 - x2, dy32 = y3 - y2;
225     tola = tol*std::max(std::max(fabs(dx10), fabs(dy10)),
226                          std::max(fabs(dx30), fabs(dy30)));
227     tolb = tol*std::max(std::max(fabs(dx12), fabs(dy12)),
228                          std::max(fabs(dx32), fabs(dy32)));
229     ca = dx10*dx30 + dy10*dy30;
230     cb = dx12*dx32 + dy12*dy32;
231     if (tola < ca && tolb < cb) {
232         value = -1;
233     } else if (ca < -tola && cb < -tolb) {
234         value = 1;
235     } else {
236         tola = std::max(tola, tolb);
237         s = (dx10*dy30 - dx30*dy10)*cb + (dx32*dy12 - dx12*dy32)*ca;
238         if (tola < s) {
239             value = -1;
240         } else if (s < -tola) {
241             value = 1;
242         } else {
243             value = 0;
244         }
245     }
246     return value;
247 }
248
249 int swapec(int i, int *top, int *btri, int *bedg, int point_num,
250             double point_xy[] [2], int tri_num, int tri_nodes[] [3],
251             int tri_neigh[] [3], int stack[]) {
252     int a, b, c, e, ee, em1, ep1, f, fm1, fp1, l, r, s, swap, t, tt, u;
253     double x = point_xy[i - 1][0], y = point_xy[i - 1][1];
254     for (;;) {
255         if (*top <= 0) {
256             break;
257         }
258         t = stack[*top - 1];
259         *top -= 1;
260         if (tri_nodes[t - 1][0] == i) {
261             e = 2;
262             b = tri_nodes[t - 1][2];
263         } else if (tri_nodes[t - 1][1] == i) {
264             e = 3;
265             b = tri_nodes[t - 1][0];
266         } else {
267             e = 1;
268             b = tri_nodes[t - 1][1];
269         }
270         a = tri_nodes[t - 1][e - 1];
271         u = tri_neigh[t - 1][e - 1];
272         if (tri_neigh[u - 1][0] == t) {
273             f = 1;
274             c = tri_nodes[u - 1][2];

```

```

275 } else if (tri_neigh[u - 1][1] == t) {
276     f = 2;
277     c = tri_nodes[u - 1][0];
278 } else {
279     f = 3;
280     c = tri_nodes[u - 1][1];
281 }
282 swap = diaedg(x, y, point_xy[a - 1][0], point_xy[a - 1][1],
283                 point_xy[c - 1][0], point_xy[c - 1][1],
284                 point_xy[b - 1][0], point_xy[b - 1][1]);
285 if (swap == 1) {
286     em1 = wrap(e - 1, 1, 3);
287     ep1 = wrap(e + 1, 1, 3);
288     fm1 = wrap(f - 1, 1, 3);
289     fp1 = wrap(f + 1, 1, 3);
290     tri_nodes[t - 1][ep1 - 1] = c;
291     tri_nodes[u - 1][fp1 - 1] = i;
292     r = tri_neigh[t - 1][ep1 - 1];
293     s = tri_neigh[u - 1][fp1 - 1];
294     tri_neigh[t - 1][ep1 - 1] = u;
295     tri_neigh[u - 1][fp1 - 1] = t;
296     tri_neigh[t - 1][e - 1] = s;
297     tri_neigh[u - 1][f - 1] = r;
298     if (0 < tri_neigh[u - 1][fm1 - 1]) {
299         *top += 1;
300         stack[*top - 1] = u;
301     }
302     if (0 < s) {
303         if (tri_neigh[s - 1][0] == u) {
304             tri_neigh[s - 1][0] = t;
305         } else if (tri_neigh[s - 1][1] == u) {
306             tri_neigh[s - 1][1] = t;
307         } else {
308             tri_neigh[s - 1][2] = t;
309         }
310         *top += 1;
311         if (point_num < *top) {
312             return 8;
313         }
314         stack[*top - 1] = t;
315     } else {
316         if (u == *btri && fp1 == *bedg) {
317             *btri = t;
318             *bedg = e;
319         }
320         l = - (3*t + e - 1);
321         tt = t;
322         ee = em1;
323         while (0 < tri_neigh[tt - 1][ee - 1]) {
324             tt = tri_neigh[tt - 1][ee - 1];
325             if (tri_nodes[tt - 1][0] == a) {
326                 ee = 3;
327             } else if (tri_nodes[tt - 1][1] == a) {
328                 ee = 1;
329             } else {
330                 ee = 2;
331             }
332         }
333         tri_neigh[tt - 1][ee - 1] = l;

```

```

334     }
335     if (0 < r) {
336         if (tri_neigh[r - 1][0] == t) {
337             tri_neigh[r - 1][0] = u;
338         } else if (tri_neigh[r - 1][1] == t) {
339             tri_neigh[r - 1][1] = u;
340         } else {
341             tri_neigh[r - 1][2] = u;
342         }
343     } else {
344         if (t == *btri && ep1 == *bedg) {
345             *btri = u;
346             *bedg = f;
347         }
348         l = -(3*u + f - 1);
349         tt = u;
350         ee = fm1;
351         while (0 < tri_neigh[tt - 1][ee - 1]) {
352             tt = tri_neigh[tt - 1][ee - 1];
353             if (tri_nodes[tt - 1][0] == b) {
354                 ee = 3;
355             } else if (tri_nodes[tt - 1][1] == b) {
356                 ee = 1;
357             } else {
358                 ee = 2;
359             }
360         }
361         tri_neigh[tt - 1][ee - 1] = l;
362     }
363 }
364 }
365 return 0;
366 }
367
368 void perm_inv(int n, int p[]) {
369     int i, i0, i1, i2;
370     assert(n > 0);
371     for (i = 1; i <= n; i++) {
372         i1 = p[i - 1];
373         while (i < i1) {
374             i2 = p[i1 - 1];
375             p[i1 - 1] = -i2;
376             i1 = i2;
377         }
378         p[i - 1] = -p[i - 1];
379     }
380     for (i = 1; i <= n; i++) {
381         i1 = -p[i - 1];
382         if (0 <= i1) {
383             i0 = i;
384             for (;;) {
385                 i2 = p[i1 - 1];
386                 p[i1 - 1] = i0;
387                 if (i2 < 0) {
388                     break;
389                 }
390                 i0 = i1;
391                 i1 = i2;
392             }
}

```

```

393     }
394   }
395 }
396
397 int dtris2(int point_num, double point_xy[] [2],
398             int tri_nodes[] [3], int tri_neigh[] [3]) {
399   double cmax;
400   int e, error;
401   int i, j, k, l, m, m1, m2, n;
402   int ledg, lr, ltri, redg, rtri, t, top;
403   double tol;
404   int *stack = new int[point_num];
405   tol = 100.0*std::numeric_limits<double>::epsilon();
406   int *idx = sort_heap(point_num, point_xy);
407   permute(point_num, point_xy, idx);
408   m1 = 0;
409   for (i = 1; i < point_num; i++) {
410     m = m1;
411     m1 = i;
412     k = -1;
413     for (j = 0; j <= 1; j++) {
414       cmax = std::max(fabs(point_xy[m][j]), fabs(point_xy[m1][j]));
415       if (tol*(cmax + 1.0) < fabs(point_xy[m][j] - point_xy[m1][j])) {
416         k = j;
417         break;
418       }
419     }
420     assert(k != -1);
421   }
422   m1 = 1;
423   m2 = 2;
424   j = 3;
425   for (;;) {
426     assert(point_num >= j);
427     m = j;
428     lr = lrline(point_xy[m - 1][0], point_xy[m - 1][1],
429                 point_xy[m1 - 1][0], point_xy[m1 - 1][1],
430                 point_xy[m2 - 1][0], point_xy[m2 - 1][1], 0.0);
431     if (lr != 0) {
432       break;
433     }
434     j++;
435   }
436   int tri_num = j - 2;
437   if (lr == -1) {
438     tri_nodes[0][0] = m1;
439     tri_nodes[0][1] = m2;
440     tri_nodes[0][2] = m;
441     tri_neigh[0][2] = -3;
442     for (i = 2; i <= tri_num; i++) {
443       m1 = m2;
444       m2 = i + 1;
445       tri_nodes[i - 1][0] = m1;
446       tri_nodes[i - 1][1] = m2;
447       tri_nodes[i - 1][2] = m;
448       tri_neigh[i - 1][0] = -3*i;
449       tri_neigh[i - 1][1] = i;
450       tri_neigh[i - 1][2] = i - 1;
451     }

```

```

452     tri_neigh[tri_num - 1][0] = -3*tri_num - 1;
453     tri_neigh[tri_num - 1][1] = -5;
454     ledg = 2;
455     ltri = tri_num;
456 } else {
457     tri_nodes[0][0] = m2;
458     tri_nodes[0][1] = m1;
459     tri_nodes[0][2] = m;
460     tri_neigh[0][0] = -4;
461     for (i = 2; i <= tri_num; i++) {
462         m1 = m2;
463         m2 = i+1;
464         tri_nodes[i - 1][0] = m2;
465         tri_nodes[i - 1][1] = m1;
466         tri_nodes[i - 1][2] = m;
467         tri_neigh[i - 2][2] = i;
468         tri_neigh[i - 1][0] = -3*i - 3;
469         tri_neigh[i - 1][1] = i - 1;
470     }
471     tri_neigh[tri_num - 1][2] = -3*(tri_num);
472     tri_neigh[0][1] = -3*(tri_num) - 2;
473     ledg = 2;
474     ltri = 1;
475 }
476 top = 0;
477 for (i = j + 1; i <= point_num; i++) {
478     m = i;
479     m1 = tri_nodes[ltri - 1][ledg - 1];
480     if (ledg <= 2) {
481         m2 = tri_nodes[ltri - 1][ledg];
482     } else {
483         m2 = tri_nodes[ltri - 1][0];
484     }
485     lr = lrline(point_xy[m - 1][0], point_xy[m - 1][1],
486                 point_xy[m1 - 1][0], point_xy[m1 - 1][1],
487                 point_xy[m2 - 1][0], point_xy[m2 - 1][1], 0.0);
488     if (0 < lr) {
489         rtri = ltri;
490         redg = ledg;
491         ltri = 0;
492     } else {
493         l = -tri_neigh[ltri - 1][ledg - 1];
494         rtri = l / 3;
495         redg = (l % 3) + 1;
496     }
497     vbedg(point_xy[m - 1][0], point_xy[m - 1][1],
498           point_num, point_xy, tri_num, tri_nodes, tri_neigh,
499           &ltri, &ledg, &rtri, &redg);
500     n = tri_num + 1;
501     l = -tri_neigh[ltri - 1][ledg - 1];
502     for (;;) {
503         t = l / 3;
504         e = (l % 3) + 1;
505         l = -tri_neigh[t - 1][e - 1];
506         m2 = tri_nodes[t - 1][e - 1];
507         if (e <= 2) {
508             m1 = tri_nodes[t - 1][e];
509         } else {
510             m1 = tri_nodes[t - 1][0];

```

```

511     }
512     tri_num++;
513     tri_neigh[t - 1][e - 1] = tri_num;
514     tri_nodes[tri_num - 1][0] = m1;
515     tri_nodes[tri_num - 1][1] = m2;
516     tri_nodes[tri_num - 1][2] = m;
517     tri_neigh[tri_num - 1][0] = t;
518     tri_neigh[tri_num - 1][1] = tri_num - 1;
519     tri_neigh[tri_num - 1][2] = tri_num + 1;
520     top++;
521     assert(point_num >= top);
522     stack[top - 1] = tri_num;
523     if (t == rtri && e == redg) {
524         break;
525     }
526 }
527 tri_neigh[ltri - 1][ledg - 1] = -3*n - 1;
528 tri_neigh[n - 1][1] = -3*tri_num - 2;
529 tri_neigh[tri_num - 1][2] = -1;
530 ltri = n;
531 ledg = 2;
532 error = swapec(m, &top, &ltri, &ledg, point_num, point_xy,
533                 tri_num, tri_nodes, tri_neigh, stack);
534 assert(error == 0);
535 }
536 for (i = 0; i < 3; i++) {
537     for (j = 0; j < tri_num; j++) {
538         tri_nodes[j][i] = idx[tri_nodes[j][i] - 1];
539     }
540 }
541 perm_inv(point_num, idx);
542 permute(point_num, point_xy, idx);
543 delete[] idx;
544 delete[] stack;
545 return tri_num;
546 }

547 /**
548 *** Wrapper ***
549
550 const double EPS = 1e-9;
551 #define EQ(a, b) (fabs((a) - (b)) <= EPS)
552
553 typedef std::pair<double, double> point;
554 #define x first
555 #define y second
556
557 struct triangle {
558     point a, b, c;
559
560     triangle(const point &a, const point &b, const point &c) : a(a), b(b), c(c) {}
561
562     bool operator==(const triangle &t) const {
563         return EQ(a.x, t.a.x) && EQ(a.y, t.a.y) &&
564             EQ(b.x, t.b.x) && EQ(b.y, t.b.y) &&
565             EQ(c.x, t.c.x) && EQ(c.y, t.c.y);
566     }
567 };
568
569 template<class It>

```

```
570 std::vector<triangle> delaunay_triangulation(It lo, It hi) {
571     int n = hi - lo;
572     double points[n][2];
573     int tri_nodes[3*n][3], tri_neigh[3*n][3];
574     int curr = 0;
575     for (It it = lo; it != hi; ++curr, ++it) {
576         points[curr][0] = it->x;
577         points[curr][1] = it->y;
578     }
579     int m = dtris2(n, points, tri_nodes, tri_neigh);
580     std::vector<triangle> res;
581     for (int i = 0; i < m; i++) {
582         res.push_back(triangle(lo[tri_nodes[i][0] - 1],
583                             lo[tri_nodes[i][1] - 1],
584                             lo[tri_nodes[i][2] - 1]));
585     }
586     return res;
587 }
588
589 /** Example Usage **/
590
591 #include <cassert>
592 using namespace std;
593
594 int main() {
595     vector<point> v;
596     v.push_back(point(1, 3));
597     v.push_back(point(1, 2));
598     v.push_back(point(2, 1));
599     v.push_back(point(0, 0));
600     v.push_back(point(-1, 3));
601     vector<triangle> t;
602     t.push_back(triangle(point(-1, 3), point(0, 0), point(1, 2)));
603     t.push_back(triangle(point(-1, 3), point(1, 2), point(1, 3)));
604     t.push_back(triangle(point(1, 2), point(0, 0), point(2, 1)));
605     t.push_back(triangle(point(1, 3), point(1, 2), point(2, 1)));
606     assert(delaunay_triangulation(v.begin(), v.end()) == t);
607     return 0;
608 }
```